

DISCOVER

METEOR

Building Real-Time JavaScript Web Apps

TOM COLEMAN & SACHA GREIF

Versión 1.9 (updated 27 de junio de 2015)

Cover photo credit: **Perseid Hunting** by Darren Blackburn, licensed under a Creative Commons Attribution 2.0 Generic license.

www.discovermeteor.com

Hagamos un pequeño experimento mental. Imaginemos que abrimos dos ventanas del explorador de archivos de nuestro ordenador mostrando la misma carpeta.

Ahora borramos un archivo en una de las dos ventanas. ¿Habrá desaparecido en la otra?

Cuando modificamos algo en nuestro sistema de archivos local, el cambio se aplica en todas partes sin necesidad de refrescos o **callbacks**. Simplemente sucede.

Ahora, vamos a pensar qué pasaría en la web en esta misma situación. Por ejemplo, digamos que abrimos el mismo WordPress en dos ventanas del navegador y creamos un post en una de ellas. A diferencia del escritorio, la otra ventana no reflejará el cambio a menos que la recargues.

Nos hemos acostumbrado a la idea de que un sitio web es algo con lo que solo te comunicas como a ráfagas separadas.

Meteor es parte de una nueva ola de frameworks y tecnologías que buscan desafiar el statu quo haciendo webs reactivas y en tiempo real.

Meteor es una plataforma para crear aplicaciones web en tiempo real construida sobre Node.js. Meteor se localiza entre la base de datos de la aplicación y su interfaz de usuario y se encarga que las dos partes estén sincronizadas.

Como Meteor usa Node.js, se utiliza JavaScript en el cliente y en el servidor. Y más aún, Meteor es capaz de compartir código entre ambos entornos.

El resultado es una plataforma muy potente y muy sencilla ya que Meteor abstracta muchas de las molestias y dificultades que nos encontramos habitualmente en el desarrollo de aplicaciones web.

¿Por qué dedicar tiempo a aprender Meteor en lugar de cualquier otro framework web? Dejando a un lado las características de Meteor, creemos que todo se reduce a una sola cosa: Meteor es “full stack” y es fácil de aprender.

Meteor permite crear una aplicación web en tiempo real en cuestión de horas. Y si ya hemos hecho desarrollo web, estaremos familiarizados con JavaScript, y ni siquiera tendremos que aprender un nuevo lenguaje.

Meteor podría ser la plataforma ideal para nuestras necesidades, o, quizás no. Pero ¿por qué no probarlo y descubrirlo por nosotros mismos?

Durante los últimos años, hemos estado trabajando en numerosos proyectos con Meteor, desde aplicaciones web hasta aplicaciones móviles, y desde proyectos comerciales hasta proyectos de código abierto.

Hemos aprendido un montón, pero no siempre ha sido fácil encontrar respuestas a todas nuestras preguntas. Tuvimos que encajar piezas de muchas fuentes diferentes, y en muchos casos incluso inventamos nuestras propias soluciones. Con este libro, queremos compartir todas estas lecciones, y crear una sencilla guía para construir una aplicación desde cero con Meteor.

La aplicación que construiremos es una versión simplificada de una red social como [Hacker News](#) o [Reddit](#), a la que llamaremos Microscope (por analogía con su hermana mayor, la aplicación de código abierto, [Telescope](#)). Durante su construcción, veremos todos los elementos que intervienen en una aplicación desarrollada con Meteor, tales como cuentas de usuario, colecciones, enrutamiento, y mucho más.

Uno de nuestros objetivos al escribir este libro es mantener las cosas accesibles y fáciles de entender, así que cualquiera debería ser capaz de seguirlo, aunque no tenga experiencia con Meteor, Node.js, o frameworks MVC, o incluso con la programación en general en el lado del servidor.

Por otro lado, se asume cierta familiaridad con los conceptos y la sintaxis básica de JavaScript. Si alguna vez has hackeado algo de código jQuery o jugado un poco con la consola de desarrollo del navegador, verás como no tendrás problemas en seguirlo.

Si aún no te sientes cómodo usando JavaScript, te sugerimos que eches un vistazo a nuestra entrada [JavaScript primer for Meteor](#) de nuestro blog, antes de seguir con el libro.

Si te estás preguntando quienes somos y por qué deberías confiar en nosotros, a continuación tienes algo más de información sobre nosotros dos.

Tom Coleman forma parte de [Percolate Studio](#), una tienda de desarrollo web centrada en la calidad y la experiencia de usuario. Además, es uno de los mantenedores del repositorio de paquetes [Atmosphere](#), y está detrás de otros proyectos dentro de Meteor (como el [Iron Router](#)).

Sacha Greif ha trabajado como diseñador en startups como [Hipmunk](#) y [Ruby Motion](#). Es el creador de [Telescope](#) y [Sidebar](#) (basada en Telescope), y es también el fundador de [Folio](#).

Para que este libro sea de utilidad tanto para el principiante como para el programador avanzado, sus capítulos están divididos en dos categorías: los capítulos normales (numerados del 1 al 14) y las barras laterales o sidebars (números .5).

Los capítulos normales son la guía para construir la aplicación, y su objetivo es conseguir que funcione de la forma más rápida posible, explicando los pasos más importantes sin entrar en demasiados detalles.

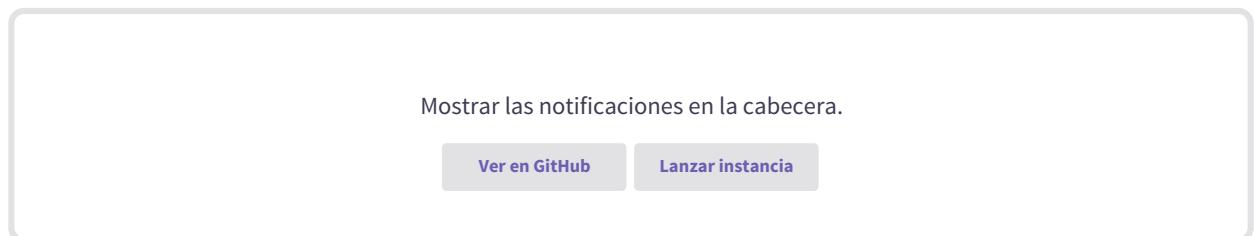
Por otro lado, las barras laterales profundizan en los entresijos de Meteor, y nos ayudarán a comprender mejor lo que realmente ocurre entre bastidores.

Así que, si nos consideramos principiantes, deberíamos de saltarnos las barras laterales en una primera lectura, y volver a

ellas más tarde una vez que hayamos jugado un poco con Meteor.

No hay nada peor que estar siguiendo un libro de programación y de repente darnos cuenta de que nuestro código se ha roto y que nada funciona como debería.

Para evitarlo, hemos creado un [repositorio en GitHub para Microscope](#), ofreciendo enlaces a commits de git cada pocos cambios de código. Además, cada commit se enlaza con una instancia on-line de la aplicación, por lo que se puede comparar con nuestra copia local. He aquí un ejemplo de lo que podrás ver:



Solo una cosa, ten en cuenta que el hecho de que ofrezcamos estos commits, no significa que tengas que ir de un checkout al siguiente. ¡Aprenderás mucho más si dedicas el tiempo necesario a escribir el código de tu aplicación!

Si quieras aprender más acerca de un aspecto particular de Meteor, la [documentación oficial de Meteor](#) es el mejor sitio al que ir para empezar.

También te recomendamos [Stack Overflow](#) para solucionar problemas y dudas, y el [canal IRC #meteor](#) si necesitas ayuda directa.



-
- Si deseas ponerte en contacto con nosotros, puedes enviarnos un correo electrónico a hello@discovermeteor.com.
 - Además, si encuentras un error tipográfico o cualquier otro error en el contenido del libro, puedes [reportarlo en este repositorio de GitHub](#).
 - Si encuentras un problema en el código de Microscope, puedes enviarlo al [repositorio de Microscope](#).

- Por último, para cualquier otra pregunta, puedes dejarnos un comentario en el panel lateral de esta aplicación.

Las primeras impresiones son las que cuentan. La instalación de Meteor debería ser muy sencilla y, en la mayoría de los casos, sólo cuesta 5 minutos ponerlo en marcha.

Para empezar, si estamos usando Mac OS o GNU/Linux, podemos instalar Meteor con el siguiente comando desde la consola:

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

Si estás usando Windows, echa un vistazo a la guía oficial de instalación:[install instructions](#) en la web de Meteor.

Se instalará el ejecutable `meteor` en nuestro sistema y lo dejará listo para empezar a usar Meteor.

Sin

Si no podemos (o no queremos) instalar Meteor de forma local, recomendamos usar[Nitrous.io](#).

Nitrous.io es un servicio que te permite ejecutar aplicaciones y editar el código directamente en tu navegador, y hemos escrito una breve guía para ayudarte a ponerte en marcha.

Sólo tienes que seguir [esta guía](#) hasta completar la sección “Installing Meteor”, y luego seguir con este capítulo a partir de la sección “Crear y ejecutar una aplicación”.

Ahora que tenemos instalado Meteor, vamos a crear nuestra aplicación. Para ello, utilizaremos la herramienta de línea de comandos `meteor`:

```
meteor create microscope
```

Este comando crea un proyecto básico listo para usar. Cuando termina, deberíamos ver un directorio llamado `microscope/`, que contiene lo siguiente:

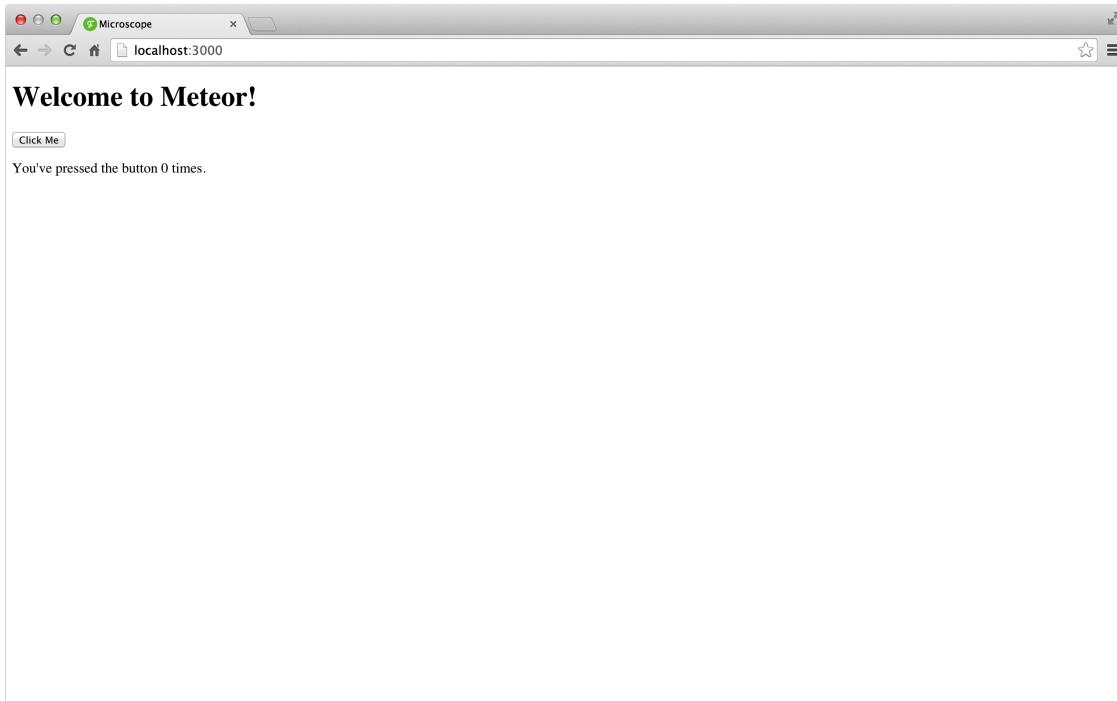
```
.meteor  
microscope.css  
microscope.html  
microscope.js
```

La aplicación que se ha creado es una aplicación básica que demuestra sólo algunas sencillas pautas.

A pesar de que nuestra aplicación no hace casi nada, ya podemos ejecutarla. Para hacerlo, volvemos al terminal y escribimos:

```
cd microscope  
meteor
```

Ahora abrimos `http://localhost:3000/` (o su equivalente `http://0.0.0.0:3000/`) en el navegador y deberíamos ver algo como esto:



Un proyecto básico.

[Ver en GitHub](#)

[Lanzar instancia](#)

¡Enhorabuena! ya tenemos nuestra primera aplicación Meteor funcionando. Por cierto, para parar la aplicación, todo lo que hay que hacer es abrir la pestaña terminal donde se ejecuta y pulsar `ctrl+c`.

Si estás utilizando Git, este sería un buen momento para iniciar el repositorio con `git init`.

Hubo un tiempo en el que Meteor utilizaba un gestor de paquetes externo llamado Meteorite. Desde la versión 0.9.0 de Meteor, Meteorite ya no es necesario, ya que sus características se han incorporado a Meteor.

Así que si encuentras referencias a `mrt` de Meteorite a lo largo de este libro o mientras lees material relacionado con Meteor, puedes reemplazarlo con total seguridad por el habitual `meteor`.

Ahora vamos a usar el sistema de paquetes de Meteor para incluir el framework **Bootstrap** en nuestro proyecto:

Esto no es distinto de añadir Bootstrap de la forma habitual, incluyendo manualmente los ficheros CSS y JavaScript, excepto en que confiamos en el mantenedor del paquete para que lo mantenga actualizado para nosotros.

Ya que estamos, añadiremos también el paquete **Underscore**. Underscore es una librería de utilidades JavaScript, y es muy útil cuando necesitemos manipular estructuras de datos.

El paquete `bootstrap` lo mantiene el usuario `twbs`, por lo que el nombre completo del paquete es `twbs:bootstrap`.

El paquete `underscore` forma parte de los paquetes “oficiales” incluidos en Meteor, lo que quiere decir que no hay que incluir el nombre del autor:

```
meteor add twbs:bootstrap
meteor add underscore
```

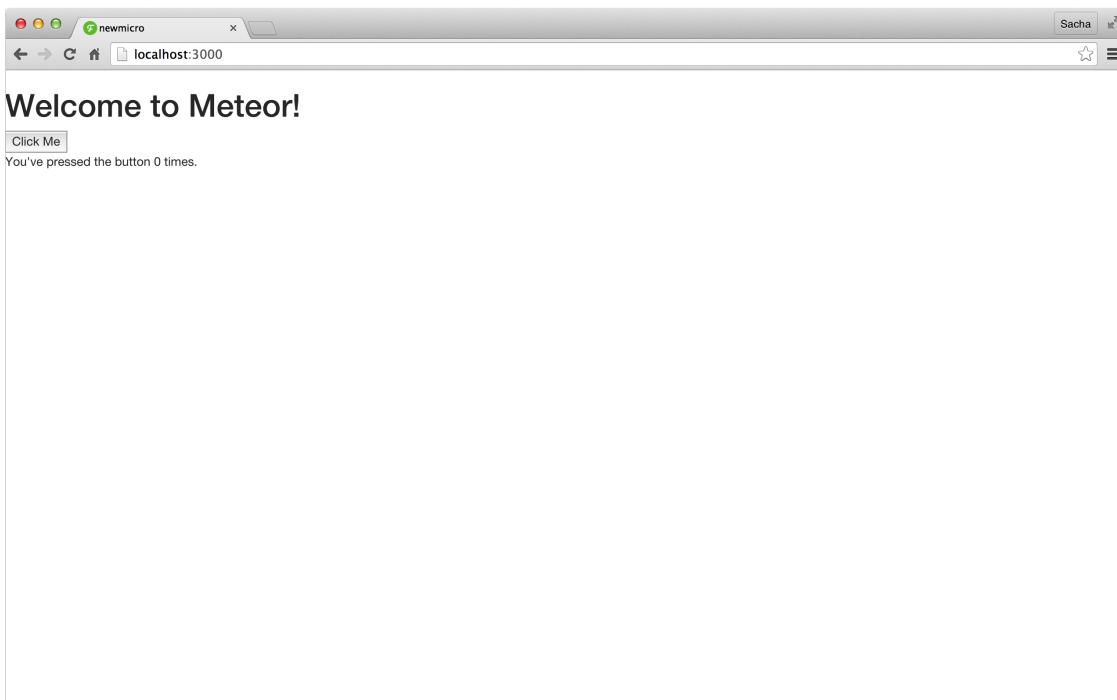
Fíjate que estamos añadiendo Bootstrap **3**. Algunas de las capturas de este libro están tomadas de una versión antigua de Microscope con Bootstrap **2**, por lo que podrían parecer ligeramente diferentes.

Añadido el paquete
bootstrap.

[Ver en GitHub](#)

[Lanzar instancia](#)

Tan pronto como agregues el paquete Bootstrap, deberías notar un cambio en el aspecto de nuestra aplicación:



Al contrario de la forma “tradicional” en la que incluimos recursos externos, no tenemos que agregar enlaces a ningún fichero CSS o JavaScript, ¡porque Meteor lo hace por nosotros! Esta es sólo una de las muchas ventajas de los paquetes Meteor.

Al hablar acerca de los paquetes en el contexto de Meteor, vale la pena ser específico. Meteor puede usar cinco tipos básicos de paquetes:

- El mismo núcleo de Meteor está dividido en diferentes **paquetes de la plataforma Meteor**. Están incluidos en cada app y probablemente nunca tengas que preocuparte por ellos.
- Los paquetes ordinarios se conocen como “**isopacks**”, o paquetes isomórficos (paquetes que funcionan en los dos lados, cliente y servidor). Los paquetes **First-party**, como `accounts-ui` o `appcache`, están mantenidos por los desarrolladores de Meteor y **vienen incluidos en Meteor**.
- Los **paquetes de terceros** son paquetes isopacks que otros usuarios han subido al servidor de paquetes de Meteor. Puedes echarles un vistazo en **Atmosphere** o con el comando `meteor search`.
- Los **paquetes locales** son paquetes personalizados que puedes crear tu mismo y colocarlos en el directorio `/packages`.
- Los **paquetes NPM** (Node.js Packaged Modules) son paquetes de Node.js. A pesar de que no funcionan por defecto con Meteor, *pueden* ser utilizados por los tipos de paquete anteriores.

Antes de empezar a escribir código debemos estructurar de forma adecuada nuestro proyecto. Para asegurarte de que dispones de un entorno limpio y claro, abre el directorio `microscope` y borra los archivos `microscope.html`, `microscope.js`, y `microscope.css`.

A continuación, crea cuatro directorios dentro de `/microscope`: `/client`, `/server`, `/public` y `/lib`.

Ahora, crearemos dos archivos vacíos `main.html` y `main.js` dentro de `/client`. Por ahora, no te preocupes si esto rompe completamente la app, empezaremos a llenar los nuevos ficheros en el siguiente capítulo.

Debemos mencionar que algunos de los directorios que hemos creado son especiales y Meteor tiene reglas para ellos:

- El código de `/server` se ejecuta en el servidor.
- El código de `/client` se ejecuta en el cliente.
- Todo lo demás se ejecuta en las dos partes, cliente y servidor.
- Las cosas estáticas (fuentes, imágenes, etc.) van en el directorio `/public`.

Y también es útil saber como Meteor decide en qué orden cargan los ficheros:

- Los archivos de `/lib` se cargan antes que nada.
- Los archivos con nombre `main.*` se cargan *después* que todos los demás.
- Todo se carga por orden alfabético según el nombre del fichero.

Ten en cuenta que aunque Meteor tiene todas estas reglas, en realidad no nos obliga a utilizar una estructura de archivos

predefinida. Así que la estructura que sugerimos es sólo nuestra forma de hacer las cosas, no son reglas inamovibles.

Os animamos a echar un vistazo a la [documentación oficial Meteor](#) para conocer más detalles acerca de la estructura de las aplicaciones.

Si hemos usado otros frameworks, como Ruby on Rails, puede que nos preguntemos si las aplicaciones de Meteor adoptan el patrón **MVC** (Model View Controller).

La respuesta corta es no. A diferencia de Rails, Meteor no impone ninguna estructura predefinida para su aplicación. Así que en este libro vamos a exponer código de la forma que más sentido tenga para nosotros, sin preocuparnos demasiado por las siglas.

Bueno, mentimos. En realidad no vamos a necesitar `public/` por la sencilla razón de que Microscope no utiliza ningún archivo estático. Pero como en la mayoría de aplicaciones se van a incluir al menos un par de imágenes, pensamos que era importante incluirlo.

Por cierto, te puedes haber dado cuenta de que se ha creado un directorio oculto llamado `.meteor`. Aquí es donde Meteor almacena su propio código. Cambiar cosas aquí dentro es, en general, una muy mala idea con las únicas excepciones de los archivos `.meteor/packages` y `.meteor/release`, que se utilizan, respectivamente, para listar nuestros paquetes y para establecer la versión de Meteor que queremos utilizar.

Lo único que vamos a decir sobre el viejo debate del guión bajo (`my_variable`) contra el camelCase (`myVariable`) es que en realidad no importa el que elijas, siempre y cuando lo adoptes en todo el proyecto.

En este libro, utilizamos camelCase porque es la forma habitual en JavaScript (después de todo, es JavaScript no `java_script!`).

Las únicas excepciones a esta regla son los nombres de los archivos, para los que se van a utilizar guiones bajos (`my_file.js`), y las clases CSS, para las que usaremos guiones (`.my-class`). La razón es que en el sistema de archivos, el subrayado es lo más común, mientras que en la propia sintaxis, CSS ya utiliza guiones (`font-family`, `text-align`, etc).

Este libro no trata sobre CSS. Así que para evitar entrar en detalles de estilo, hemos decidido que la hoja de estilos esté disponible desde el principio, así, no será necesario preocuparse por ella nunca más.

Meteor carga el CSS minimizado y de forma automática, por lo que, a diferencia de otros recursos estáticos, va en `/client`, no en `/public`. Vamos a crear el archivo `client/stylesheets/style.css` y a añadirle este CSS:

```
.grid-block, .main, .post, .comments li, .comment-form {
  background: #fff;
  border-radius: 3px;
  padding: 10px;
  margin-bottom: 10px;
  -webkit-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);
  -moz-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);
  box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15); }

body {
  background: #eee;
  color: #666666; }

#main {
  position: relative;
}

.page {
  position: absolute;
  top: 0px;
  width: 100%;
}

.navbar {
  margin-bottom: 10px; }
/* line 32, ../sass/style.scss */
.navbar .navbar-inner {
  border-radius: 0px 0px 3px 3px; }

#spinner {
  height: 300px; }

.post {
  /* For modern browsers */
  /* For IE 6/7 (trigger hasLayout) */
  *zoom: 1;
  position: relative;
  opacity: 1;
  .post:before, .post:after {
    content: "";
    display: table; }
  .post:after {
    clear: both; }
  .post.invisible {
    opacity: 0; }
  .post.instant {
    -webkit-transition: none;
    -moz-transition: none;
    -o-transition: none;
    transition: none; }
  .post.animate{
    -webkit-transition: all 300ms 0ms;
    -moz-transition: all 300ms 0ms ease-in;
    -o-transition: all 300ms 0ms ease-in;
    transition: all 300ms 0ms ease-in; }
  .post .upvote {
    display: block;
    margin: 7px 12px 0 0;
    float: left; }
  .post .post-content {
    float: left; }
  .post .post-content h3 {
    margin: 0;
    line-height: 1.4;
    font-size: 18px; }
  .post .post-content h3 a {
    display: inline-block;
    margin-right: 5px; }
  .post .post-content h3 span {
    font-weight: normal;
    font-size: 14px;
    display: inline-block;
    color: #aaaaaa; }
```

```
.post .post-content p {
  margin: 0; }
.post .discuss {
  display: block;
  float: right;
  margin-top: 7px; }

.comments {
  list-style-type: none;
  margin: 0; }
.comments li h4 {
  font-size: 16px;
  margin: 0; }
.comments li h4 .date {
  font-size: 12px;
  font-weight: normal; }
.comments li h4 a {
  font-size: 12px; }
.comments li p:last-child {
  margin-bottom: 0; }

.dropdown-menu span {
  display: block;
  padding: 3px 20px;
  clear: both;
  line-height: 20px;
  color: #bbb;
  white-space: nowrap; }

.load-more {
  display: block;
  border-radius: 3px;
  background: rgba(0, 0, 0, 0.05);
  text-align: center;
  height: 60px;
  line-height: 60px;
  margin-bottom: 10px; }
.load-more:hover {
  text-decoration: none;
  background: rgba(0, 0, 0, 0.1); }

.posts .spinner-container{
  position: relative;
  height: 100px;
}

.jumbotron{
  text-align: center;
}
.jumbotron h2{
  font-size: 60px;
  font-weight: 100;
}

@-webkit-keyframes fadeIn {
  0% {opacity: 0;}
  10% {opacity: 1;}
  90% {opacity: 1;}
  100% {opacity: 0;}
}

@keyframes fadeIn {
  0% {opacity: 0;}
  10% {opacity: 1;}
  90% {opacity: 1;}
  100% {opacity: 0;}
}

.errors{
  position: fixed;
  z-index: 10000;
  padding: 10px;
```

```
padding: 10px,
top: 0px;
left: 0px;
right: 0px;
bottom: 0px;
pointer-events: none;
}
.alert {
    animation: fadeOut 2700ms ease-in 0s 1 forwards;
-webkit-animation: fadeOut 2700ms ease-in 0s 1 forwards;
-moz-animation: fadeOut 2700ms ease-in 0s 1 forwards;
width: 250px;
float: right;
clear: both;
margin-bottom: 5px;
pointer-events: auto;
}
```

client/stylesheets/style.css

Estructura de ficheros
reorganizada.

[Ver en GitHub](#)

[Lanzar instancia](#)

En este libro vamos usar JavaScript puro. Pero si quisieramos usar CoffeeScript, bastaría con añadir el paquete CoffeeScript y estaríamos listos para continuar:

```
meteor add coffeescript
```

A algunos les gusta trabajar silenciosamente en un proyecto hasta que queda perfecto. Otros quieren mostrarlo al mundo lo más pronto posible.

Si eres de los primeros y prefieres desarrollar a nivel local, no dudes en saltarte este capítulo. Pero si prefieres aprender a desplegar tu aplicación Meteor en la Web, ahora te explicamos cómo hacerlo.

Vamos a aprender a desplegar una aplicación Meteor de diferentes formas. Eres libre de utilizar cualquiera de ellas en cualquier etapa del desarrollo, ya sea trabajando en Microscope o en otra aplicación. ¡Vamos a empezar!

Este capítulo es una **barra lateral** (*sidebar*). En estos capítulos se adopta una mirada más profunda a temas más generales sobre Meteor, de forma independiente al resto del libro.

Así que si prefieres seguir construyendo Microscope, puedes saltarse estos capítulos y volver a ellos más tarde.

Desplegar en un subdominio de Meteor (por ejemplo `http://myapp.meteor.com`) es la opción más sencilla, y será lo primero que probaremos. Es muy útil para mostrar la aplicación durante las primeras etapas del desarrollo o para configurar rápidamente un servidor de prueba.

Desplegar en Meteor es muy simple. Solo tienes que abrir el terminal, ir al directorio de la aplicación y escribir:

```
meteor deploy myapp.meteor.com
```

Por supuesto que tienes que tener cuidado de reemplazar “myapp” con un nombre de tu elección, y preferiblemente uno que no esté en uso.

Si es la primera vez que despliegas una aplicación, te pedirá crear una cuenta en Meteor. Y si todo va bien, después de unos segundos podrás acceder a la aplicación desde `http://myapp.meteor.com`.

Puedes mirar [la documentación oficial](#) para obtener más información sobre cosas como el acceso a la base de datos de la instancia, o cómo configurar un dominio personalizado.

Modulus es una gran opción para desplegar aplicaciones basadas en Node.js. Es uno de los pocos proveedores de PaaS (plataforma-como-servicio) que apoyan oficialmente a Meteor, y ya hay un buen número de gente corriendo aplicaciones en producción allí.

Puedes aprender más sobre Modulus, leyendo su guía de despliegue de aplicaciones Meteor:[deployment guide for](#)

Aunque todos los días aparecen nuevas soluciones en la nube, a menudo vienen con su propia cuota de problemas y limitaciones. Así que, actualmente, el despliegue en tu propio servidor sigue siendo la mejor manera de poner una aplicación Meteor en producción. El problema es que, hacerlo uno mismo no es tan sencillo, especialmente si lo que estás buscando es un despliegue de calidad.

Meteor Up (o `mup`, para abreviar) es un intento de solucionar este problema con una utilidad de línea de comandos que se encarga por nosotros de la instalación y el despliegue. Así que veamos cómo desplegar Microscope utilizando Meteor Up.

Antes de nada, vamos a necesitar un servidor. Recomendamos o bien **Digital Ocean**, desde \$5 al mes, o bien **AWS**, que proporciona Micro instancias gratuitas (con las que rápidamente tendremos problemas de escala, aunque deberían ser suficientes si solo buscamos empezar a jugar con Meteor).

Sea cual sea el servicio que elijas, debes obtener tres cosas: la dirección IP del servidor, un inicio de sesión (normalmente `root` o `ubuntu`), y una contraseña. Guarda estas cosas en un lugar seguro, ¡pronto las necesitaremos!.

Para empezar, necesitamos instalar Meteor Up vía `npm`:

```
npm install -g mup
```

A continuación, crearemos un directorio especial donde pondremos la configuración de Meteor Up para un despliegue en particular. Vamos a utilizar un directorio independiente por dos razones: primero, porque, por lo general, es la mejor manera de evitar incluir credenciales privadas en tu repositorio Git, especialmente si estás trabajando en un repositorio público.

En segundo lugar, mediante el uso de directorios separados, podremos manejar múltiples configuraciones en paralelo. Esto será muy útil para, por ejemplo, desplegar instancias de desarrollo y producción.

Así que vamos a crear este nuevo directorio y lo utilizaremos para iniciar un nuevo proyecto Meteor Up:

```
mkdir ~/microscope-deploy  
cd ~/microscope-deploy  
mup init
```

Una manera de asegurarse de que todo el equipo de desarrollo utilice la misma configuración de despliegue es crear la carpeta de configuración de Meteor Up dentro de Dropbox, o cualquier servicio similar.

Cuando inicializamos un nuevo proyecto, Meteor Up crea dos archivos: `mup.json` y `settings.json`.

`mup.json` contendrá los ajustes relacionados con el despliegue, mientras que `settings.json` contendrá todos los ajustes relacionados con la aplicación (tokens OAuth, tokens para análisis, etc.)

El siguiente paso es configurar el archivo `mup.json`. Aquí está el archivo `mup.json` que `mup init` genera por defecto. Todo lo que hay que hacer es rellenar los espacios en blanco:

```
{  
  //server authentication info  
  "servers": [{  
    "host": "hostname",  
    "username": "root",  
    "password": "password"  
    //or pem file (ssh based authentication)  
    //"/pem": "~/.ssh/id_rsa"  
  }],  
  
  //install MongoDB in the server  
  "setupMongo": true,  
  
  //location of app (local directory)  
  "app": "/path/to/the/app",  
  
  //configure environmental  
  "env": {  
    "ROOT_URL": "http://supersite.com"  
  }  
}
```

`mup.json`

Vamos a repasar cada uno de estos valores.

Autenticación del servidor

Te habrás dado cuenta de que con Meteor Up puedes usar SSH con usuario y contraseña o una clave privada (PEM), por lo que lo podemos usar con casi cualquier proveedor de la nube.

Nota importante: si eliges utilizar la autenticación basada en contraseña, asegúrate de instalar primero `sshpass`. ([Echa un vistazo a esta guía](#)).

Configuración de MongoDB

El siguiente paso es configurar una base de datos MongoDB. Recomendamos usar [Compose](#) o cualquier otro proveedor en la nube porque ofrecen un apoyo profesional y mejores herramientas de gestión.

Si has decidido usar Compose, configura `setupMongo` como `false` y añade la variable de entorno `MONGO_URL` en un bloque `env` en `mup.json`. Si por el contrario decides usar MongoDB en el propio servidor, configura `setupMongo` como `true` y Meteor Up se hará cargo de todo.

El path de la aplicación Meteor

Debido a que la configuración de Meteor Up reside en un directorio diferente. Mediante la propiedad `app` le decimos cómo llegar hasta nuestra aplicación. Sólo hay que introducir la ruta local completa, que se puede obtener con el comando `pwd` de la terminal, cuando estamos dentro del directorio de la aplicación.

Environment Variables

Dentro del bloque `env` podemos especificar todas las variables de entorno de nuestra la aplicación (por ejemplo, `ROOT_URL`, `MAIL_URL`, `MONGO_URL`, etc.).

Antes de que podamos desplegar, tendremos que configurar el servidor para que esté listo para alojar aplicaciones Meteor. La magia de Meteor Up encapsula este complejo proceso ¡en un solo comando!

```
mup setup
```

Esto llevará un tiempo dependiendo del rendimiento del servidor y la conectividad de la red. Después de que la instalación termine correctamente, por fin podemos desplegar nuestra aplicación con:

```
mup deploy
```

Esto empaqueta la aplicación, y la despliega en el servidor que acabamos de configurar.

Los registros son muy importantes y Meteor Up proporciona una forma fácil de manejarlos, emulando el comando `tail -f`. Solo tienes que escribir:

```
mup logs -f
```

Aquí termina nuestro resumen de lo que puede hacer Meteor Up. Para más información, le sugerimos visitar [el repositorio GitHub de Meteor Up](#).

Estas tres formas de desplegar aplicaciones Meteor deberían ser suficiente para la mayoría de los casos de uso. Por supuesto, sabemos que algunos de vosotros preferiríais tener el control y configurar un servidor Meteor desde cero. Eso, lo dejaremos para otro día... o tal vez para otro libro!

Para introducirnos de manera sencilla en el desarrollo con Meteor, adoptaremos un enfoque de afuera hacia adentro, es decir, primero construiremos el envoltorio exterior y luego lo conectaremos al funcionamiento interno de la aplicación.

Esto implica que, en este capítulo, solo utilizaremos el directorio `/client`.

Si todavía no lo has hecho, crea un nuevo archivo `main.html` dentro del directorio `client`, rellenándolo con el siguiente código:

```
<head>
  <title>Microscope</title>
</head>
<body>
  <div class="container">
    <header class="navbar navbar-default" role="navigation">
      <div class="navbar-header">
        <a class="navbar-brand" href="/">Microscope</a>
      </div>
    </header>
    <div id="main">
      {{> postsList}}
    </div>
  </div>
</body>
```

`client/main.html`

Esta será la plantilla principal de la aplicación. Como se puede ver, todo es HTML excepto la etiqueta `{{> postsList}}`, que es un punto de inserción de la plantilla `postsList`. Ahora, vamos a crear un par de plantillas más.

La aplicación que estamos construyendo va a ser una red social de noticias que estará compuesta de mensajes (en adelante, `posts`) organizados en listas, y así es como organizaremos nuestras plantillas.

Vamos a crear el directorio `/templates` dentro de `/client`. Aquí pondremos todas nuestras plantillas, pero además, para mantener las cosas ordenadas creamos el directorio `/posts` dentro de `/templates` para las plantillas relacionadas con los `posts`.

Meteor es bueno encontrando archivos. No importa en qué lugar pongamos el código dentro de `/client`, Meteor lo encontrará y lo compilará correctamente. Esto significa que no hay que escribir manualmente rutas para los archivos CSS o JavaScript.

También significa que se podrían poner todos los archivos en el mismo directorio, o incluso todo el código en el mismo archivo. Pero como, de todas formas, Meteor lo va a compilar todo en un solo archivo minimizado, preferimos mantener las cosas bien organizadas y utilizar una estructura de archivos lo más limpia posible.

Ya estamos listos para nuestra segunda plantilla. Dentro de `client/templates/posts`, crea el fichero

`posts_list.html`:

```
<template name="postsList">
<div class="posts">
  {{#each posts}}
    {{> postItem}}
  {{/each}}
</div>
</template>
```

client/templates/posts/posts_list.html

Y `post_item.html`:

```
<template name="postItem">
<div class="post">
  <div class="post-content">
    <h3><a href="{{url}}>{{title}}</a><span>{{domain}}</span></h3>
  </div>
</div>
</template>
```

client/templates/posts/post_item.html

Fíjate en el atributo `name="postsList"` del elemento template. Este será el nombre que Meteor usará para saber donde va cada plantilla (fíjate que el nombre del *fichero* no es importante).

Es el momento de introducir **Spacebars** el sistema de plantillas de Meteor. Spacebars es simplemente HTML mas tres cosas: *inclusiones* (también llamadas “*partials*” o *plantillas parciales*), *expresiones* (*expressions*) y bloques de ayuda (*block helpers*).

Las *inclusiones* usan la sintaxis `{{> templateName}}` y simplemente le dicen a Meteor que reemplace la inclusión por la plantilla del mismo nombre (en nuestro caso `postItem`).

Las *expresiones* como `{{title}}` pueden, o bien llamar a una propiedad del objeto actual o bien, al valor de retorno de un ayudante (*helper*) como el que definiremos más adelante en nuestro gestor de plantilla.

Los *bloques de ayuda* son tags especiales para mantener el control del flujo de la plantilla, por ejemplo `{{#each}}...` `{{/each}}` o `{{#if}}...{{/if}}`.

Siquieres saber más sobre Spacebars puedes consultar la [documentación oficial](#).

Con estos conocimientos, ya podemos entender cómo van a funcionar nuestras plantillas:

Primero, en la plantilla `postsList` iteramos sobre un objeto `posts` usando un bloque `{{#each}}...{{/each}}`, y para

cada iteración, incluimos la plantilla `postItem`.

Pero, ¿de dónde viene el objeto `posts`? Buena pregunta. Es un **ayudante de plantilla**, y puedes pensar en ellos como un cajón o hueco para valores dinámicos.

La plantilla `postItem` es bastante sencilla. Solo usa tres expresiones: `{{url}}` y `{{title}}` devuelven propiedades, y `{{domain}}` llama a un ayudante.

Hasta ahora hemos estado tratando con Spacebars, que es poco más que HTML con algunas etiquetas extra. A diferencia de otros lenguajes como PHP (o páginas HTML con JavaScript), Meteor mantiene las plantillas y su lógica separadas, de forma que nuestras plantillas por sí mismas no hacen casi nada.

Para que una plantilla tenga vida, necesita **ayudantes**. Puedes pensar en ellos como los cocineros que toman los ingredientes (tus datos) y los preparan, antes de entregar el plato terminado (las plantillas) al camarero, que, finalmente, los entrega.

En otras palabras, mientras la función de las plantillas es mostrar o iterar sobre variables, los ayudantes son los que hacen el trabajo pesado asignando un valor a cada variable.

Puede ser tentador pensar que los ficheros que contienen estos ayudantes son una especie de “controlador”. Pero eso sería ambiguo, ya que los controladores (al menos en el sentido de controladores MVC) normalmente tienen un papel diferente.

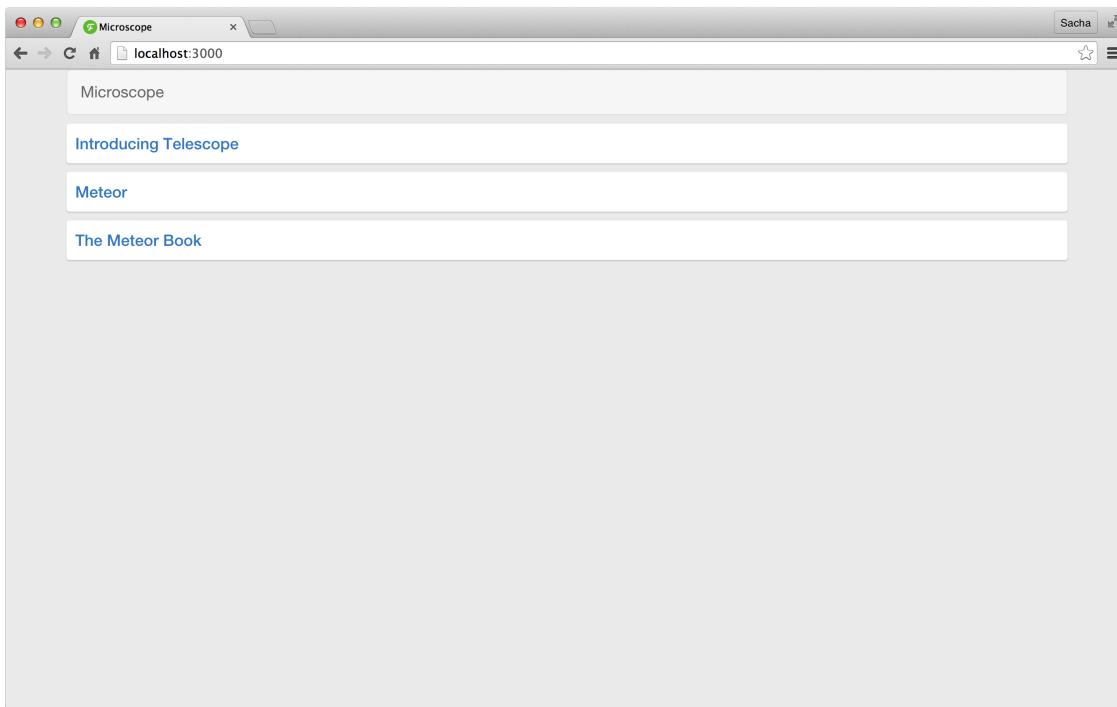
Así que hemos decidido apartarnos de esa terminología, y simplemente nos referimos a “ayudantes de plantillas” o “lógica de plantilla” cuando hablamos del código JavaScript que acompaña a las plantillas.

Para mantener las cosas ordenadas, adoptaremos la convención de nombrar al fichero que contiene la plantilla con el mismo nombre, pero con la extensión `.js`. Así que vamos a crear un fichero `posts_list.js` dentro de

`/client/templates/posts` para construir nuestro primer ayudante:

```
var postsData = [
  {
    title: 'Introducing Telescope',
    url: 'http://sachagreif.com/introducing-telescope/'
  },
  {
    title: 'Meteor',
    url: 'http://meteor.com'
  },
  {
    title: 'The Meteor Book',
    url: 'http://themeteorbook.com'
  }
];
Template.postsList.helpers({
  posts: postsData
});
```

Si todo está bien, ya se pueden ver los datos en el navegador:



Estamos haciendo dos cosas. Primero, creamos algunos datos prototipo en `postsData`. Normalmente, estos datos vienen de la base de datos, pero como no hemos visto cómo hacerlo todavía (espera al siguiente capítulo), hacemos trampa mediante el uso de datos estáticos.

Segundo, usamos la función `Template.postsList.helpers()` para definir un ayudante de plantilla llamado `posts` que, sencillamente devuelve nuestros datos creados en `postsData`.

Y si recuerdas, estamos usando el ayudante `posts` en nuestra plantilla `postsList`:

```
<template name="postsList">
  <div class="posts page">
    {{#each posts}}
      {{> postItem}}
    {{/each}}
  </div>
</template>
```

Al definir el ayudante `posts`, conseguimos que esté disponible para usarlo en la plantilla, así que nuestra plantilla será capaz de recorrer el array `postData` pasando la plantilla `postItem` para cada uno de sus elementos.

Añadimos una plantilla básica para recorrer los posts y d...

[Ver en GitHub](#)

[Lanzar instancia](#)

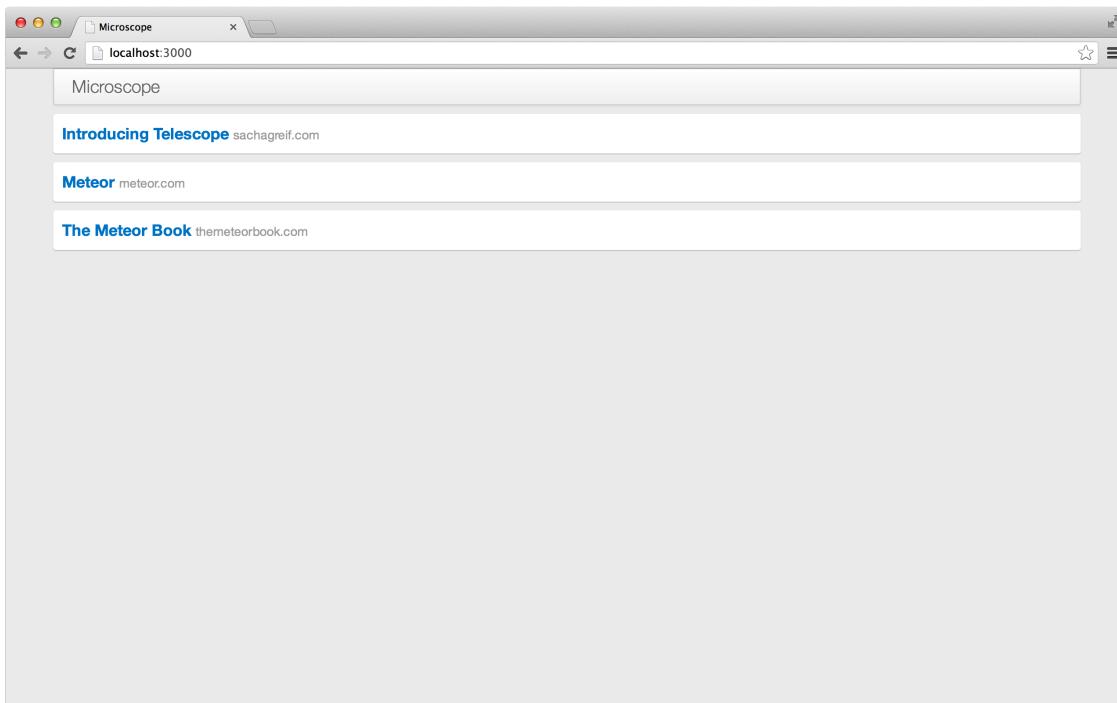
domain

De forma similar, crearemos un fichero `post_item.js` para albergar la lógica de la plantilla `postItem`:

```
Template.postItem.helpers({
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  }
});
```

client/templates/posts/post_item.js

Esta vez el valor de nuestro ayudante `domain`, no son datos sino una función anónima. Este patrón es mucho más común (y más útil) en comparación con nuestros ejemplos de datos ficticios.



El ayudante `domain` coge una URL y devuelve su dominio a través de un poco de magia JavaScript. Pero, ¿de dónde saca esa url la primera vez?.

Para responder a esta pregunta tenemos que volver a nuestra plantilla `posts_list.html`. El bloque `{{#each}}` no solo itera nuestros datos, sino que también **establece el valor de `this` dentro del bloque al objeto siendo iterado**.

Esto significa que entre las dos etiquetas `{{each}}`, el valor de `this` es asignado a cada `post` sucesivamente, y esto se hace extensivo al gestor de la plantilla (`post_item.js`).

Ahora entendemos porqué `this.url` devuelve la URL del post actual. Y más aún, como utilizamos `{{title}}` y `{{url}}` dentro de nuestra plantilla `post_item.html`, Meteor sabe que lo que queremos es `this.title` y `this.url` y devuelve los valores correctos.

Establecemos un ayudante `domain` en `postItem`.

[Ver en GitHub](#)

[Lanzar instancia](#)

Aunque esto no es específico de Meteor, he aquí una breve explicación de la “magia de JavaScript”. En primer lugar, estamos creando un elemento HTML ancla (`a`) vacío y lo almacenamos en la memoria.

A continuación, establecemos el atributo `href` para que sea igual a la URL del post actual (como acabamos de ver, en un ayudante, `this` es el objeto que se está usando en este momento).

Por último, aprovechamos de la propiedad `hostname` del elemento `a` para devolver el nombre de dominio del post sin el resto de la URL.

Si todo ha ido bien deberíamos ver una lista de posts en el navegador. Esta lista son solo datos estáticos, lo que, por el momento, no nos permite aprovechar las características de tiempo real de Meteor. ¡Aprenderemos cómo hacerlo en el próximo capítulo!

Probablemente ya habrás notado que no es necesario recargar la ventana del navegador cada vez que cambiamos un archivo de código.

Esto se debe a que Meteor hace un seguimiento de todos los archivos en el directorio del proyecto, y los actualiza automáticamente en el navegador cada vez que detecta un cambio en alguno de ellos.

La recarga automática es bastante inteligente, ya que incluso, ¡conserva el estado de la aplicación entre dos refrescos!

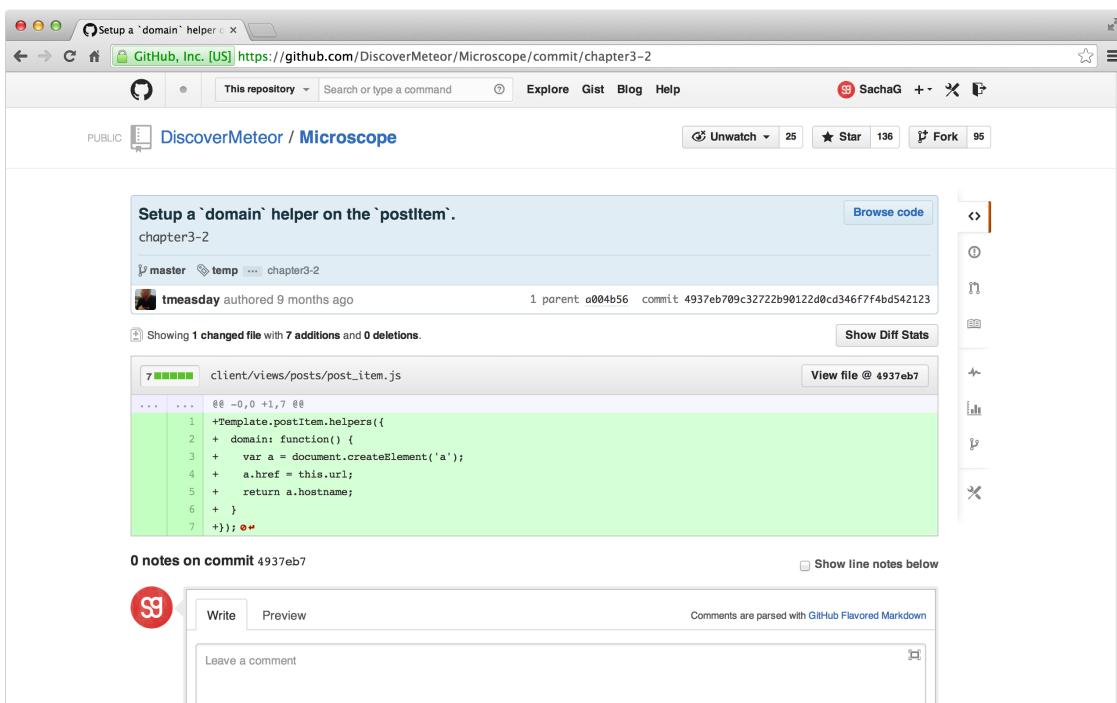
GitHub es un repositorio para proyectos de código abierto basado en el sistema de control de versiones **Git**, y su función es hacer que sea fácil compartir código y colaborar en proyectos. Pero también es una gran herramienta de aprendizaje. En este capítulo, vamos a ver cómo usar GitHub para seguir *Descubriendo Meteor*.

Esta barra lateral asume que no estás familiarizado con Git ni con Github. Si ya manejas las dos cosas, ¡no dudes en pasar al siguiente capítulo!

El bloque básico de trabajo de un repositorio git es el *commit* o *confirmación de código*. Puedes pensar en un commit como en una instantánea del estado de tu código en un momento dado.

En lugar de darte el código final de microscope, hemos ido tomando instantáneas en cada paso del proceso de construcción, y las hemos compartido todas en GitHub.

Por ejemplo, este es **el último commit del capítulo anterior** y se ve así:



Lo que ves es el “diff” (de “diferencia”) del archivo `post_item.js`, es decir, los cambios introducidos por este commit. En este caso, hemos creado el archivo `post_item.js` a partir de cero, por lo que todo su contenido se destaca en verde.

Vamos a compararlo con un ejemplo de **más adelante**:

The screenshot shows a GitHub commit page for a Meteor project. The commit message is "Added basic upvoting algorithm." and it's part of the "chapter13-1" branch. The commit was authored by tmeasday 9 months ago with a single parent commit (abfd312). The code changes are shown in two files: "client/views/posts/post_item.html" and "client/views/posts/post_item.js". In "post_item.html", lines 3, 4, 5, 7, and 8 are highlighted in green, indicating additions. In "post_item.js", lines 7 and 8 are highlighted in green, also indicating additions. The GitHub interface includes a sidebar with various icons for repository management.

Esta vez, solo se resaltan en verde las líneas modificadas.

Y, por supuesto, a veces no estás añadiendo o modificando código, también **boras cosas**:

The screenshot shows a GitHub commit page for a Meteor project. The commit message is "Augmented the postsList route to take a limit" and it's part of the "chapter12-2" branch. The commit was authored by tmeasday 9 months ago with a single parent commit (c7af59e). The code changes are shown in two files: "client/views/posts/posts_list.js" and "lib/router.js". In "posts_list.js", lines 1 through 5 are highlighted in red, indicating deletions. In "router.js", lines 5 and 6 are highlighted in red, also indicating deletions. The GitHub interface includes a sidebar with various icons for repository management.

Ya hemos visto el primer uso de GitHub: ver de un vistazo lo que ha cambiado.

La vista de commits muestra los cambios incluidos en este commit en particular, pero a veces, puede que quieras ver los

archivos que *no* han cambiado, solo para asegurarnos cuál es el estado del código en esta etapa del proceso.

Una vez más GitHub nos echa una mano. Cuando estás en una página de un commit, haz clic en el código **Browse code**:

The screenshot shows a GitHub commit detail page for a repository named 'DiscoverMeteor / Microscope'. The commit is titled 'chapter3-2' and was authored by 'tmeasday' 9 months ago. The commit message is 'Setup a `domain` helper on the `postitem`.'. The code editor displays a diff for the file 'client/views/posts/post_item.js', showing 7 additions and 0 deletions. The additions are as follows:

```
@@ -0,0 +1,7 @@
+Template.postItem.helpers({
+  domain: function() {
+    var a = document.createElement('a');
+    a.href = this.url;
+    return a.hostname;
+  }
});
```

Below the code editor, there is a note on the commit: '0 notes on commit 4937eb7'. A comment section is also present.

Como ves, tienes acceso al repo *tal y como estaba en ese commit específico*:

The screenshot shows the GitHub repository page for 'DiscoverMeteor / Microscope'. The repository has 5 commits, 16 branches, 89 releases, and 1 contributor. The commit '4937eb709c' is selected in the tree dropdown. The main content area shows the commit details for 'chapter3-2' and the file 'README.markdown' which contains the text 'Microscope' and its description as a social news app.

GitHub no nos da muchas pistas visuales de que estamos viendo un commit en particular, pero si lo comparáramos con la rama máster, veremos que la estructura de archivos es muy diferente:

Acabamos de ver cómo explorar todo un repositorio desde la web de GitHub. Pero, ¿cómo lo hacemos a nivel local? Por ejemplo, para ejecutar la aplicación y ver cómo se comporta en un commit en concreto.

Para verlo, empezaremos a usar la `utilidad de línea de comandos` de git. Para empezar, **asegúrate de que tienes instalado Git** y entonces **clona** (o, descarga una copia local) el repositorio Microscope:

```
git clone https://github.com/DiscoverMeteor/Microscope.git github_microscope
```

`github_microscope` es el nombre del directorio local dónde se le “clonará” la aplicación. Si ya existe el directorio `microscope`, solo tienes que elegir cualquier otro nombre (no tiene que tener el mismo nombre que el repositorio GitHub).

Ahora entramos con `cd` en el repositorio recién descargado y estamos listos para usar la `utilidad de línea de comando` de git:

```
cd github_microscope
```

Cuando clonamos el repositorio de GitHub, descargamos *todo* el código, lo que significa que lo que vemos es el último commit ('HEAD') de la aplicación.

Afortunadamente, hay una forma de volver atrás en el tiempo y revisar (“check out”) un commit específico sin afectar a los demás. Vamos a probarlo:

```
git checkout chapter3-1  
Nota: revisando el commit 'chapter3-1'.
```

Ahora, nuestro repositorio se ha "**aislado**" del commit HEAD. Podemos revisarlo, hacer cambios, experimentar con ellos y hacer commits y descartarlos sin que esto afecte a otras ramas cuando hagamos otros checkouts.

Si quieres mantener los cambios que has hecho, puedes crear una nueva rama usando la opción -b con el comando checkout. Por ejemplo:

```
git checkout -b new_branch_name
```

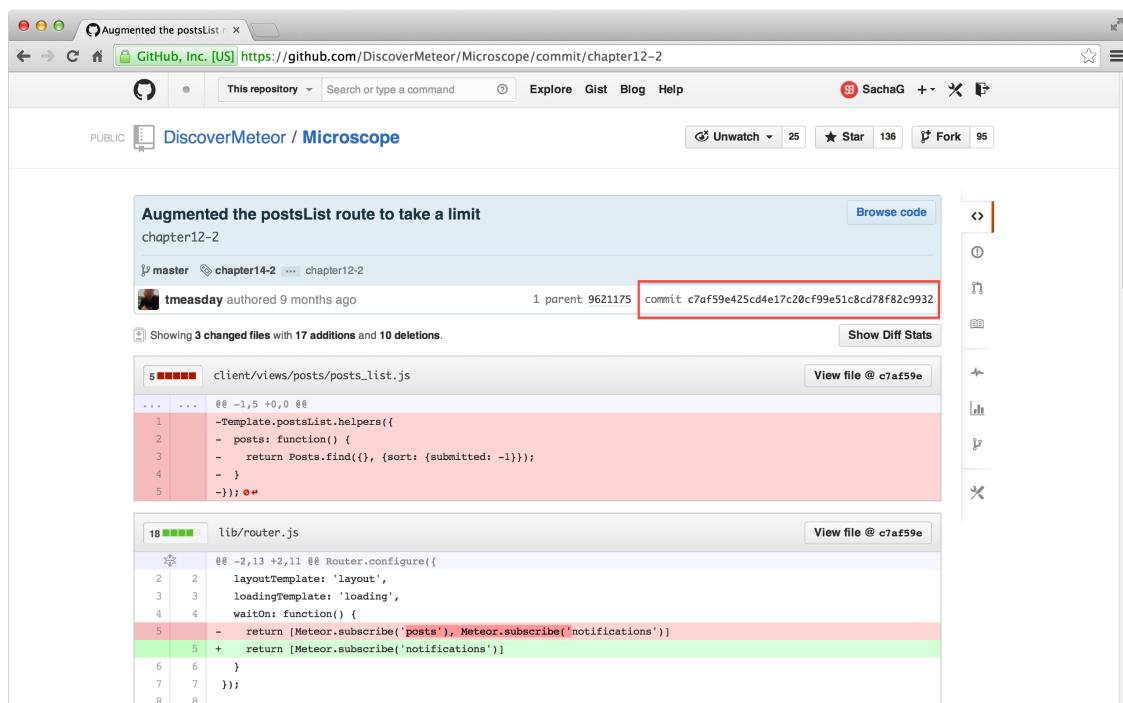
```
HEAD is now at a004b56... Added basic posts list template and static data.
```

Git nos está informando de que estamos “aislados de HEAD”, lo que significa que ahora podremos ver commits del pasado pero no podremos cambiarlos, como si los viéramos a través de una bola de cristal.

(Ten en cuenta que Git tiene comandos que permiten *cambiar* commits del pasado. Esto sería como viajar en el tiempo y pisar una mariposa, pero queda fuera del alcance de esta breve introducción).

Hemos sido capaces de escribir tan solo `chapter3-1` porque todos los commits de Microscope están etiquetados con el número de capítulo correcto. Si este no fuera el caso, tendrías que encontrar el **hash** o identificador único del commit que quieras ver.

Una vez más, GitHub nos lo pone fácil. Podemos encontrar el hash de cada commit en la esquina inferior derecha de la cabecera azul como se puede ver a continuación:



Así que vamos a intentarlo con el hash en vez de con la etiqueta:

```
git checkout c7af59e425cd4e17c20cf99e51c8cd78f82c9932
Previous HEAD position was a004b56... Added basic posts list template and static data.
HEAD is now at c7af59e... Augmented the postsList route to take a limit
```

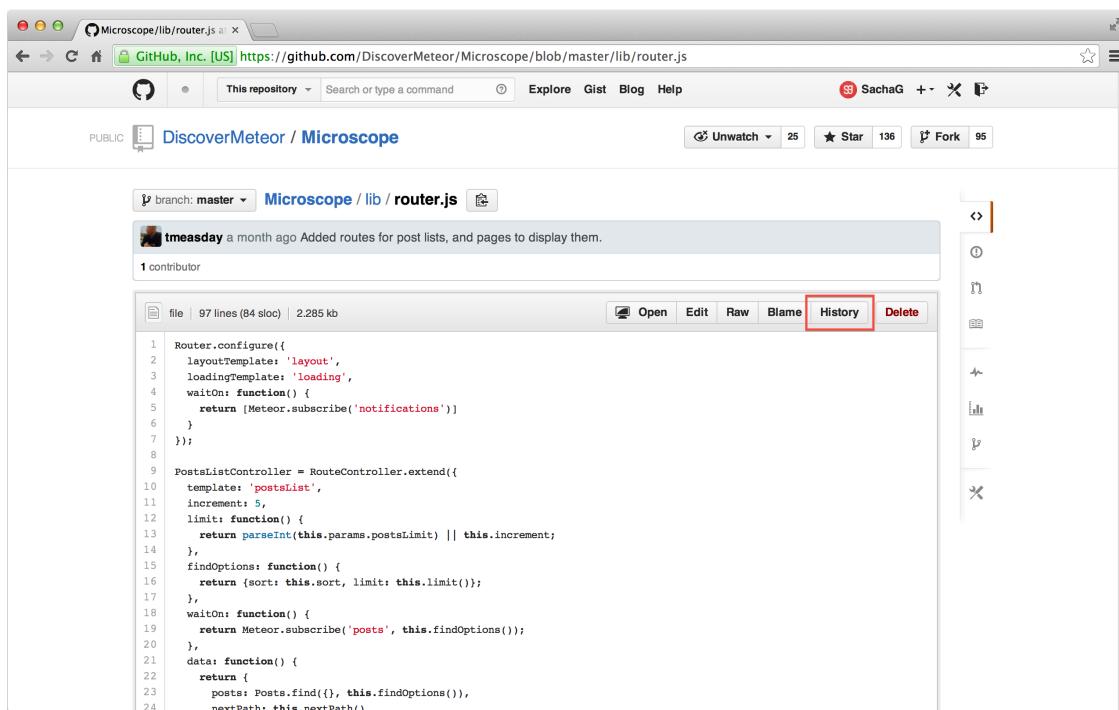
Y por último, ¿qué pasa si queremos dejar de mirar la bola mágica y volver al presente? Pues le decimos a Git que queremos volver a la rama **master**:

```
git checkout master
```

Recuerda que, en cualquier momento del proceso, puedes ejecutar la aplicación usando el comando `meteor`, aunque el repositorio no esté en HEAD. Puede ser necesario ejecutar un `meteor update` primero si Meteor se queja de que faltan paquetes, ya que el código de los paquetes no está incluido en el repositorio.

Este es otro escenario muy común: estás mirando un archivo y ves cambios que no habías visto antes y no recuerdas cuándo se han hecho. Podríamos revisar comit a commit hasta encontrar el que nos interesa pero, hay una forma más fácil gracias a la característica **History** de GitHub.

En primer lugar, accede a uno de los archivos del repositorio en GitHub, y a continuación, busca el botón “History”:



Ahora, dispones de una lista ordenada de todos los commits que afectaron a este archivo en particular:

Vamos a echar un vistazo a la vista **Blame**:

Esta vista ordenada nos muestra línea por línea quién modificó el archivo en cada commit (en otras palabras, a quién hay que echar la culpa si las cosas ya no funcionan):

The screenshot shows a GitHub repository page for 'DiscoverMeteor / Microscope'. The specific file is 'posts.js' under the 'collections' directory. The blame view displays the history of changes made to the file. Each commit includes the author, date, message, and a snippet of the modified code. The commits are as follows:

- 589ef5d5 - tmeasday, 2013-04-07: Added a posts collection
- 7b001288 - tmeasday, 2013-04-07: Removed insecure, and allowe...
- 46fb2c5c - tmeasday, 2013-04-07: Added basic permission to ch...
- f66b77ab - tmeasday, 2013-04-07: Only allow changing certain -
- 46fb2c5c - tmeasday, 2013-04-07: Added basic permission to ch...
- c71f9b52 - tmeasday, 2013-04-07: Meteor.methods({

The code snippets show the implementation of a Meteor collection named 'Posts' with its permissions and methods defined.

Git, al igual que GitHub, son herramientas complejas, por lo que no podemos pretender conocerlas con profundidad en un solo capítulo. De hecho, apenas hemos arañado la superficie de lo que es posible hacer con ellas. Pero, lo poco que hemos visto te va a permitir seguir el resto del libro sin problemas.

En el Capítulo uno hablamos sobre la característica central de Meteor: la sincronización automática de datos entre cliente y servidor.

En este capítulo miraremos más de cerca todo esto y observaremos cómo funciona la tecnología que lo hace posible, las **Colecciones** Meteor.

Una colección es una estructura de datos especial que se encarga de almacenar los datos de forma permanente, en una base de datos MongoDB en el servidor, y de la sincronización de datos en tiempo real con el navegador de cada usuario conectado.

Queremos que nuestros posts sean permanentes y los podamos compartir con otros usuarios, así que vamos a empezar creando una colección llamada `Posts` para poder almacenarlos.

Las colecciones son el eje central de cualquier aplicación, así que para asegurarnos de que se definen primero, las pondremos en el directorio `lib`. Si todavía no lo has hecho, crea un directorio llamado `collections/` dentro de `lib`, crea un archivo llamado `posts.js` y añade lo siguiente:

```
Posts = new Mongo.Collection('posts');
```

lib/collections/posts.js

Colección Posts

[Ver en GitHub](#)

[Lanzar instancia](#)

En Meteor, la palabra clave `var` limita el alcance del objeto al archivo actual. Nosotros queremos que los Posts estén disponibles para toda nuestra aplicación, por eso *no* usamos `var`.

Las aplicaciones web tienen a su disposición básicamente tres formas de almacenar datos, cada una desempeñando un rol diferente:

- **La memoria del navegador:** cosas como las variables JavaScript son almacenadas en la memoria del navegador, lo que implica que no son *permanentes*: son datos locales a la pestaña del navegador y desaparecerán tan pronto como la cierras.
- **El almacén del navegador:** los navegadores también pueden almacenar datos de forma permanente usando cookies o **Local Storage**. Aunque estos datos sean permanentes de una sesión a otra, son *locales* al usuario actual

(pero disponible entre las pestañas) y no se puede compartir de forma sencilla con otros usuarios.

- **La base de datos del servidor:** el mejor lugar para almacenar datos de forma permanente para que puedan estar disponibles para más de un usuario es una base de datos (Siendo MongoDB la solución por defecto para las aplicaciones Meteor).

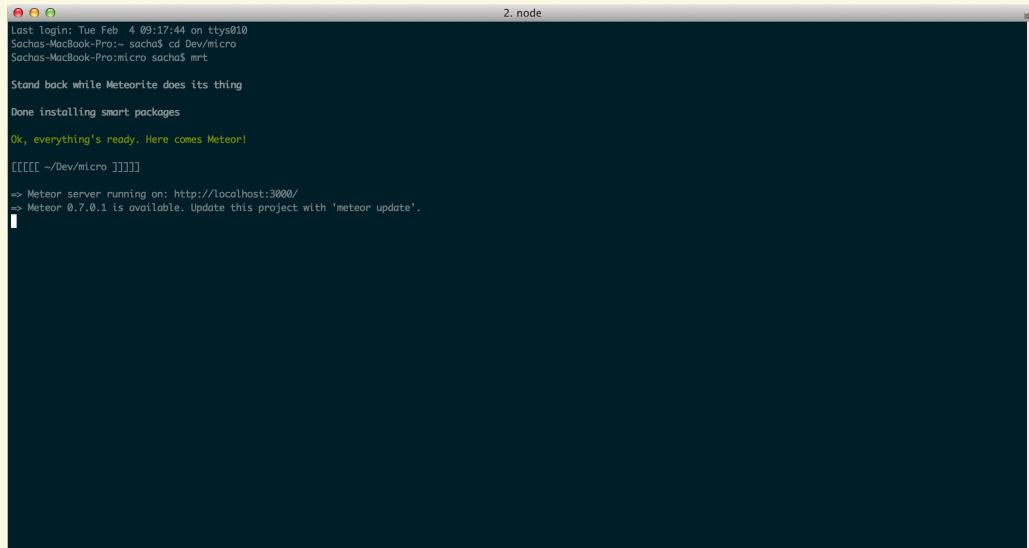
Meteor hace uso de estas tres formas, y a veces sincroniza los datos de un lugar a otro (como veremos pronto). Dicho esto, la base de datos permanece como la fuente de datos “canónica” que contiene la copia maestra de los datos.

El código dentro de las carpetas que no sean `client/` ni `server/` se ejecutará en *ambos* contextos. Por lo que la colección `Posts` estará disponible en el lado cliente y servidor. Sin embargo, lo que hace la colección en cada entorno es bastante diferente.

En el servidor, la colección tiene la tarea de hablar con la base de datos MongoDB y leer y escribir cualquier cambio. En este sentido, se puede comparar con una librería de base de datos estándar.

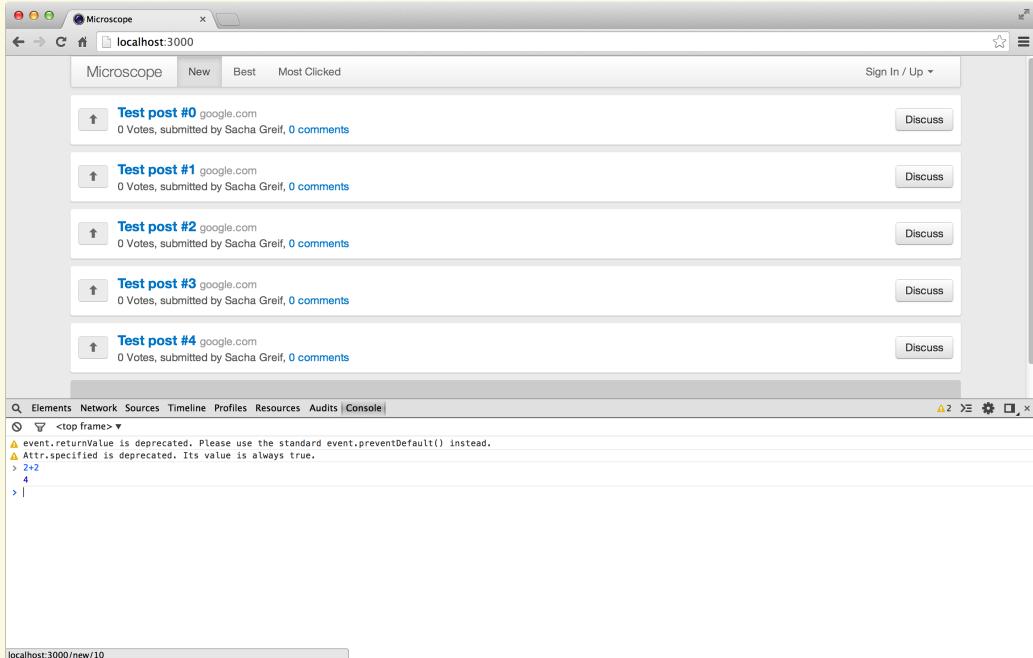
En el cliente sin embargo, la colección es una copia de *un subconjunto* de la colección canónica. La colección del lado del cliente se mantiene actualizada, de forma constante y (normalmente) transparente con ese subconjunto de datos en tiempo real.

En este capítulo hemos empezado a usar la **consola del navegador**, que no debemos confundir con la **terminal**, con la **Shell de Meteor** o con la **Shell de Mongo**. A continuación, explicamos brevemente cada una de ellas.



```
Last Login: Tue Feb  4 09:17:44 on ttys010
Sachos-MacBook-Pro:- sachos$ cd Dev/micro
Sachos-MacBook-Pro:micro sachos$ mrt
Stand back while Meteorite does its thing
Done installing smart packages
Ok, everything's ready. Here comes Meteor!
[[[[[ ~/Dev/micro ]]]]]
=> Meteor server running on: http://localhost:3000/
=> Meteor 0.7.0.1 is available. Update this project with 'meteor update'.
```

- Se abre desde el sistema operativo.
- Las llamadas a `console.log()` en el **lado del servidor** se muestran por aquí.
- Prompt: `$`.
- También se conoce como: Shell, Bash



- Se abre desde dentro del navegador, ejecuta código JavaScript.
- Las llamadas a `console.log()` en el **lado del cliente** se muestran por aquí.
- Prompt: > .
- También se conoce como: JavaScript Console, DevTools Console

A screenshot of a terminal window titled "2. node". It shows a series of commands being run in a Node.js environment, including navigating to a directory, clearing the screen, and starting a "meteor shell". The terminal then displays the "Welcome to the server-side interactive shell!" message and provides help information for global variables and the server shell.

- Se abre desde la Terminal con `meteor shell` .
- Te da acceso directo al código de la parte del servidor de tu aplicación.
- Prompt: > .

```

Sachos-MacBook-Pro:~ sachos$ cd Dev/micro
Sachos-MacBook-Pro:micro sachos$ mrt mongo

Stand back while Meteorite does its thing

Done installing smart packages

Ok, everything's ready. Here comes Meteor!

MongoDB shell version: 2.4.4
connecting to: 127.0.0.1:3002/meteor
> db.posts.find()
{
  "_id": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://sachagreif.com/introducing-telescope/",
  "submitted": 1391453182226,
  "commentsCount": 2,
  "upvoters": [],
  "votes": [
    {
      "_id": "pSNiphW43dNtfVQo"
    }
  ],
  "title": "Introducing Telescope",
  "userId": "45r74hbK858Cty4j",
  "author": "Tom Coleman",
  "url": "http://meteor.com",
  "submitted": 1391442382226,
  "commentsCount": 0,
  "upvoters": [],
  "votes": [
    {
      "_id": "3fG8mUJZd8nIR2"
    }
  ]
}
{
  "_id": "43n74hbK858CtY4j",
  "author": "Tom Coleman",
  "url": "http://themeterebook.com",
  "submitted": 139145182226,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "uHudPMiYuZhEZSLm"
    }
  ],
  "title": "Test post #0",
  "userId": "Sacha Greif",
  "author": "Sacha Greif",
  "url": "3w4cWap35gyt2wb7M",
  "submitted": 1391478382227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "r29nT8hefjtqaoas"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ],
  "title": "Test post #1",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-1",
  "submitted": 1391474782227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "2mbz2l6GnsmtNtou"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ],
  "title": "Test post #2",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-2",
  "submitted": 1391471182227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "NkEmrBRtpsaQqFLy"
    }
  ],
  "votes": [
    {
      "_id": "3w4cWap35gyt2wb7M"
    }
  ],
  "title": "Test post #3",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-3",
  "submitted": 1391467582227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "HNGEfF6g2zRdMvsbk"
    }
  ],
  "votes": [
    {
      "_id": "3w4cWap35gyt2wb7M"
    }
  ],
  "title": "Test post #4",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-4",
  "submitted": 1391463982227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "TmW3NQerf8nCKX8dA"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ],
  "title": "Test post #5",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-5",
  "submitted": 1391460382227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "0"
    }
  ],
  "votes": [
    {
      "_id": "T9EKnx3zGg5Op2EM"
    }
  ],
  "title": "Test post #6",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-6",
  "submitted": 1391456782227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "iTKpRIly8MgNBWJX"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ],
  "title": "Test post #7",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-7",
  "submitted": 1391453182227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "FKRKYQ9pdpxfulEFzS"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ],
  "title": "Test post #8",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-8",
  "submitted": 1391449582227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "rqWnxyzCyljy3lNS"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ],
  "title": "Test post #9",
  "userId": "3w4cWap35gyt2wb7M",
  "author": "Sacha Greif",
  "url": "http://google.com/?q=test-9",
  "submitted": 1391445982227,
  "commentsCount": 0,
  "upvoters": [
    {
      "_id": "cFPPrvFsRtJapZq2M"
    }
  ],
  "votes": [
    {
      "_id": "0"
    }
  ]
}
>

```

- Se abre desde la Terminal con `meteor mongo`.
- Te da acceso directo a la base de datos de tu aplicación.
- Prompt: `>`.
- También se conoce como: Mongo Console

Ten en cuenta que no hay que escribir el carácter prompt (`$`, `>`, or `>`) como parte de un comando. Y que puedes asumir como *salida*, todo lo que *no* empieza con el prompt.

Volviendo al servidor, la colección actúa como una API de nuestra base de datos Mongo. En el código del lado del servidor, esto nos permite escribir comandos Mongo como `Posts.insert()` o `Posts.update()`, que harán cambios en la colección `posts` almacenada dentro de Mongo.

Para mirar el interior de la base de datos Mongo, abrimos una segunda ventana de terminal (mientras Meteor se está ejecutando en la primera), vamos al directorio de la aplicación y ejecutamos el comando `meteor mongo` para iniciar una shell de Mongo, en la que podemos escribir los comandos estándares de Mongo (y como de costumbre, salir con `ctrl+c`). Por ejemplo, vamos a insertar un nuevo post:

```

meteor mongo

> db.posts.insert({title: "A new post"});

> db.posts.find();
{ "_id": ObjectId("..."), "title" : "A new post"};

```

Consola de mongo

Debemos saber que cuando alojamos nuestra aplicación en `*.meteor.com`, también podemos acceder a la consola de Mongo usando `meteor mongo myApp`.

Y ya que estamos, también podemos obtener los logs de nuestra aplicación escribiendo `meteor logs myApp`.

La sintaxis de Mongo es familiar, ya que utiliza una interfaz JavaScript. No vamos a hacer ningún tipo de manipulación de datos adicional en la consola de Mongo, pero podemos echar un vistazo de vez en cuando solo para ver lo que pasa por ahí.

Las colecciones son más interesantes en el lado del cliente. Cuando se declara `Posts = new Mongo.Collection('posts');` en el cliente, lo que se está creando es una *caché local dentro del navegador* de la colección real de Mongo. Cuando decimos que las colecciones del lado del cliente son una “caché”, queremos decir que contiene un *subconjunto* de los datos, y ofrece un acceso *muy rápido*.

Es importante entender este punto, ya que es fundamental para comprender la forma en la que funciona Meteor. En general, una colección del lado del cliente consiste en un subconjunto de todos los documentos almacenados en la colección de Mongo (por lo general, nunca querremos enviar *toda nuestra base de datos* al cliente).

En segundo lugar, los documentos se almacenan en *la memoria del navegador*, lo que significa que el acceso a ellos es prácticamente instantáneo. Así que, cuando se llama, por ejemplo, a `Posts.find()` desde el cliente, no hay caminos lentos hasta el servidor o a la base de datos, ya que los datos ya están precargados.

La implementación de Mongo en el lado del cliente de Meteor se llama MiniMongo. Todavía no está implementada por completo y es posible que podamos encontrar algunas características de Mongo que no funcionan en MiniMongo. Sin embargo, todas las que cubrimos en este libro funcionan de manera similar.

La parte más importante de todo esto es cómo se sincronizan los datos de la colección del cliente con la colección del mismo nombre (en nuestro caso `posts`) del servidor.

Mejor que explicarlo en detalle, vamos a verlo.

Empezaremos abriendo dos ventanas del navegador, y accediendo a la consola en cada uno de ellos. A continuación, abrimos la consola de Mongo en la línea de comandos.

En este punto, deberíamos ser capaces de encontrar el único documento que hemos creado antes desde la consola de Mongo (ten en cuenta que el *interfaz* de nuestra aplicación estará mostrando todavía los tres posts de prueba anteriores. Ignóralos por ahora).

```
> db.posts.find();
{title: "A new post", _id: ObjectId("...")};
```

Consola de Mongo

```
> Posts.findOne();
{title: "A new post", _id: LocalCollection._ObjectID};
```

Consola del primer navegador

Creemos un nuevo post en una de las ventanas del navegador ejecutando un insert:

```
> Posts.find().count();
1
> Posts.insert({title: "A second post"});
'xxx'
> Posts.find().count();
2
```

Consola del primer navegador

Como era de esperar, el post aparece en la colección local. Ahora vamos a comprobar Mongo:

```
> db.posts.find();
{title: "A new post", _id: ObjectId("...")};
{title: "A second post", _id: 'yyy'};
```

Consola de Mongo

Como puedes ver, el post ha viajado hasta la base de datos sin escribir una sola línea de código para enlazar nuestro cliente hasta el servidor (bueno, en sentido estricto, hemos escrito una *sola* línea de código: `new Mongo.Collection("posts")`). ¡Pero eso no es todo!

Escribamos esto en la consola del segundo navegador:

```
> Posts.find().count();
2
```

Consola del segundo navegador

¡El post está ahí también! A pesar de que no hemos refrescado ni interactuado con el segundo navegador, y desde luego no hemos escrito código para insertar actualizaciones. Todo ha sucedido por arte de magia – e instantáneamente. Todo esto se hará más evidente más adelante.

Lo que ha pasado es que la colección del cliente ha informado de un nuevo post a la colección del servidor, que inmediatamente se pone a distribuirlo en la base de datos Mongo y a todos los clientes conectados a la colección `post`.

Ver los posts en la consola del navegador no es muy útil. Vamos a aprender cómo conectar estos datos a nuestras plantillas, y de esta forma, convertir nuestro prototipo HTML en una aplicación web en tiempo real.

Ver el contenido de nuestras colecciones en la consola del navegador es una cosa, pero lo que realmente nos gustaría es mostrar los datos y sus cambios en la pantalla. Cuando esto ocurra, habremos convertido nuestra sencilla *página* web que muestra datos estáticos, en una *aplicación* web en tiempo real en la que los datos cambian de forma dinámica.

Lo primero que vamos a hacer es meter unos cuantos datos en la base de datos. Lo haremos mediante un archivo que carga un conjunto de datos estructurados en la colección de `Posts` cuando el servidor se inicia por primera vez.

En primer lugar, vamos a asegurarnos de que no hay nada en la base de datos. Para borrar la base de datos y restablecer el proyecto usaremos `meteor reset`. Por supuesto, hay que ser muy cuidadoso con este comando una vez que se empieza a trabajar en proyectos del mundo-real.

Paramos el servidor Meteor (pulsando `ctrl-c`) y, a continuación, en la línea de comandos, ejecutamos:

```
meteor reset
```

El comando `reset` borra completamente la base de datos Mongo. Es útil en el desarrollo cuándo hay bastantes posibilidades de que nuestra base de datos caiga en un estado inconsistente.

Vamos a iniciar nuestra aplicación Meteor de nuevo:

```
meteor
```

Ahora que la base de datos está vacía, podemos añadir lo siguiente a `server/fixtures.js` para cargar tres posts cuando el servidor arranca y encuentra la colección `Posts` vacía:

```
if (Posts.find().count() === 0) {
  Posts.insert({
    title: 'Introducing Telescope',
    url: 'http://sachagreif.com/introducing-telescope/'
  });

  Posts.insert({
    title: 'Meteor',
    url: 'http://meteor.com'
  });

  Posts.insert({
    title: 'The Meteor Book',
    url: 'http://themeteorbook.com'
  });
}
```

```
server/fixtures.js
```

Datos para la colección de posts.

[Ver en GitHub](#)

[Lanzar instancia](#)

Hemos ubicado este archivo en el directorio `/server`, por lo que no se cargará en el navegador de ningún usuario. El código se ejecutará inmediatamente cuando se inicia el servidor, y hará tres llamadas a `insert` para agregar tres sencillos posts en la colección de Posts.

Ahora ejecutamos nuevamente el servidor con `meteor`, y estos tres posts se cargarán en la base de datos.

Si abrimos una consola de navegador, veremos los tres mensajes cargados desde MiniMongo:

```
➤ Posts.find().fetch();
```

Consola del navegador

Para ver estos mensajes renderizados en HTML, podemos utilizar un ayudante de plantilla.

En el Capítulo 3 vimos cómo Meteor nos permite enlazar un *contexto de datos* a nuestras plantillas Spacebars para construir vistas HTML a partir de estructuras de datos simples. Bien, pues, de la misma forma vamos a enlazar los datos de nuestra colección. Simplemente reemplazamos el objeto JavaScript estático `postsData` por una colección dinámica.

A propósito, no dudes en borrar el código de `postsData`. Así es cómo debe quedar

`client/templates/posts/posts_list.js`:

```
Template.postsList.helpers({
  posts: function() {
    return Posts.find();
  }
});
```

`client/templates/posts/posts_list.js`

Conexión entre la colección Posts y la plantilla
`postList`.

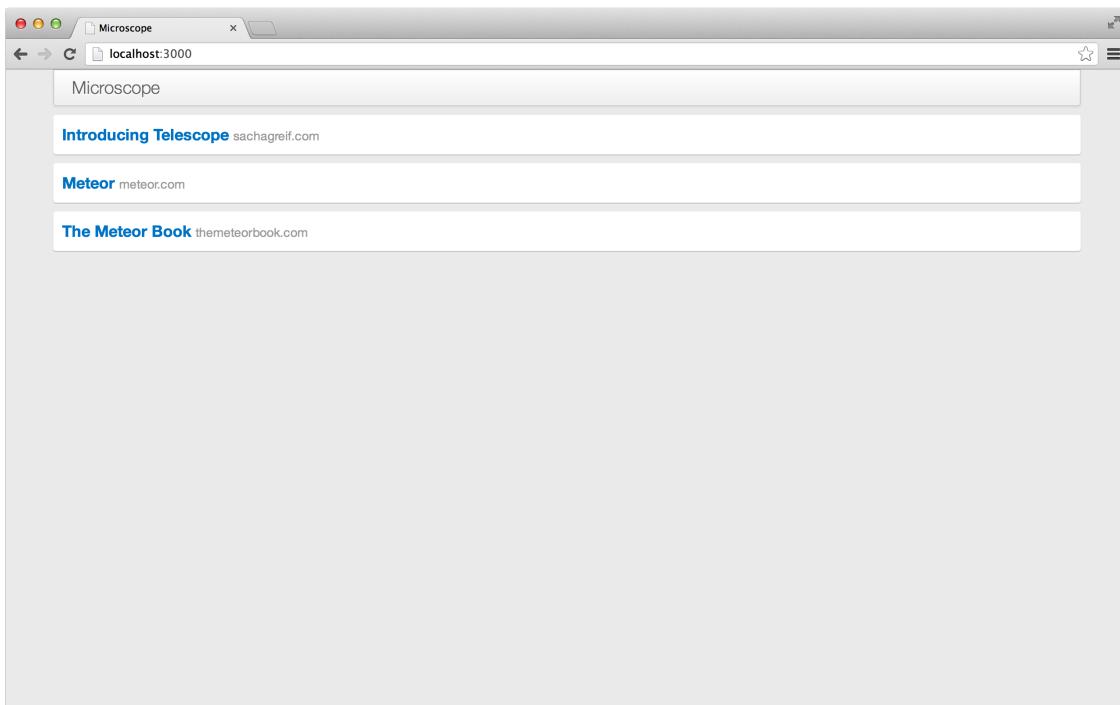
[Ver en GitHub](#)

[Lanzar instancia](#)

En Meteor, `find()` devuelve un cursor que es una **fuente de datos reactiva**. Cuando queramos usar los contenidos a los que apunta el cursor, podemos usar `fetch()` sobre él para trasformarlo en un array.

Dentro de una aplicación, Meteor es lo suficientemente inteligente para saber cómo iterar sobre cursores sin tener que convertirlos de forma explícita en arrays. Por eso no veremos a menudo `fetch()` en el código Meteor (y por eso no lo hemos usado en el ejemplo anterior).

Ahora, en lugar de cargar una lista de mensajes como un array estático desde una variable, ahora estamos devolviendo un cursor a nuestro ayudante `posts` (aunque la cosa no parece muy diferente puesto que estamos devolviendo exactamente los mismos datos):



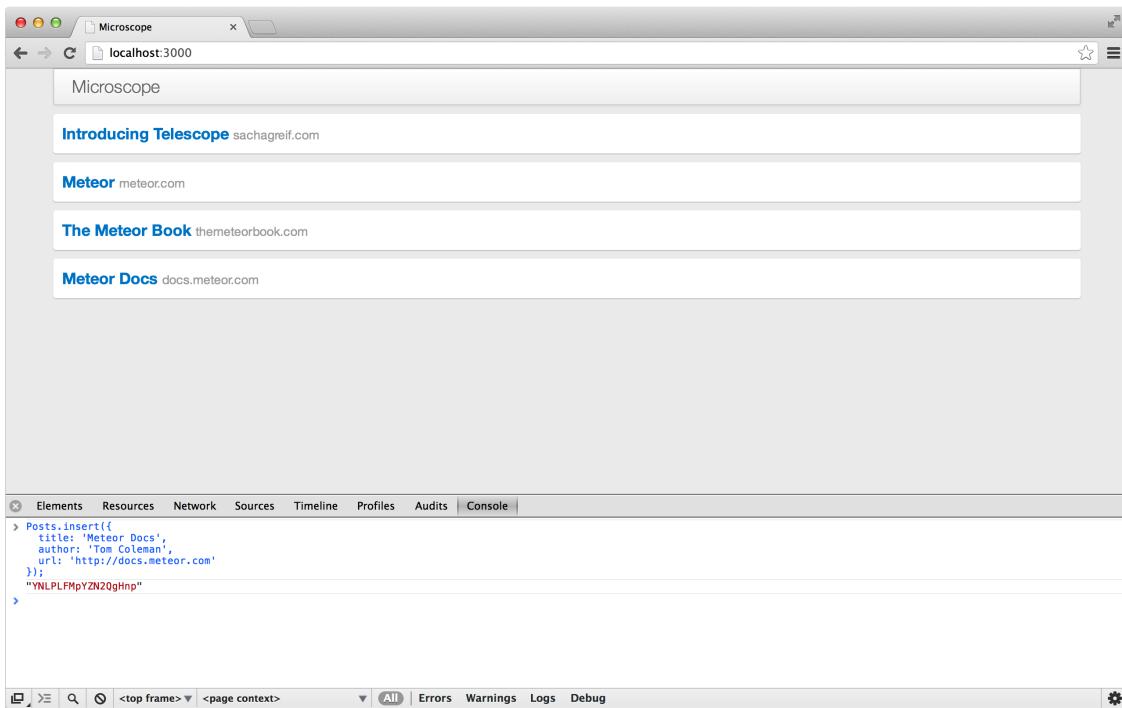
Nuestro ayudante `{{#each}}` ha recorrido todos nuestros `Posts`, y los ha mostrado en la pantalla. La colección del lado del servidor ha tomado los posts de Mongo, los ha pasado a nuestra colección del lado del cliente, y nuestro ayudante Spacebars los ha pasado a la plantilla.

Ahora iremos un paso más allá, y vamos a añadir otro post a través de la consola del navegador:

```
> Posts.insert({  
  title: 'Meteor Docs',  
  author: 'Tom Coleman',  
  url: 'http://docs.meteor.com'  
});
```

Consola del navegador

Vuelve a mirar el navegador – deberías ver esto:



Acabas de ver la reactividad en acción por primera vez. Cuando le pedimos a Spacebars que recorra el cursor `Posts.find()`, él ya sabe cómo monitorizar este cursor en busca de cambios, y de esa forma, alterar el código HTML para mostrar los datos correctos en la pantalla.

En este caso, el cambio más simple posible es añadir otro `<div class="post"> ... </div>`. Si queremos asegurarnos de que esto es realmente lo que ocurre, solo tenemos que abrir el inspector DOM del navegador y seleccionar el `<div>` correspondiente a uno de los posts existentes.

Ahora, desde la consola, insertamos otro post. Cuando volvemos de nuevo al inspector, podremos ver un `<div>`, correspondiente al nuevo post, pero seguirás teniendo el mismo `<div>` seleccionado. Esta es una manera útil de saber cuándo han vuelto a ser renderizados los elementos y cuándo no.

Meteor tiene habilitado por defecto el paquete `autopublish`, algo que no es conveniente para aplicaciones en producción. Este paquete indica que las colecciones son compartidas en su totalidad con cada cliente conectado. Esto no es lo que realmente queremos, así que vamos a deshabilitarlo.

Abrimos una nueva ventana de terminal y escribimos:

```
meteor remove autopublish
```

Esto tiene un efecto instantáneo. Si miramos ahora el navegador, veremos que todos nuestros posts han desaparecido! Esto se debe a que confiábamos en `autopublish` para asegurarnos de que nuestra colección del lado del cliente era una réplica de todos los posts de la base de datos.

Con el tiempo vamos a necesitar asegurarnos de que solo trasferimos los posts que el usuario realmente necesita ver (teniendo en cuenta cosas como la paginación). Pero, por ahora, lo vamos a configurar para que la colección `Posts` se publique en su totalidad (tal y como lo teníamos hasta ahora).

Para ello, creamos una función `publish()` que devuelve un cursor que referencia a todos los posts:

```
Meteor.publish('posts', function() {  
  return Posts.find();  
});
```

server/publications.js

En el cliente, hay que *suscribirse* a la publicación. Añadimos la siguiente línea a `main.js`:

```
Meteor.subscribe('posts');
```

client/main.js

`autopublish` eliminado y configurada una publicación
bás...

[Ver en GitHub](#)

[Lanzar instancia](#)

Si comprobamos el navegador de nuevo, veremos que nuestros posts están de vuelta. ¡Uf!

Entonces, ¿qué hemos logrado? Bueno, a pesar de que no tenemos interfaz de usuario, lo que tenemos es una aplicación web completamente funcional. Podríamos desplegar esta aplicación en Internet, y (mediante la consola del navegador) empezar a publicar nuevas historias y verlas aparecer en los navegadores de otros usuarios de todo el mundo.

Las publicaciones y las suscripciones son dos de los conceptos más importantes de Meteor, pero puede que sean difíciles de comprender si acabas de empezar.

Esto ha acarreado una gran cantidad de malentendidos, como la creencia de que Meteor es inseguro, o que las aplicaciones no pueden manejar grandes cantidades de datos.

La “magia” que hace Meteor es la razón más importante de que ocurra esto al principio. Aunque la magia es, en última instancia muy útil, puede ocultar lo que realmente está pasando entre bastidores (como suele pasar con la magia). Así que vamos a desenvolver las capas de dicha magia para tratar de entender lo que está pasando.

Pero primero, vamos a echar una mirada a los buenos tiempos allá por 2011, cuando todavía no existía Meteor. Digamos que estás haciendo una sencilla aplicación con Rails. Cuando un usuario llega a tu sitio, el cliente (es decir, su navegador) envía una solicitud a tu aplicación, que reside en el servidor.

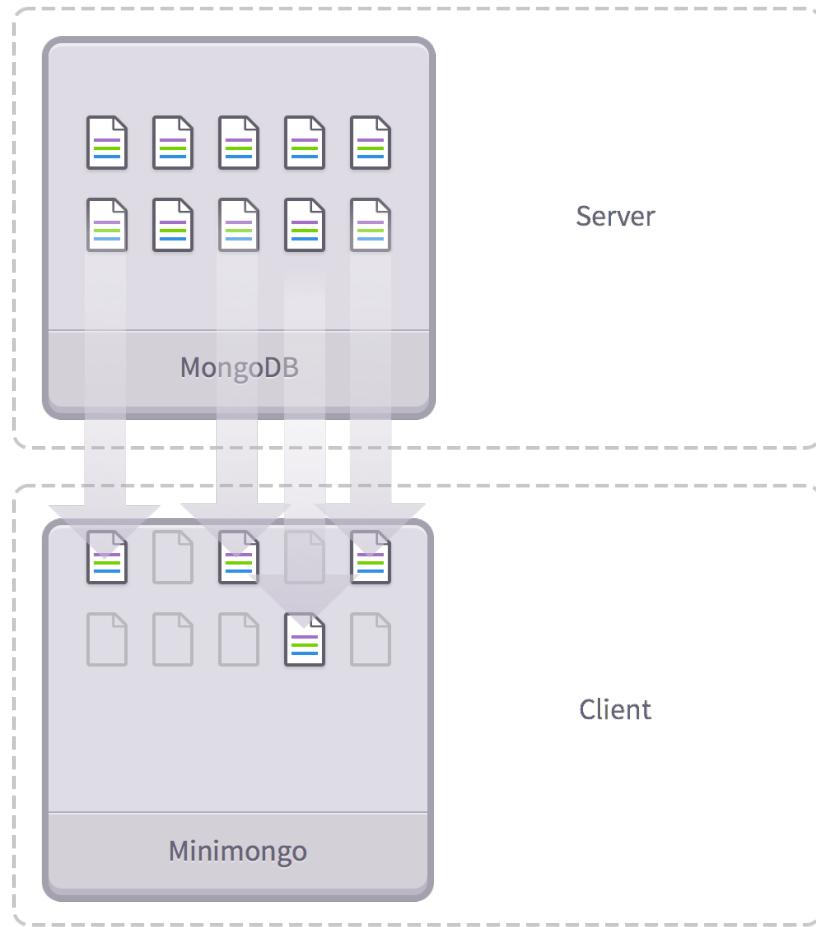
Lo primero que hace la App es averiguar qué datos necesita ver el usuario. Estos podrían ser la página 12 de resultados de búsqueda, la información del perfil de usuario de Mary, los 20 tweets más recientes de Bob, y así sucesivamente. Básicamente, podríamos verlo como un dependiente de una librería buscando por los pasillos el libro que has pedido.

Una vez que tiene los datos correctos, el segundo trabajo de la aplicación es traducir esos datos a un formato HTML agradable y legible (o JSON en el caso de una API).

En la metáfora de la librería, estaríamos envolviendo el libro y metiéndolo en una bolsa bonita. Esta es la “Vista”, del famoso modelo Model-View-Controller.

Por último, la aplicación coge el código HTML y lo envía hacia el navegador. El trabajo de la aplicación ha terminado, y puede relajarse tomando una cerveza mientras espera la siguiente solicitud.

Veamos lo que hace Meteor tan especial. Como hemos visto, la principal innovación de Meteor es que, mientras que una aplicación Rails solo vive **en el servidor**, una aplicación Meteor también incluye componentes que se ejecutarán**en el cliente** (el navegador).



Así que lo que ocurre es que el empleado de la tienda, no solo encuentra el libro, sino que además te sigue a casa para leértelo por la noche (admitiremos que suena un poco raro).

Esta arquitectura permite a Meteor hacer cosas muy interesantes, la más importante es lo que Meteor llama **base de datos en todas partes**. En pocas palabras, Meteor tomará un subconjunto de la base de datos y la copiará en el cliente.

Esto tiene dos grandes implicaciones: la primera es que en lugar de enviar código HTML, una aplicación Meteor envía **datos actuales en bruto** al cliente y deja que el cliente se ocupe de ellos (**data on the wire**). Lo segundo es que serás capaz de **acceder, e incluso modificar esos datos instantáneamente** sin tener que esperar al servidor (**latency compensation**).

Una base de datos de una aplicación puede contener decenas de miles de documentos, algunos de los cuales podrían ser privados o confidenciales. Así que, obviamente, por razones de seguridad y escalabilidad, no deberíamos sincronizar toda la base de datos en el cliente.

Por lo tanto, vamos a necesitar una forma de decirle a Meteor qué **subconjunto** de los datos se pueden enviar al cliente, y lo podremos hacer utilizando las **publicaciones**.

Volvamos a Microscope. Aquí están todos los posts de nuestra aplicación que hay en la base de datos:



Aunque esta función no existe realmente en Microscope, imaginemos que algunos de nuestros posts se han marcado como entradas con lenguaje abusivo. Aunque queramos mantenerlos en nuestra base de datos, no deben ponerse a disposición de los usuarios (es decir, enviarse a los clientes).

Nuestra primera tarea será decirle a Meteor qué datos queremos enviar. Le diremos que solo queremos **publicar** posts sin marcar:



Este sería el código correspondiente, que estaría en el servidor:

```
// on the server
Meteor.publish('posts', function() {
  return Posts.find({flagged: false});
});
```

Esto asegura que **no hay manera posible** de que el cliente pueda acceder a un post marcado. Esta es la forma de hacer una aplicación segura con Meteor: basta con asegurarse de que solo publicas los datos a los que el cliente actual debe tener acceso.

Puedes pensar en el sistema de publicaciones/suscripciones como un embudo que trasfiere datos desde una colección en el servidor (origen) a una en el cliente (destino).

El protocolo que se habla dentro del embudo se llama **DDP** (que significa Protocolo de Datos Distribuidos). Para aprender más sobre DDP, puedes ver [esta charla de la conferencia en Real-timed](#) Matt DeBergalis (uno de los fundadores de Meteor), o [este screencast](#) de Chris Mather que te guía a través de este concepto con un poco más de detalle.

A pesar de que no vamos a poner a disposición de los clientes los posts marcados, pueden quedar miles que no debemos enviar de una sola vez. Necesitamos una forma de que los clientes especifiquen qué subconjunto necesitan en un momento determinado, y aquí es exactamente donde entran las **suscripciones**.

Cualquier dato que se suscribe, se **refleja** en el cliente gracias a Minimongo, la implementación de MongoDB en el lado del cliente que provee Meteor.

Por ejemplo, digamos que estamos viendo la página del perfil de Bob Smith, y que solo queremos *versus* posts.



En primer lugar, podríamos modificar nuestra publicación para que acepte un parámetro:

```
// on the server
Meteor.publish('posts', function(author) {
  return Posts.find({flagged: false, author: author});
});
```

Entonces podríamos definir ese parámetro cuando *nos suscribimos* a esa publicación desde el cliente:

```
// on the client
Meteor.subscribe('posts', 'bob-smith');
```

Esta es la forma de hacer escalable una aplicación Meteor: en lugar de suscribirse a todos los datos disponibles, solo escogemos las piezas que necesitamos en un momento dado. De esta manera, evitaremos sobrecargar la memoria del navegador, sin importar si el tamaño de la base de datos del servidor es enorme.

Ahora resulta que los mensajes de Bob tienen varias categorías (por ejemplo: “JavaScript”, “Ruby”, y “Python”). Tal vez todavía queremos cargar todos los Mensajes de Bob en la memoria, pero en este momento solo queremos mostrar los de la categoría “JavaScript”. Aquí es donde “la búsqueda” entra en juego



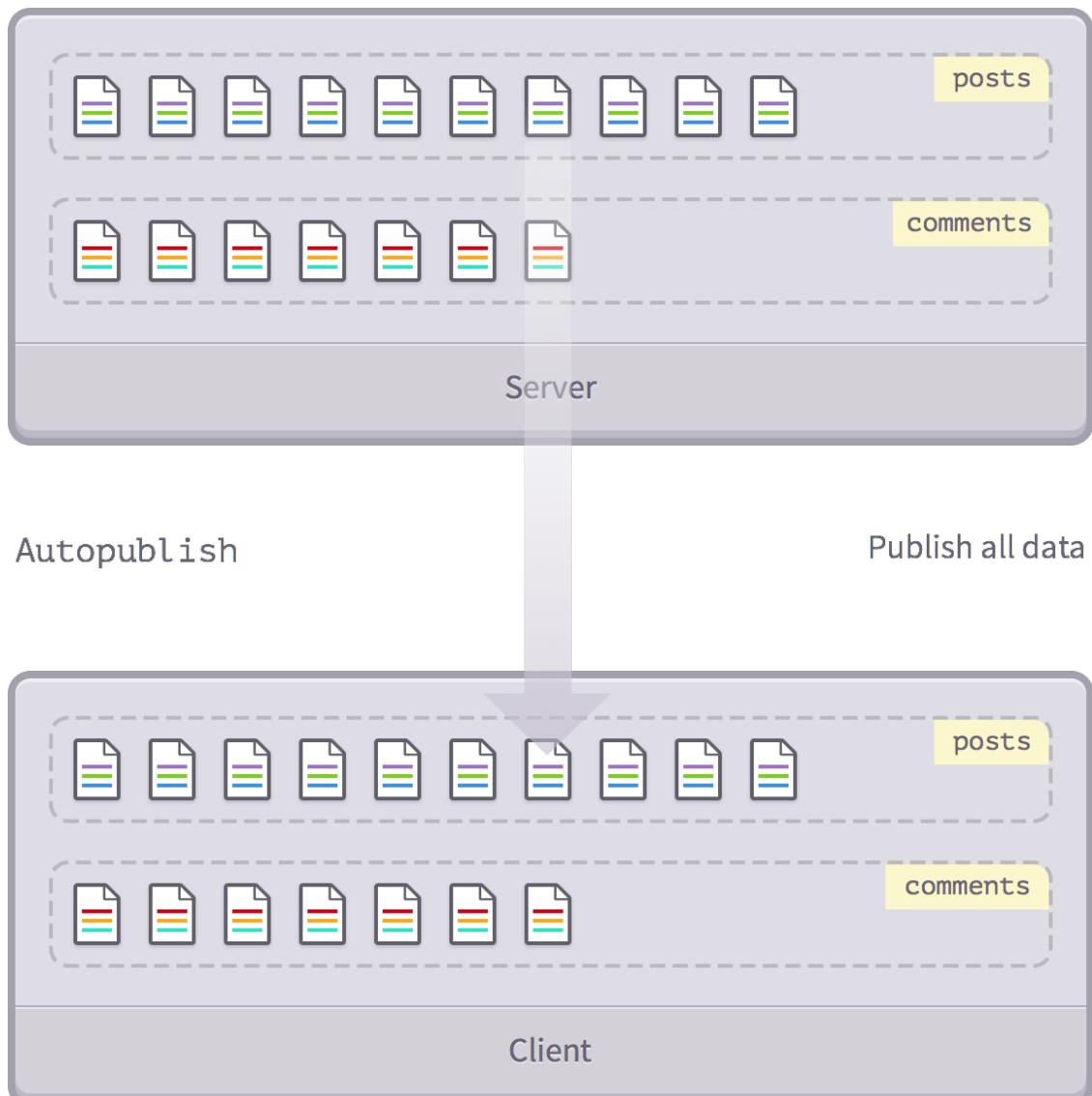
Al igual que hicimos en el servidor, vamos a utilizar la función `Posts.find()` para seleccionar un subconjunto de estos datos:

```
// on the client
Template.posts.helpers({
  posts: function(){
    return Posts.find({author: 'bob-smith', category: 'JavaScript'});
  }
});
```

Ahora que tenemos una buena comprensión de cómo funcionan las publicaciones y suscripciones, vamos a profundizar un poco más, repasando los patrones de diseño más comunes.

Si creas un proyecto Meteor desde cero (es decir, usando `meteor create`), el paquete `autopublish` se habilitará automáticamente. Como punto de partida, vamos a hablar acerca de lo que hace exactamente este paquete.

El objetivo de `autopublish` es que sea muy fácil empezar a desarrollar y lo hace reflejando *todos los datos* del servidor en el cliente, lo que permite olvidarse de publicaciones y suscripciones y empezar directamente a escribir el código de la aplicación.



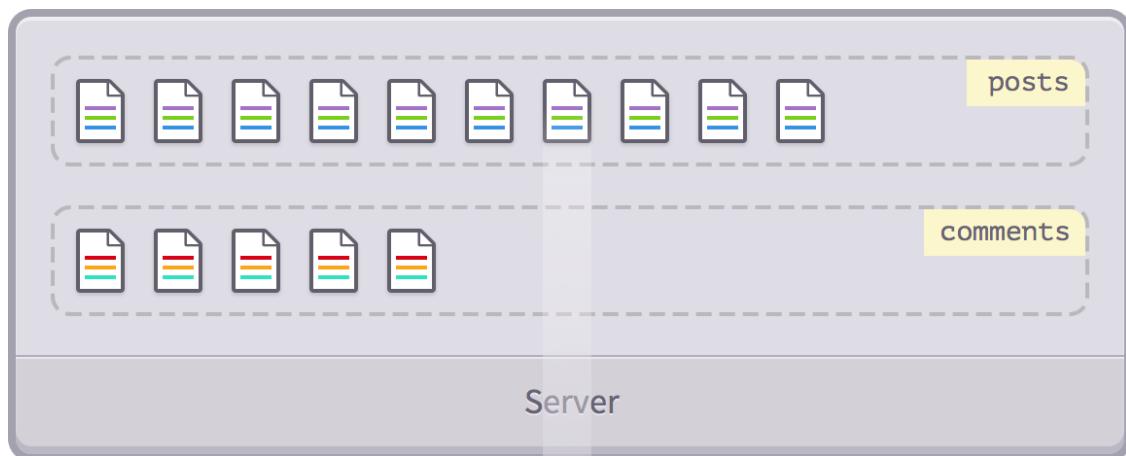
¿Y cómo funciona? Supón que tienes una colección en el servidor llamada 'posts'. Entonces `autopublish` buscará todos los posts que haya en la base de datos Mongo y los enviará automáticamente a una colección llamada 'posts' en el cliente.

Así que si usas `autopublish`, no necesitas pensar en publicaciones. Los datos son omnipresentes, y todo es más sencillo. Por supuesto, aparecen problemas obvios al tener una copia completa de la base de datos en la caché de cada usuario.

Por esta razón, `autopublish` solo es apropiado cuando estamos empezando, cuando todavía no se ha pensado en las publicaciones.

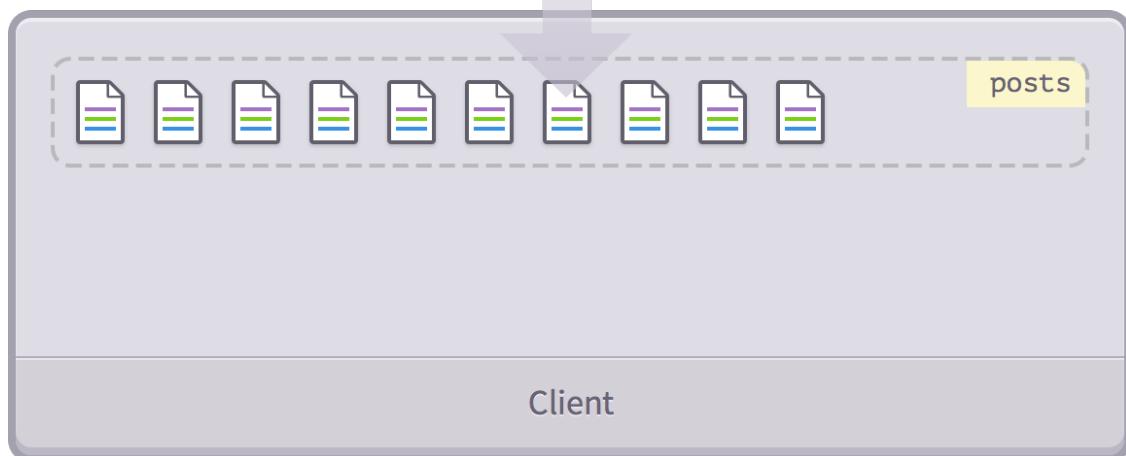
Si eliminamos `autopublish`, te darás cuenta de que todos los datos han desaparecido del cliente. Una forma fácil de traerlos de vuelta es, simplemente, replicar lo que hace `autopublish` publicando una colección en su totalidad. Por ejemplo:

```
Meteor.publish('allPosts', function(){
  return Posts.find();
});
```



```
Meteor.publish('allPosts', function(){
  return Posts.find();
});
```

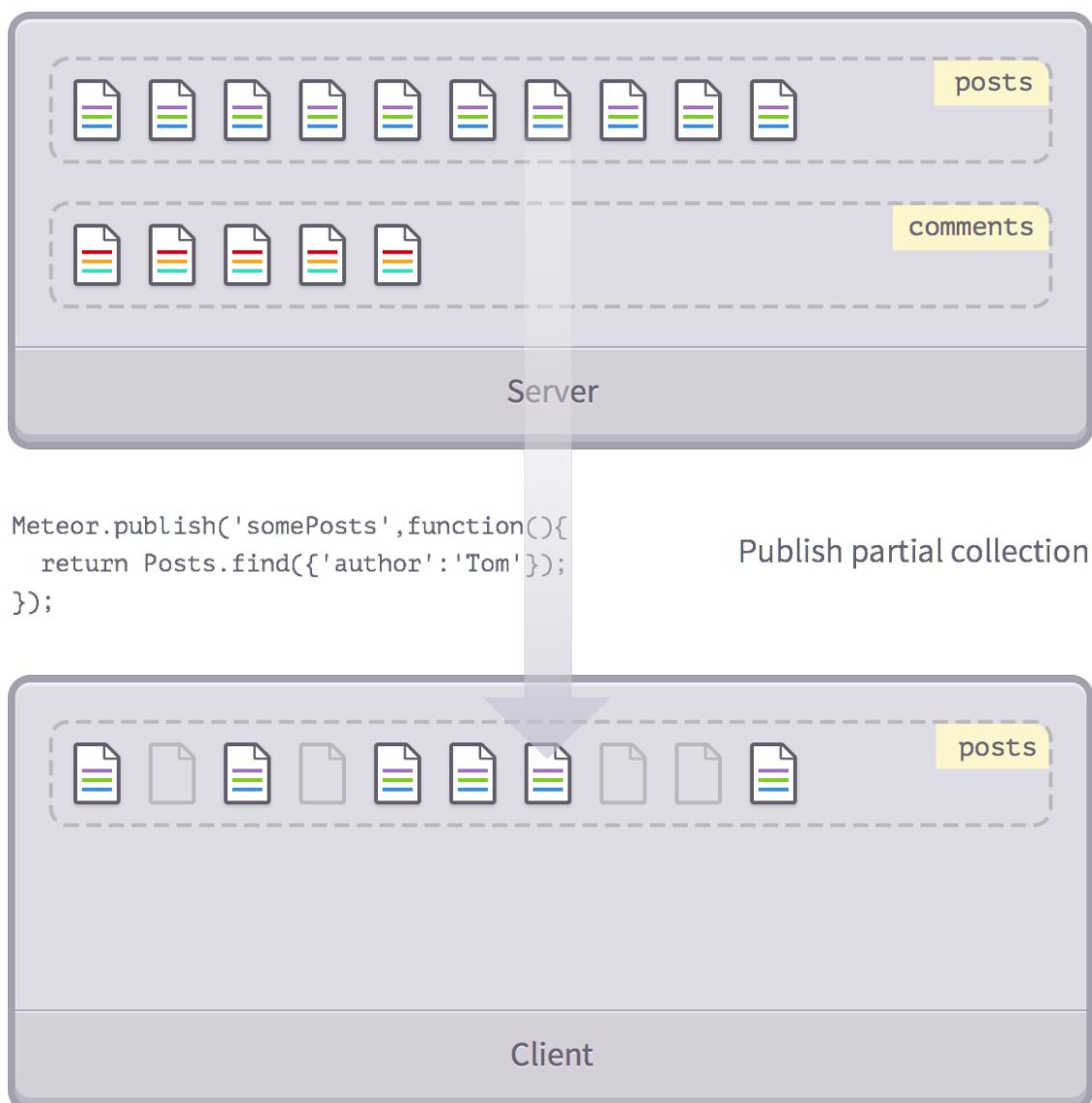
Publish full collection



Todavía publicamos colecciones completas, pero al menos ahora tenemos control sobre qué colecciones publicamos. En este caso, publicamos la colección `Posts` pero no `Comments`.

El siguiente nivel de control es publicar *solo una parte* de una colección. Por ejemplo, solo los posts que pertenecen a un determinado autor:

```
Meteor.publish('somePosts', function(){
  return Posts.find({'author':'Tom'});
});
```



Si has leído la [documentación de Meteor sobre publicaciones](#), tal vez te habrás sentido abrumado al oír hablar de `added()` y `ready()` para establecer los atributos de los registros en el cliente, y te habrá costado cuadrarlo con aplicaciones basadas en Meteor que hayas podido ver y que nunca usan esos métodos.

La razón es que Meteor ofrece una comodidad muy importante: el método `_publishCursor()`. ¿Todavía no lo has utilizado? Tal vez no directamente, pero eso es exactamente lo que Meteor utiliza cuando devuelve un **cursor** (por ejemplo, `Posts.find({'author': 'Tom'})`) desde una función `publish`.

Cuando Meteor comprueba que la publicación `somePosts` ha devuelto un cursor, automáticamente llama a `_publishCursor()` para publicar ese cursor.

Esto es lo que hace `_publishCursor()`:

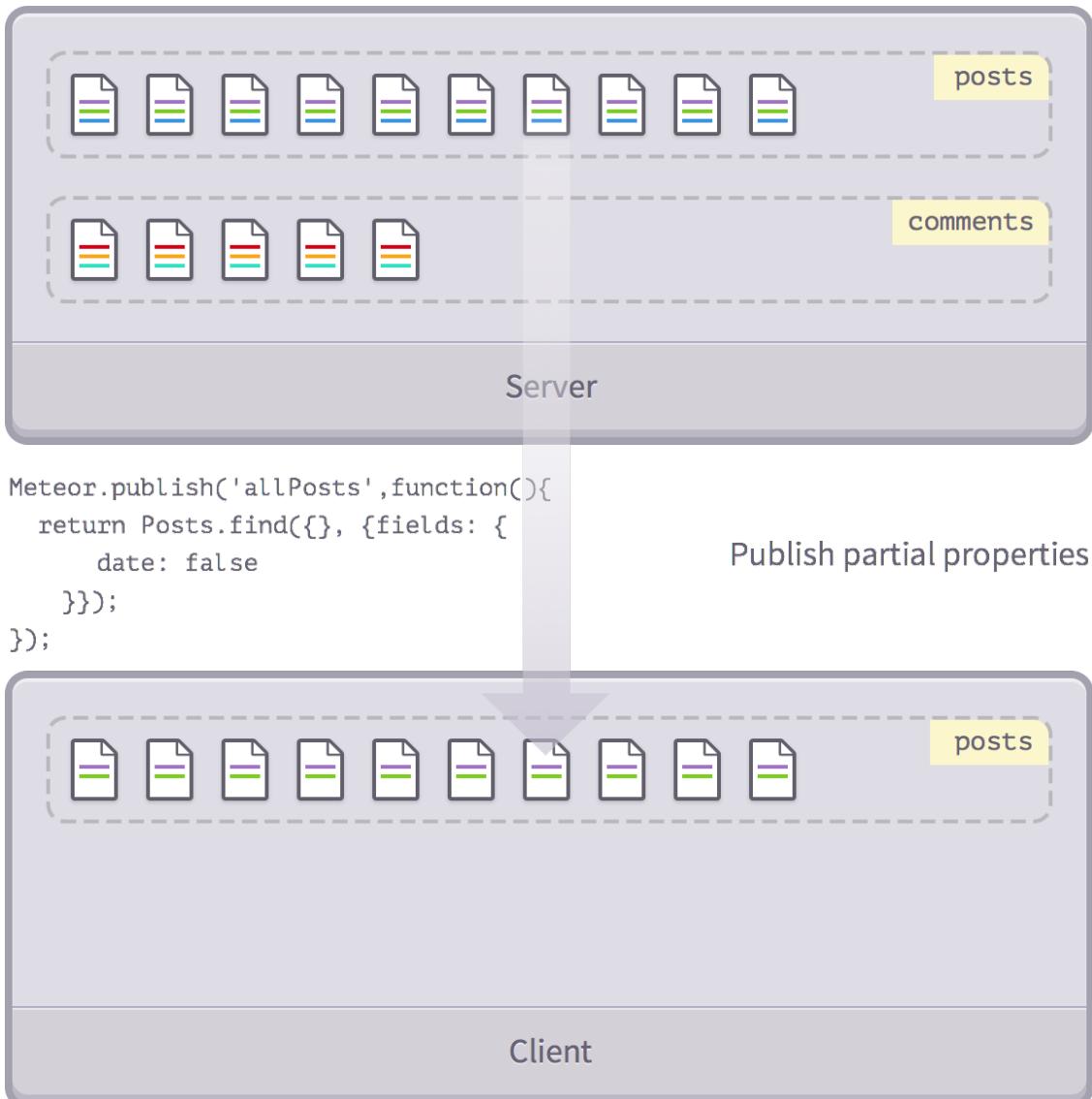
- Se comprueba el nombre de la colección en el servidor.
- Toma todos los documentos que coinciden con el cursor y los envía al cliente en una colección del *mismo nombre*. (Para hacer esto, utiliza `.added()`).
- Cada vez que se añade, elimina o modifica un documento, envía esos cambios a la colección del lado del cliente. (Para hacerlo, utiliza `.observe()` en el cursor y `.added()`, `.changed()` y `.removed()`).

Así, en el ejemplo anterior, podemos asegurar que el usuario solo tiene en la caché, los posts en los que está interesado (los escritos por Tom).

Hemos visto cómo publicar solo algunos de nuestros posts, pero ¡todavía podemos seguir recortando! Vamos a ver cómo publicar sólo algunas *propiedades*.

Al igual que antes, vamos a utilizar `find()` para devolver un cursor, pero esta vez vamos a excluir ciertos campos:

```
Meteor.publish('allPosts', function(){
  return Posts.find({}, {fields: {
    date: false
 }});
});
```



Por supuesto, también podemos combinar ambas técnicas. Por ejemplo, si quisieramos devolver todos los posts de Tom, dejando de lado sus fechas, escribiríamos:

```

Meteor.publish('allPosts', function(){
  return Posts.find({author:'Tom'}, {fields: {
    date: false
 }});
});

```

Hemos visto cómo pasar de publicar todas las propiedades de todos los documentos de todas las colecciones (con `autopublish`) a publicar solo *algunas* propiedades de *algunos* documentos de *algunas* colecciones.

Esto cubre los fundamentos de lo que se puede hacer con las publicaciones en Meteor, y estas técnicas tan sencillas deberían servir para la gran mayoría de casos de uso.

Aún así, en ocasiones, tendrás que ir más allá combinando, vinculando, o fusionando publicaciones. ¡Todo esto lo veremos más adelante en uno de los capítulos del libro!

Ahora que tenemos una lista de posts (que eventualmente serán enviados por los usuarios), necesitamos una página individual donde nuestros usuarios puedan discutir sobre cada post.

Nos gustaría que estas páginas fueran accesibles a través de un enlace con una URL *permanente* de la forma `http://myapp.com/posts/xyz` (donde `xyz` es un identificador `_id` de MongoDB) que sea única para cada post.

Esto significa que necesitaremos algún tipo de *enrutamiento* o *routing* para analizar lo que hay dentro de la barra de direcciones del navegador y mostrar el contenido correcto.

Iron Router es un paquete de enrutado que ha sido concebido específicamente para aplicaciones Meteor.

No solo ayuda con el enrutamiento (creación de rutas), sino también puede hacerse cargo de filtros (asignar acciones a algunas de estas rutas) e incluso administrar suscripciones (control de qué ruta tiene acceso a qué datos). (Nota: Iron Router ha sido desarrollado en parte por Tom Coleman, coautor de este libro).

En primer lugar, vamos a instalar el paquete desde Atmosphere:

```
meteor add iron:router
```

Terminal

Este comando descarga e instala el paquete iron-router dentro de nuestra aplicación. Hay que tener en cuenta que a veces puede ser necesario reiniciar la aplicación (con `ctrl+c` para parar y `meteor` para iniciar de nuevo) antes de poder usar algunos paquetes.

En este capítulo vamos a tocar un montón de características del Router. Si tienes experiencia con un framework como Rails, ya estarás familiarizado con la mayoría de estos conceptos. Si no, aquí hay un glosario para ponerte al día:

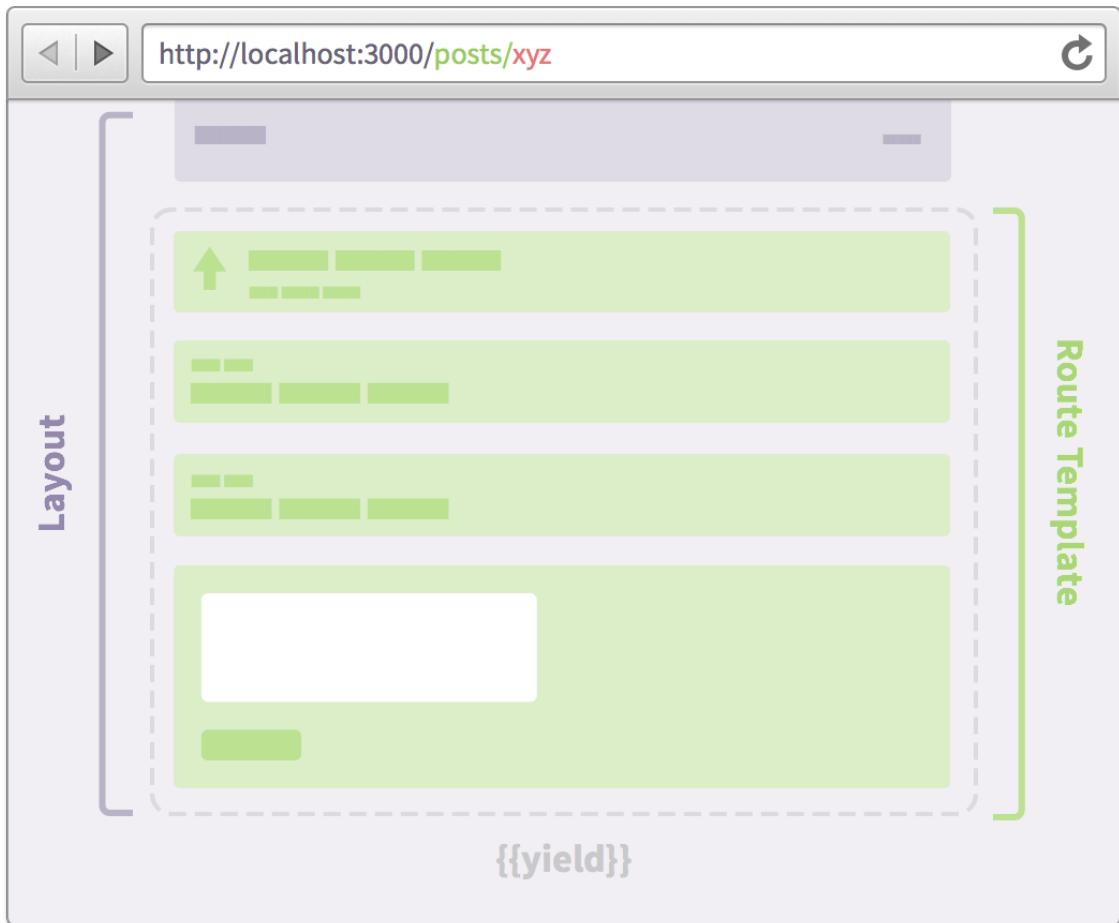
- **Routes:** Una ruta es la pieza de construcción básica del enrutamiento. Es básicamente el conjunto de instrucciones que le dicen a la aplicación a dónde ir y qué hacer cuando se encuentra con una URL.
- **Paths:** Un path es una dirección URL dentro de la aplicación. Puede ser estática (`/terms_of_service`) o dinámica (`/posts/xyz`), e incluso puede incluir parámetros de consulta (`/search?Keyword=meteor`).
- **Segments:** Las diferentes partes de un Path, delimitadas por barras inclinadas (`/`).
- **Hooks:** Son acciones que nos gustaría realizar antes, después o incluso durante el proceso de enrutamiento. Un ejemplo típico sería comprobar si el usuario tiene las credenciales adecuadas antes de mostrar una página.
- **Filters:** Son simplemente Hooks o acciones que se definen de forma global para una o más rutas.
- **Route Templates:** Cada ruta debe apuntar a una plantilla. Si no se especifica una, el router buscará una plantilla con el mismo nombre que la ruta por defecto.
- **Layouts:** Puedes pensar en los layouts como si fueran “marcos” para tu contenido. Contienen todo el código HTML que envuelve la plantilla actual, y seguirá siendo el mismo, aunque la plantilla cambie.
- **Controllers:** Algunas veces, nos daremos cuenta de que muchas de nuestras plantillas utilizan los mismos parámetros. En lugar de duplicar el código, podemos dejar que todas estas rutas se hereden desde un solo *controlador de enrutamiento* que contendrá toda la lógica necesaria.

Para obtener más información acerca de Iron Router, echa un vistazo a la[documentación completa en GitHub](#).

Hasta ahora, hemos construido nuestro diseño usando una plantilla fija (como `{ {> postsList}}`). Así que, aunque el contenido de nuestra aplicación puede cambiar, la estructura básica de la página es siempre la misma: una cabecera, con una lista de posts debajo de ella.

Iron Router nos permite romper este molde al tomar el control de lo que se muestra en el interior de la etiqueta HTML `<body>`. Por eso no vamos a definir el contenido como lo haríamos con una página HTML normal. En vez de eso, vamos a indicar al router que apunte a una plantilla especial que contiene un ayudante `{ {> yield}}`.

El ayudante `{ {> yield}}` definirá una zona dinámica especial que mostrará automáticamente lo que corresponde a la ruta actual (a modo de convención, llamaremos a esta plantilla especial “route template” o “plantilla de ruta”):



Empezaremos creando nuestro layout y añadiendo el ayudante `{{> yield}}`. En primer lugar, vamos a eliminar la etiqueta `<body>` del fichero `main.html`, y movemos su contenido a su propia plantilla, `layout.html` (que colocaremos dentro del directorio `client/templates/application`).

Iron Router se ocupará de insertar nuestro layout en nuestro `main.html` adelgazado, que ahora quedará así:

```
<head>
  <title>Microscope</title>
</head>
```

`client/main.html`

Mientras que el nuevo fichero `layout.html`, contendrá ahora el diseño exterior de la aplicación:

```
<template name="layout">
<div class="container">
  <header class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <a class="navbar-brand" href="/">Microscope</a>
    </div>
  </header>
  <div id="main">
    {{> yield}}
  </div>
</div>
</template>
```

Te habrás dado cuenta de que hemos cambiado la inclusión de la plantilla `postsList` con una llamada al ayudante `yield`.

Después de este cambio, nuestra pestaña del navegador mostrará la página de ayuda de Iron Router. Esto es debido a que no le hemos dicho al router qué debe hacer con la URL `/`, por lo que simplemente sirve una plantilla vacía.

Para comenzar, podemos recuperar el comportamiento anterior mapeando la URL raíz `/` a la plantilla `postsList`. Vamos a crear un nuevo fichero `router.js` dentro del directorio `/lib` en la raíz de nuestro proyecto:

```
Router.configure({
  layoutTemplate: 'layout'
});

Router.route('/', {name: 'postsList'});
```

lib/router.js

Hemos hecho dos cosas importantes. En primer lugar, le hemos dicho al router que utilice el `layout` que hemos creado como diseño predeterminado para todas las rutas.

En segundo lugar, hemos definido una nueva ruta llamada `postsList` y la hemos mapeado a `/`.

/lib

Meteor garantiza que cualquier cosa que pongamos dentro de la carpeta `/lib` se cargará antes que cualquier otra cosa de la aplicación (con la posible excepción de los smart packages). Esto hace que sea un gran lugar para poner cualquier código auxiliar que debe estar disponible en todo momento.

Solo una pequeña advertencia: ten en cuenta que, dado que la carpeta `/lib` no está dentro ni de `/client` ni de `/server`, sus contenidos estarán disponibles para ambos entornos.

Vamos a aclarar un poco las cosas. Hemos llamado a nuestra ruta `postsList`, pero también tenemos una plantilla llamada `postsList`. Entonces, ¿qué está pasando?

De forma predeterminada, Iron Router buscará una plantilla con el mismo nombre que la ruta. De hecho, incluso intentará buscar un camino basado en el nombre de la `url` que proporciones. Aunque no funcionará en este caso particular (ya que nuestra ruta es `/`), Iron Router podría encontrar la plantilla correcta si usamos `http://localhost:3000/postsList` como nuestra url.

Te estarás preguntando por qué necesitamos nombrar nuestras rutas. Hacerlo nos permite utilizar algunas características del Iron Router que hacen que sea más fácil construir enlaces dentro de nuestra aplicación. La más útil es el ayudante de Spacebars `{{pathFor}}`, que devuelve los componentes del `path` de cualquier ruta.

Queremos que el enlace principal apunte de nuevo a la lista de mensajes, así que en vez de especificar una URL estática `/`, podemos utilizar el ayudante Spacebars. El resultado final será el mismo, pero tendremos más flexibilidad porque el ayudante siempre obtendrá la dirección URL correcta, incluso si posteriormente cambiamos el path de la ruta en la configuración del router.

```
<header class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <a class="navbar-brand" href="{{pathFor 'postsList'}}>Microscope</a>
  </div>
</header>

//...
```

client/templates/application/layout.html

Enrutado básico.

[Ver en GitHub](#)

[Lanzar instancia](#)

Si despliegas la versión actual de la aplicación (o lanzas la instancia mediante el enlace anterior), te darás cuenta de que la lista aparece vacía durante unos instantes antes de que aparezcan los posts. Esto es porque cuando la página se carga por primera vez, no hay posts para mostrar hasta que se completa la suscripción `posts` obteniendo los datos enviados desde el servidor.

Tendríamos una mejor experiencia de usuario si proporcionáramos alguna información visual de que algo está pasando, y que el usuario debe esperar un poco.

Por suerte, Iron Router proporciona una forma fácil de hacerlo: podemos decirle que espere (`waitOn`) a la suscripción.

Empezaremos moviendo nuestra suscripción `posts` desde `main.js` hasta el router:

```
Router.configure({
  layoutTemplate: 'layout',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
```

lib/router.js

Lo que estamos diciendo aquí es que para *cualquier* ruta del sitio (ahora mismo solo tenemos una, ¡pero pronto vendrán más!), queremos suscribirnos a la suscripción `posts`.

La diferencia clave entre esto y lo que teníamos antes (cuando la suscripción estaba en `main.js`, que **ahora debería estar vacío y lo podemos eliminar**), es que ahora Iron Router sabe cuando la ruta está “preparada” (“ready”) – esto es,

cuando la ruta tiene los datos que necesita para renderizarse.

Saber cuando la ruta `postsList` está lista no nos sirve de mucho si de todas formas vamos a estar mostrando una plantilla vacía. Afortunadamente, Iron Router proporciona una forma de retrasar el renderizado de una plantilla hasta que la ruta esté preparada, y mostrar una plantilla de cargando en su lugar (`loading`):

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
```

lib/router.js

Fíjate que como hemos definido nuestra función `waitOn` de forma global a nivel del router, esto solo ocurrirá una sola vez cuando el usuario acceda por primera vez a la aplicación. Después de esto, los datos ya estarán cargados en la memoria del navegador y el router no necesitará volver a esperar de nuevo.

La pieza final del rompecabezas es la plantilla de carga. Vamos a utilizar el paquete `spin` para crear un buen efecto de carga animada. Lo añadimos con `meteor add sacha:spin`, y luego creamos la plantilla `loading` de carga en el directorio `client/templates/includes`:

```
<template name="loading">
  {{>spinner}}
</template>
```

client/templates/includes/loading.html

Ten en cuenta que `{{>spinner}}` está contenido en el paquete `spin`. A pesar de que proviene de “fuera” de nuestra aplicación, podemos incluirlo como cualquier otra plantilla.

Por lo general es una buena idea esperar a las suscripciones, no solo por la experiencia de usuario, sino también porque significa que podemos asumir con seguridad que los datos estarán siempre disponibles dentro de una plantilla. Esto elimina la necesidad de enredarse con plantillas que se muestran antes de que los datos que usan estén disponibles, cosa que a menudo requiere soluciones difíciles.

Esperando a la
suscripción.

[Ver en GitHub](#)

[Lanzar instancia](#)

La reactividad es una parte fundamental de Meteor, y aunque todavía queda un poco para conocerla, nuestra plantilla de carga nos da un primer vistazo a este concepto.

Redireccionar a una plantilla de carga de datos si no se ha cargado todavía está muy bien, pero ¿cómo sabe el router cuándo redirigir al usuario una vez han llegado los datos?

Por ahora, solo diremos que aquí es exactamente donde entra en juego la reactividad. Pero no te preocupes, aprenderás más sobre ella muy pronto!

Ahora que hemos visto cómo enrutar hacia la plantilla `postsList`, vamos a configurar una ruta para mostrar los detalles de un solo post.

Solo hay un problema: no podemos continuar definiendo rutas una por una para cada post, ya que podría haber cientos de ellos. Así que tendremos que crear una ruta *dinámica* y hacer que se vea esta nos muestre cualquier post que queremos.

Para empezar, vamos a crear una nueva plantilla `post_page.html` que simplemente muestra la misma plantilla para un post que hemos utilizado anteriormente en la lista de posts.

```
<template name="postPage">
  <div class="post-page page">
    {{> postItem}}
  </div>
</template>
```

client/templates/posts/post_page.html

Más adelante añadiremos más elementos a esta plantilla (como los comentarios), pero, por ahora, solo la vamos a usar para mostrar `{{> PostItem}}`.

Ahora vamos a crear otra ruta con nombre, esta vez, mapeando URLs de la forma `/posts/<ID>` hacia la plantilla `postPage`:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
Router.route('/posts/:_id', {
  name: 'postPage'
});
```

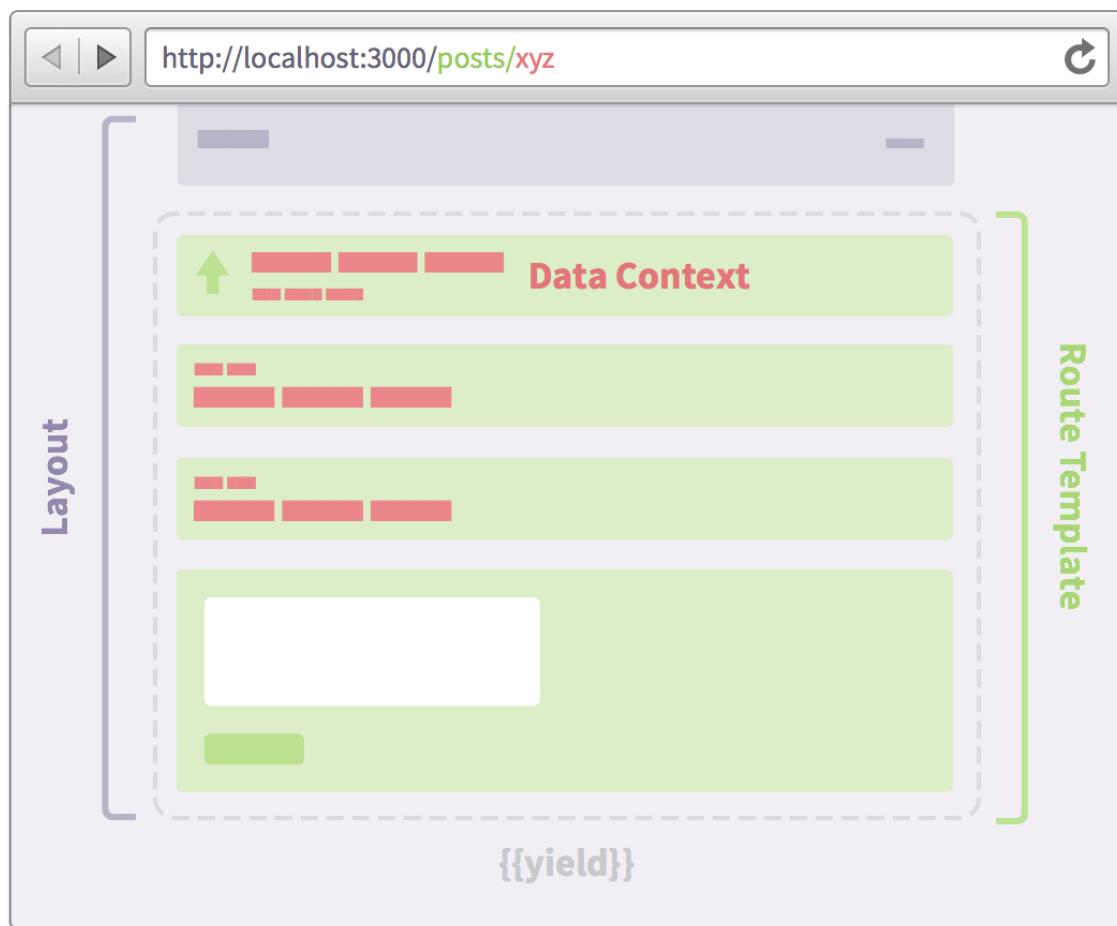
lib/router.js

La sintaxis especial `:_id` le dice dos cosas al router: primero, que encuentre cualquier ruta de la forma `/posts/xyz/`, donde “xyz” puede ser cualquier cadena. En segundo lugar, poner lo que encuentra dentro de una propiedad `_id` en el vector de parámetros del router.

Ten en cuenta que usamos el `_id` como cadena porque así lo queremos. El router no tiene manera de saber si le pasamos un `_id` real, o simplemente una cadena de caracteres al azar.

Ya enrutamos a la plantilla correcta, pero todavía nos falta algo: el router conoce el `_id` del post que nos gustaría ver, pero la plantilla todavía no tiene ni idea. Entonces, ¿cómo solucionamos este problema?

Afortunadamente, el router integra una solución inteligente: permite especificar el **contexto de datos** de una plantilla. Puedes pensar en el contexto de datos como lo que rellena un delicioso pastel hecho de plantillas y diseños. En pocas palabras, son los datos con los que llenamos la plantilla:



En nuestro caso, podemos obtener el contexto de datos correcto mediante la búsqueda de nuestro post basado en el `_id` que recibimos de la URL:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});
```

lib/router.js

De esta forma, cada vez que un usuario accede a esta ruta, encontraremos el post adecuado y lo pasaremos a la plantilla. Recuerda que `findOne` devuelve un solo post, el que coincide con la consulta, y que proporcionar solo un `_id` como argumento es una abreviatura de `{_id: id}`.

Dentro de la función `data` de una ruta, `this` se corresponde con la ruta actual, y podemos usar `this.params` para acceder a las propiedades de la ruta (que habíamos indicado con el prefijo `:` dentro de nuestro `path`).

Al establecer el contexto de datos de una plantilla, se puede controlar el valor de `this` dentro de los ayudantes de la plantilla.

Esto se hace implícitamente con el iterador `{{#each}}`, que ajusta automáticamente el contexto de datos de cada iteración para el elemento que se está iterando:

```
{{#each widgets}}
  {{> widgetItem}}
{{/each}}
```

Pero también podemos hacerlo explícitamente utilizando `{{#with}}`, que simplemente dice “toma este objeto, y le aplicas la siguiente plantilla”. Por ejemplo, se puede escribir:

```
{{#with myWidget}}
  {{> widgetPage}}
{{/with}}
```

Resulta que se consigue el mismo resultado pasando el contexto como un argumento en la llamada a la plantilla. Así que el bloque de código anterior se puede reescribir como:

```
{{> widgetPage myWidget}}
```

Para una exploración con profundidad sobre los contextos de datos sugerimos [leer nuestro blog](#) sobre este tema.

Por último, crearemos un nuevo botón “Discuss” que enlazará a nuestra página individual del post. De nuevo, podríamos hacer algo como ``, pero es mucho más fiable utilizar un ayudante de ruta.

Hemos llamado a la ruta al post `postPage`, así que podemos usar el ayudante `{{pathFor 'postPage'}}`:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}>{{title}}</a><span>{{domain}}</span></h3>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  </div>
</template>
```

client/templates/posts/post_item.html

Ruta para un único
post.

[Ver en GitHub](#)

[Lanzar instancia](#)

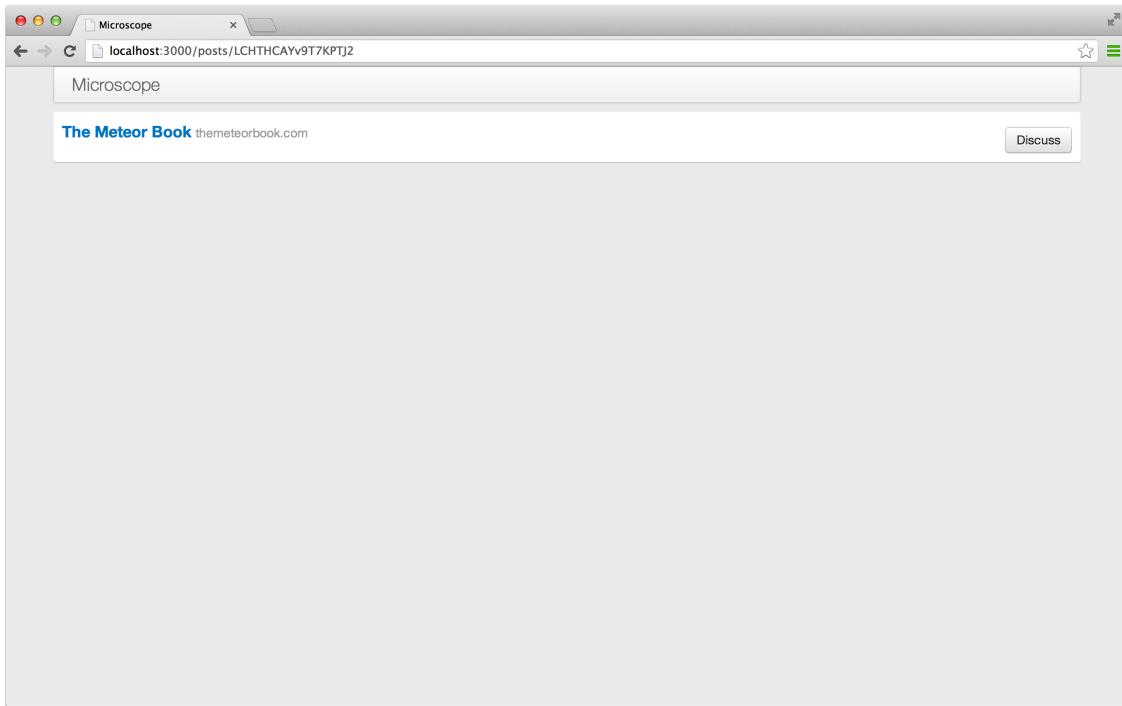
Pero espera, ¿cómo sabe el router dónde conseguir la parte `xyz` en `/posts/xyz`? Después de todo, no le hemos pasado ninguna `_id`.

Resulta que Iron Router es lo suficientemente inteligente como para averiguarlo por sí mismo. Le estamos diciendo que use la ruta `postPage`, y el router sabe que esta ruta requiere un `_id` de algún tipo (así es como hemos definido nuestro `path`).

Así que el router buscará este `_id` en el lugar más lógico: el contexto de datos del ayudante `{{pathFor 'postPage'}}`, en otras palabras: `this`. Y da la casualidad de que nuestro `this` corresponde a un post, que (¡sorpresa!) tiene una propiedad `_id`.

De forma alternativa, se puede especificar el lugar donde tiene que buscar el `_id`, pasando un segundo argumento al ayudante (es decir, `{{pathFor 'postPage' someOtherPost}}`). Un uso práctico sería, por ejemplo, conseguir los enlaces a los posts anterior y siguiente en una lista.

Para ver si todo funciona correctamente, navega a la lista de posts y haz clic en uno de los enlaces ‘Discuss’. Deberías ver algo como esto:



Una cosa que hay que tener en cuenta es que estos cambios en las URLs suceden gracias a [HTML5 pushState](#). El router recoge los clics en URL internas, y evita que el navegador salga fuera de la aplicación haciendo los cambios necesarios en su estado. Si todo funciona correctamente la página debería cambiar instantáneamente. De hecho, a veces las cosas cambian tan rápido que podría ser necesario añadir algún tipo de transición. Esto está fuera del alcance de este capítulo, aunque, no obstante, es un tema interesante.

No olvidemos que el enrutamiento funciona de ambas formas: podemos cambiar la URL cuando visitamos una página, pero también podemos mostrar una página cuando cambiamos *la URL*. Por lo que tenemos que pensar que pasa si alguien introduce una URL errónea.

Menos mal que Iron Router se preocupa por esto a través de la opción `notFoundTemplate`.

Primero, crearemos una plantilla que muestre un simple error 404:

```
<template name="notFound">
<div class="not-found page jumbotron">
  <h2>404</h2>
  <p>Sorry, we couldn't find a page at this address.</p>
</div>
</template>
```

Después, sencillamente le decimos a Iron Router que use esta plantilla:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

//...
```

lib/router.js

Para probar la nueva página de error, puedes intentar introducir una URL aleatoria como

```
http://localhost:3000/nothing-here .
```

Pero un momento, ¿qué pasa si alguien introduce una URL de la forma `http://localhost:3000/posts/xyz`, donde `xyz` no es un identificador `_id` de post válido? Esto es una ruta válida, pero no apunta a ningún dato.

Afortunadamente, Iron Router es lo suficientemente inteligente para saber esto si definimos un *hook* especial `dataNotFound` al final de `router.js`:

```
//...
Router.onBeforeAction('dataNotFound', {only: 'postPage'});
```

lib/router.js

Esto le dice a Iron Router que muestre la página de no encontrado, no solo cuando la ruta sea inválida, si no también para la ruta `postPage` cuando la función `data` devuelva un objeto falso (o `null`, `false`, `undefined` o vacío).

Añadida la plantilla de no encontrado.

[Ver en GitHub](#)

[Lanzar instancia](#)

Te sorprenderías sobre la historia detrás del nombre “Iron Router”. Según el autor Chris Mather, viene del hecho de que los meteoritos están compuestos principalmente de hierro.

Meteor es un framework reactivo. Esto significa que cuando cambian los datos, cambian cosas de la aplicación sin tener que hacer nada de forma explícita.

Ya lo hemos visto en acción viendo cómo cambian nuestras plantillas cuando cambian los datos y las rutas.

En capítulos posteriores veremos más en profundidad cómo funciona todo esto, pero ahora, nos gustaría introducir algunas características básicas de la reactividad, que son muy útiles en todas las aplicaciones.

Ahora mismo, en Microscope, el estado actual de la aplicación está contenido en su totalidad en la URL que se está mostrando (y en la base de datos).

Pero en muchos casos, necesitarás almacenar valores de estado que solo son relevantes en la versión de la aplicación para usuario actual (por ejemplo, si algún elemento se muestra o está oculto). Usar la Sesión es una forma conveniente de hacerlo.

La sesión es un almacén global de datos reactivo. Es global en el sentido de que es un objeto global: solo hay una sesión, y esta es accesible desde todas partes. Las variables globales se suelen ver como algo malo, pero en este caso la sesión se utiliza como un bus de comunicación central para diferentes partes de la aplicación.

La sesión está disponible en todas partes en el cliente como el objeto `Session`. Para establecer un valor en la sesión, puedes llamar a:

```
Session.set('pageTitle', 'A different title');
```

Consola del navegador

Puedes leer los datos de nuevo con `Session.get('mySessionProperty')`. Hemos dicho que la Sesión es una fuente de datos reactiva, lo que significa que si lo pones en un ayudante, verías un cambio en el navegador cuando cambia la variable de sesión.

Para probarlo, añade el siguiente código a la plantilla `layout`:

```
<header class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <a class="navbar-brand" href="{{pathFor 'postsList'}}">{{pageTitle}}</a>
  </div>
</header>
```

client/templates/application/layout.html

```
Template.layout.helpers({  
  pageTitle: function() { return Session.get('pageTitle'); }  
});
```

client/templates/application/layout.js

Ten en cuenta que el código mostrado en las barras laterales no forma parte del flujo principal del libro. Por lo que debería crear una nueva rama (si estás usando Git), o si no, asegurarte de deshacer los cambios al final del capítulo.

La recarga automática de Meteor (“recarga de código en caliente” o HCR, de Hot Code Reload) preserva las variables de sesión, por lo que ahora debes ver “A different title” en la barra de navegación. Si no es así, solo tienes que escribir el comando `Session.set()` de nuevo.

Si lo cambiamos de nuevo (desde la consola del navegador), debemos ver que se visualiza otro título:

```
➤ Session.set('pageTitle', 'A brand new title');
```

Consola del navegador

La sesión está disponible a nivel global, por lo que los cambios se pueden hacer desde cualquier lugar de la aplicación. Esto nos da una gran cantidad de potencia y flexibilidad, pero también puede ser una trampa si se usa demasiado.

De todas formas, es importante apuntar que el objeto de Session *no* es compartido entre distintos usuarios, ni siquiera entre distintas pestañas del navegador. Esto es por lo que si abres tu aplicación en una nueva pestaña, te encontrarás con una página en blanco.

Si modificas una variable de sesión con `Session.set()` pero la estableces al mismo valor, Meteor es lo suficientemente inteligente como para eludir la cadena reactiva, y evitar las llamadas a métodos innecesarios.

Hemos visto un ejemplo de una fuente de datos reactiva, y la hemos visto en acción dentro de un ayudante de plantilla. Pero mientras que algunos contextos en Meteor (como ayudantes de plantilla) son inherentemente reactivos, la mayoría de código de la aplicación sigue siendo el viejo y simple JavaScript.

Supongamos que tenemos el siguiente fragmento de código en algún lugar de nuestra aplicación:

```
helloWorld = function() {
  alert(Session.get('message'));
}
```

A pesar de que llamamos a una variable de sesión, el *contexto* en el que se hace no es reactivo, lo que significa que no vamos a ver `alerts` cada vez que cambia su valor.

Aquí es dónde entra en juego **Autorun**. Como su nombre indica, el código dentro de un bloque `autorun` se ejecutará automáticamente cada vez que cambien las fuentes de datos reactivas utilizadas dentro.

Prueba a escribir esto en la consola del navegador:

```
➤ Tracker.autorun( function() { console.log('Value is: ' + Session.get('pageTitle')); } );
Value is: A brand new title
```

Consola del navegador

Como era de esperar, el bloque de código situado en el interior de `autorun` se ejecuta una vez, mostrando los datos por la consola. Ahora, vamos a intentar cambiar el título:

```
➤ Session.set('pageTitle', 'Yet another value');
Value is: Yet another value
```

Consola del navegador

¡Magia! Al cambiar el valor, `autorun` sabe que tiene que ejecutar su contenido de nuevo, volviendo a mostrar el nuevo valor por la consola.

Volviendo a nuestro ejemplo anterior, si queremos activar una nueva alerta cada vez que cambien variables de sesión, lo único que tenemos que hacer es envolver nuestro código en un bloque `autorun`:

```
Tracker.autorun(function() {
  alert(Session.get('message'));
});
```

Como acabamos de ver, los `autorun` pueden ser muy útiles para rastrear fuentes de datos reactivas y reaccionar inmediatamente ante ellos.

Durante el desarrollo de Microscope, hemos estado aprovechando una de las características de ahorro de tiempo que proporciona Meteor: La recarga de código en caliente (HCR). Cada vez que guardamos uno de nuestros archivos fuente, Meteor detecta los cambios y reinicia el servidor de forma transparente, informando a todos los clientes para que recarguen la página.

Es similar a una recarga automática de página, pero con una diferencia importante.

Para entenderlo, vamos a restablecer la variable de sesión que hemos estado utilizando:

```
Session.set('pageTitle', 'A brand new title');
Session.get('pageTitle');
'A brand new title'
```

Consola del navegador

Si recargáramos la página manualmente se perderían las variables de sesión (porque estaríamos creando una nueva). Pero si provocamos una recarga en caliente (por ejemplo, guardando uno de nuestros archivos de código), la página volverá a cargar, pero todavía tendremos el valor de la variable de sesión. ¡Pruébalo!

```
Session.get('pageTitle');
'A brand new title'
```

Consola del navegador

Así que si utilizamos variables de sesión para hacer un seguimiento de lo que está haciendo el usuario, el HCR debe ser prácticamente transparente para el usuario, ya que preserva el valor de todas las variables de sesión. Esto nos permite desplegar nuevas versiones de nuestra aplicación, ya en producción, con la seguridad de que no molestaremos mucho a nuestros usuarios.

Considera esto un momento. Si podemos llegar a mantener el estado entre la URL y la sesión, podemos cambiar de forma transparente el *código que está corriendo* por debajo con una mínima interrupción.

Ahora vamos a comprobar lo que pasa cuando refrescamos la página de forma manual:

```
Session.get('pageTitle');
null
```

Browser console

Hemos perdido la sesión. En un HCR, Meteor guarda la sesión en el almacenamiento local del navegador y lo carga de nuevo tras la recarga. Esto no significa que el comportamiento de recarga explícita no tenga sentido: si un usuario recarga la página, es como si navega de nuevo a la misma URL, y el estado, debe restablecerse al que vería cualquier usuario que visita esa URL.

Las lecciones más importantes en todo esto son:

1. Guarda siempre el estado en la sesión o en la URL para no molestar mucho a los usuarios cuando ocurre una recarga en caliente.
2. Almacena cualquier estado que deba estar compartido entre usuarios *dentro de la propia URL*.

Con esto concluye nuestra exploración de la sesión, uno de las características más útiles de Meteor. No olvides deshacer cualquier cambio en el código antes de pasar al siguiente capítulo.

Hasta el momento, hemos logrado crear y mostrar algunos datos estáticos y conectarlo todo en un prototipo simple.

Incluso hemos visto cómo nuestra interfaz de usuario es sensible a los cambios en los datos, y que los cambios aparecen inmediatamente cuando se insertan o cambian los datos. Aún así, nuestro sitio está limitado por el hecho de que no podemos introducir datos. De hecho, ¡ni siquiera tenemos usuarios todavía!

Veamos cómo arreglamos esto.

En la mayoría de los frameworks web, agregar cuentas de usuario es un problema familiar. Hay que hacerlo en casi todos los proyectos, pero nunca resulta tan fácil como quisiéramos. Y si además hay que tratar con OAuth u otros esquemas de autenticación de terceros, las cosas tienden a ponerse feas rápidamente.

Por suerte, Meteor nos ampara. Gracias a la forma en la que los paquetes Meteor pueden contribuir al código del servidor (JavaScript) y al del cliente (JavaScript, HTML y CSS), podemos obtener un sistema de cuentas casi sin esfuerzo.

Podríamos usar el paquete para cuentas de usuario (`meteor add accounts-ui`), pero como hemos construido toda nuestra aplicación con Bootstrap, vamos a utilizar `ian:accounts-ui-bootstrap-3` (la única diferencia es el estilo). En la línea de comandos, tecleamos:

```
meteor add ian:accounts-ui-bootstrap-3
meteor add accounts-password
```

Terminal

Estos dos comandos hacen accesibles las plantillas para cuentas de forma que podemos incluirlas en nuestro sitio usando el ayudante `{{> loginButtons}}`. Un consejo muy útil: se puede controlar en qué lado aparece el desplegable de inicio de sesión con el atributo `align` (por ejemplo: `{{> loginButtons align="right"}}`).

Vamos a añadir los botones a nuestra cabecera. Y puesto que la cabecera está empezando a crecer, vamos a darle más espacio en su propia plantilla (la pondremos en el directorio `client/templates/includes/`). Estamos utilizando código y clases **como se indica en Bootstrap** para que todo se vea mejor:

```
<template name="layout">
<div class="container">
{{> header}}
<div id="main">
{{> yield}}
</div>
</div>
</template>
```

`client/templates/application/layout.html`

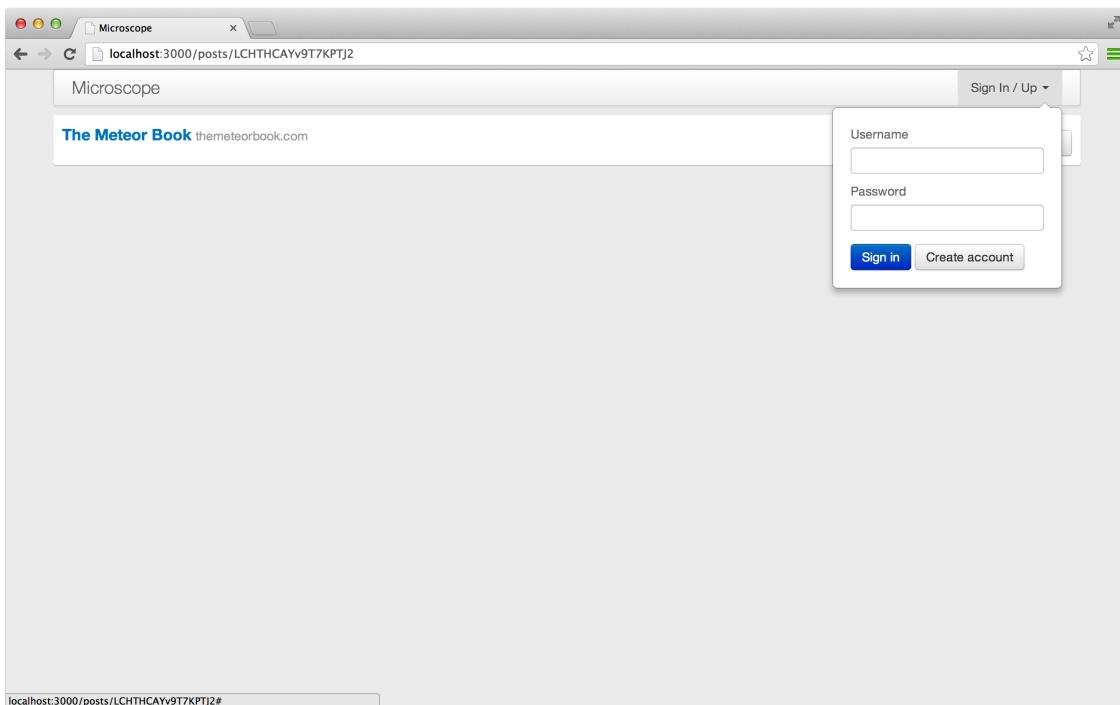
```

<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'postsList'}}>Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>

```

client/templates/includes/header.html

Ahora ya podemos ver los botones de acceso en la esquina superior derecha.



Podemos usarlos para iniciar sesión, solicitar un cambio de contraseña, y todo lo que se necesita para gestionar cuentas basadas en contraseñas.

Para decirle a nuestro sistema de cuentas que queremos que los usuarios accedan al sistema a través de solo un nombre de usuario, simplemente añadimos un bloque de configuración en un nuevo archivo `config.js` dentro de

`client/helpers/`

```
Accounts.ui.config({  
  passwordSignupFields: 'USERNAME_ONLY'  
});
```

client/helpers/config.js

Añadidas cuentas de usuario y una plantilla en la cabecera

[Ver en GitHub](#)

[Lanzar instancia](#)

Adelante, regístrate para obtener una cuenta: el botón “Sign In” cambiará para mostrar el nombre de usuario. Esto confirma que has creado una cuenta. Pero, ¿de dónde vienen los datos de la cuenta?

Cuando añadimos el paquete `accounts`, Meteor crea una nueva colección, que se puede acceder desde `Meteor.users`. Para verlo, abre la consola del navegador y escribe:

```
➤ Meteor.users.findOne();
```

Consola del navegador

La consola devuelve un objeto que representa el usuario. Si lo inspeccionamos un poco, veremos que contiene nuestro nombre de usuario, así como un identificador único `_id`. También se puede obtener el usuario que ha iniciado sesión con `Meteor.user()`.

Ahora, nos registramos con otro usuario y ejecutamos lo siguiente en la consola del navegador:

```
➤ Meteor.users.find().count();
```

1

Consola del navegador

La consola devuelve 1. ¿No debería haber 2? ¿Se ha borrado el primero? Si intentas acceder con ese usuario, verás que no es así.

Vamos a mirar en la base de datos del servidor (`meteor mongo` en la terminal):

```
> db.users.count()  
2
```

Consola de Mongo

Definitivamente hay 2, pero, ¿porqué sólo se ve uno en el navegador?

Si pensamos de nuevo en el capítulo 4, recordarás que deshabilitamos la auto-publicación `autopublish`, dejamos de enviar todos los datos procedentes del servidor a las colecciones locales de cada cliente conectado. Tuvimos que crear parejas de publicaciones y suscripciones para intercambiar datos.

Sin embargo, no hemos creado ninguna clase de publicación para usuarios. Así que ¿cómo es que podemos ver esos datos?

La respuesta es que el paquete `accounts`, autopublica los datos básicos de la cuenta del usuario actual, de otra forma, el usuario no podría acceder nunca al sitio.

El hecho de que el paquete `accounts` sólo publica el usuario *actual* explica porqué un usuario no puede ver los detalles de la cuenta de los otros usuarios.

Así que solo se publica un objeto de usuario por usuario conectado (y ninguno cuando no está autenticado).

Es más, los documentos en el navegador no parecen contener los mismos campos que en el servidor. En Mongo, un usuario tiene un montón de datos. Para verlo, vuelve a la terminal de Mongo y escribe:

```
> db.users.find()
{
  "_id": "H5kKyxtbkLhmPgtqs",
  "createdAt": ISODate("2015-02-10T08:26:48.196Z"),
  "profile": {},
  "services": {
    "password": {
      "bcrypt": "$2a$10$yGPwo3/53IHsdf fdwe766roZviT03YBGl tJ0UG"
    },
    "resume": {
      "loginTokens": [
        {
          "when": ISODate("2015-02-10T08:26:48.203Z"),
          "hashedToken": "npxGH7Rmkuxcv098wzz+qR0/jHl0EAGWr0D9Zp0w="
        }
      ]
    },
    "username": "sacha"
}
```

Consola de Mongo

Por otro lado, en el navegador el objeto de usuario tiene muchos menos campos, como se puede ver escribiendo el comando equivalente:

```
➤ Meteor.users.findOne();
Object {_id: "kYdBd9hr3fWP GPci", username: "tmeasday"}
```

Consola del navegador

Este ejemplo nos muestra como una colección local puede ser un subconjunto seguro de la base de datos real. El usuario

conectado solo ve lo necesario para poder hacer el trabajo (en este caso, el nombre de usuario). Este es un patrón que debemos aprender porque nos será muy útil más adelante.

Eso no significa que no podamos hacer públicos más datos de usuario. Si lo necesitamos, podemos consultar la [documentación de Meteor](#) para ver cómo se hace.

Si las colecciones son la característica principal de Meteor, podemos decir que la **reactividad** es lo que hace útil esta característica.

Las colecciones trasforman radicalmente la forma en que la aplicación maneja cambios en los datos. En lugar de tener que comprobarlo manualmente (por ejemplo, con una llamada AJAX) y actualizar los cambios en el código HTML, con Meteor, los cambios pueden llegar en cualquier momento y aplicarse a la interfaz de usuario sin más complicaciones.

Vamos a verlo con detenimiento: entre bastidores, Meteor es capaz de cambiar *cualquier* parte de la interfaz de usuario cuando se actualiza una colección subyacente.

La forma *imperativa* (o “a mano”) de hacerlo sería utilizar `.observe()`, una función del cursor que dispara callbacks cuando los documentos seleccionados cambian. De esta forma, podríamos hacer cambios en el DOM (el HTML de nuestra página web) a través de esos callbacks. El código resultante sería algo como esto:

```
Posts.find().observe({
  added: function(post) {
    // when 'added' callback fires, add HTML element
    $('ul').append('<li id="' + post._id + '">' + post.title + '</li>');
  },
  changed: function(post) {
    // when 'changed' callback fires, modify HTML element's text
    $('ul li#' + post._id).text(post.title);
  },
  removed: function(post) {
    // when 'removed' callback fires, remove HTML element
    $('ul li#' + post._id).remove();
  }
});
```

Es probable que te des cuenta de que este tipo de código va a tender a hacerse muy complejo muy rápidamente. Imagínate lo que sería tratar con cambios en *cada uno de los atributos* del post, y tener que cambiar HTML complejo dentro de ``. Por no hablar de casos más extremos cuando empecemos a depender de múltiples fuentes de información que pueden cambiar en tiempo real.

deberíamos `observe()`

A veces es necesario usar el patrón anterior, sobre todo cuando se trabaja con widgets de terceros. Por ejemplo, imaginemos que queremos añadir o quitar puntos en un mapa en tiempo real dentro de una Colección (por ejemplo, para mostrar las localizaciones de los usuarios actualmente conectados).

En tal caso, tendrás que usar callbacks `observe()` para conseguir que el mapa “hable” con la colección Meteor y saber cómo reaccionar a los cambios en los datos. Por ejemplo, tendrías que confiar en los callbacks `added` y `removed` para llamar a los métodos `dropPin` o `removePin()` de la API del mapa.

esta forma, podemos definir la relación entre los objetos una sola vez y podemos estar seguros de que se mantendrán siempre sincronizados, en vez de tener que especificar el comportamiento de cada uno de los posibles cambios.

Este es un concepto realmente poderoso, ya que, en un sistema de tiempo real puede haber muchas entradas y todas pueden cambiar de forma impredecible. Con “declarativo”, queremos decir que, cuando se renderiza HTML basado en una o varias fuentes de datos reactivos, Meteor se hace cargo de la tarea de sincronizar las fuentes y, de forma transparente, asumir el trabajo sucio de mantener la interfaz de usuario actualizada.

Y todo esto, sólo para acabar diciendo que, en lugar de tener que pensar en callbacks tipo `observe`, Meteor nos permite escribir:

```
<template name="postsList">
  <ul>
    {{#each posts}}
      <li>{{title}}</li>
    {{/each}}
  </ul>
</template>
```

Y obtener la lista de posts con:

```
Template.postsList.helpers({
  posts: function() {
    return Posts.find();
  }
});
```

Cuando cambian los datos reactivos, es Meteor el que, entre bastidores, está creando callbacks `observe()`, y redibujando las secciones pertinentes del HTML.

Meteor es un framework reactivo y en tiempo real, pero *no todo* el código incluido en una aplicación Meteor es reactivo. Si así fuera, la aplicación entera se volvería a ejecutar cada vez que cambia algo. En vez de eso, la reactividad se limita a áreas específicas, que llamamos **computaciones**.

En otras palabras, una computación es un bloque de código que se ejecuta cada vez que cambia una de las fuentes de datos reactivos de las que depende. Si tienes una fuente reactiva (por ejemplo, una variable de sesión) y quieres responder reactivamente a ella, tendrás que crear una computación.

Ten en cuenta que, por lo general, no es necesario hacerlo de forma explícita, ya que, Meteor provee de una computación especial a cada una de nuestras plantillas (lo que significa que puedes estar seguro de que tus plantillas reflejarán los datos de los que dependen).

Todas las fuentes de datos reactivas hacen un seguimiento de todas las computaciones que las usan para poder avisarles de que su valor ha cambiado y deben “volver a computarse”. Para hacer esto, se llama a la función `invalidate()`.

Generalmente, las computaciones se establecen para volver a evaluar su contenido, y esto es lo que ocurre con las

computaciones que Meteor crea para las plantillas (aunque, además, se añade un poco más de magia para redibujar la página de manera más eficiente). Aunque, si es necesario, se puede tener más control sobre lo que hace una de estas computaciones, en la práctica, no va a ser necesario porque Meteor nos va a dar justo el comportamiento que vamos a necesitar.

Ahora que entendemos la teoría de las computaciones, configurar una te va a parecer desproporcionadamente fácil. Usaremos la función `Tracker.autorun` para encerrar un bloque de código en una computación y hacerlo reactivo:

```
Meteor.startup(function() {
  Tracker.autorun(function() {
    console.log('There are ' + Posts.find().count() + ' posts');
  });
});
```

Ten en cuenta que tenemos que envolver el bloque `Tracker` dentro de un bloque `Meteor.startup()` para asegurarnos que solo se ejecute una vez cuando Meteor ha terminado de cargar la colección `Posts`.

Entre bastidores, `autorun` crea una computación, y la configura para que se vuelva a evaluar cada vez que cambian las fuentes de datos de las que depende. El ejemplo es muy sencillo, simplemente muestra el número actual de posts en la consola. Como `Posts.find()` es una fuente de datos de reactiva, será esta la que se encargue de hacer que se vuelva a ejecutar la computación cada vez que cambie el número de posts.

```
> Posts.insert({title: 'New Post'});
There are 4 posts.
```

El resultado es que, de forma fácil y natural, podemos escribir código que usa datos reactivos, sabiendo que, por detrás, el sistema de dependencias se encargará de todo en todo momento.

Hemos visto lo fácil que es crear posts llamando a `Posts.insert` a través de la consola pero, no podemos esperar que nuestros usuarios hagan lo mismo.

Necesitamos construir algún tipo de interfaz de usuario para que los usuarios creen nuevas entradas en la aplicación.

Empezaremos definiendo una ruta para nuestra nueva página en `lib/router.js`:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

Router.onBeforeAction('dataNotFound', {only: 'postPage'});

```

`lib/router.js`

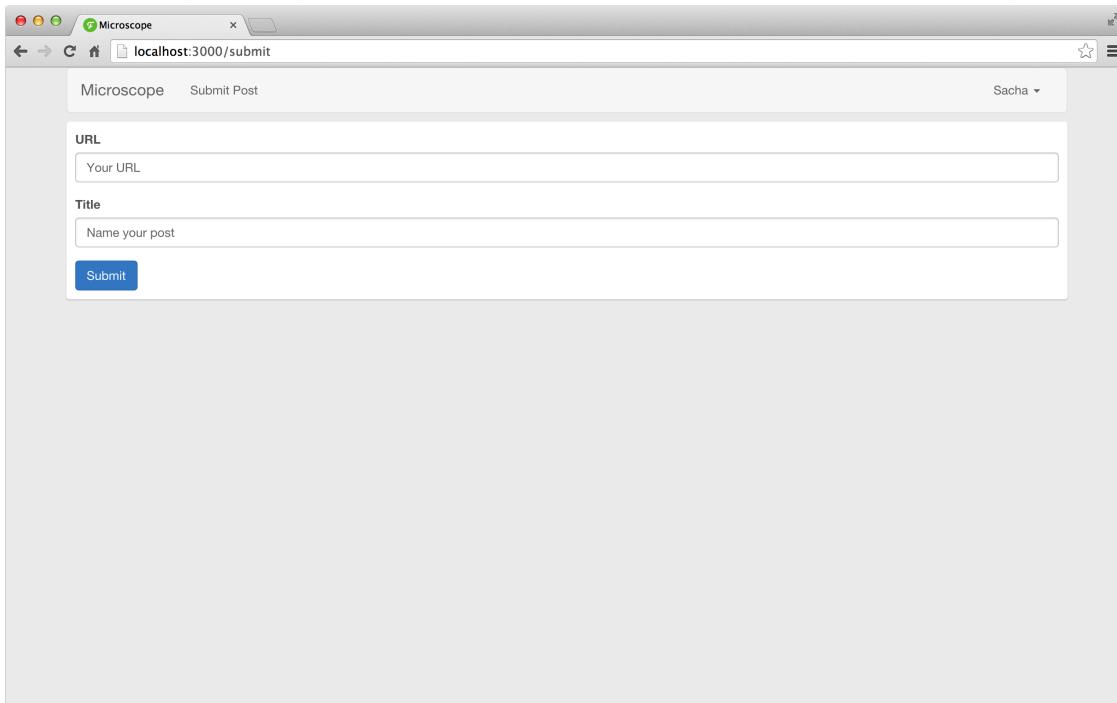
Con la ruta definida, ahora podemos añadir un enlace a la cabecera de nuestra página:

```
<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'postsList'}}>Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav">
        <li><a href="{{pathFor 'postSubmit'}}>Submit Post</a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>
```

Configurar una ruta implica que si un usuario navega a `/submit`, Meteor mostrará la plantilla `postSubmit`. Así que vamos a escribir esa plantilla:

```
<template name="postSubmit">
<form class="main form page">
  <div class="form-group">
    <label class="control-label" for="url">URL</label>
    <div class="controls">
      <input name="url" id="url" type="text" value="" placeholder="Your URL" class="form-control"/>
    </div>
  </div>
  <div class="form-group">
    <label class="control-label" for="title">Title</label>
    <div class="controls">
      <input name="title" id="title" type="text" value="" placeholder="Name your post" class="form-control"/>
    </div>
  </div>
  <input type="submit" value="Submit" class="btn btn-primary"/>
</form>
</template>
```

Aquí hay un montón de markup, pero es solo porque usamos el CSS de Twitter Bootstrap. Aunque sólo son esenciales los elementos del formulario, el marcado adicional ayudará a que nuestra aplicación se vea un poco mejor. Ahora debería tener un aspecto similar a este:



Es un simple formulario. No tenemos que preocuparnos de programar una acción para él, porque interceptaremos su evento `submit` y actualizaremos los datos vía JavaScript. (No tiene sentido proporcionar un fallback no-JS si tenemos

en cuenta que Meteor no funciona con JavaScript desactivado).

Vamos a enlazar un controlador de eventos al evento `submit` del formulario. Es mejor usar el evento `submit` (en lugar de un click en un botón), ya que cubrirá todas las posibles formas de envío (como por ejemplo pulsar intro).

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    post._id = Posts.insert(post);
    Router.go('postPage', post);
  }
});
```

client/templates/posts/post_submit.js

Nueva página de envío y enlace a ella desde la cabecera.

[Ver en GitHub](#)

[Lanzar instancia](#)

Esta función utiliza **jQuery** para analizar los valores de los distintos campos del formulario y llenar un objeto `post` con los resultados. Tenemos que asegurarnos de usar `preventDefault` para que el navegador no intente enviar el formulario si volvemos atrás o adelante después.

Al final, podemos dirigirnos a la página de nuestro nuevo post. La función `insert()` devuelve el identificador `_id` del objeto que se ha insertado en la base de datos, que podemos pasar a la función `go()` del router para que nos lleve a la página correcta.

El resultado es que el usuario pulsa en `submit`, se crea un nuevo post, y vamos inmediatamente a la página de discusión de ese nuevo post.

Tal como está ahora, cualquiera que visite la web puede crear posts. Para evitarlo, debemos hacer que los usuarios inicien sesión. Podríamos ocultar el nuevo formulario, pero aún así, se podría seguir haciendo desde la consola.

Afortunadamente, Meteor gestiona la seguridad de las colecciones de la forma adecuada, lo que ocurre es que, por defecto, esta característica viene desactivada. Esto es así para permitirnos empezar con facilidad a construir la aplicación, dejando las cosas aburridas para más tarde.

Es el momento de eliminar el paquete `insecure`:

```
meteor remove insecure
```

Terminal

Después de hacerlo, nos damos cuenta de que el formulario de posts ya no funciona. Esto es así, porque sin el paquete `insecure`, *no se permiten inserciones* en la colección de posts desde el lado del cliente.

Necesitamos escribir reglas explícitas para decirle a Meteor qué usuarios pueden insertar posts o hacer que las inserciones se hagan en el lado del servidor.

Para que nuestro formulario funcione de nuevo, vamos a ver cómo permitir posts del lado del cliente. Como veremos, al final usaremos una técnica diferente, pero por ahora, lo pondremos todo a funcionar de nuevo, de una forma sencilla: en `collections/posts.js`:

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  insert: function(userId, doc) {
    // only allow posting if you are logged in
    return !!userId;
  }
});
```

lib/collections/posts.js

Eliminado el paquete `insecure` y permitido añadir posts

...

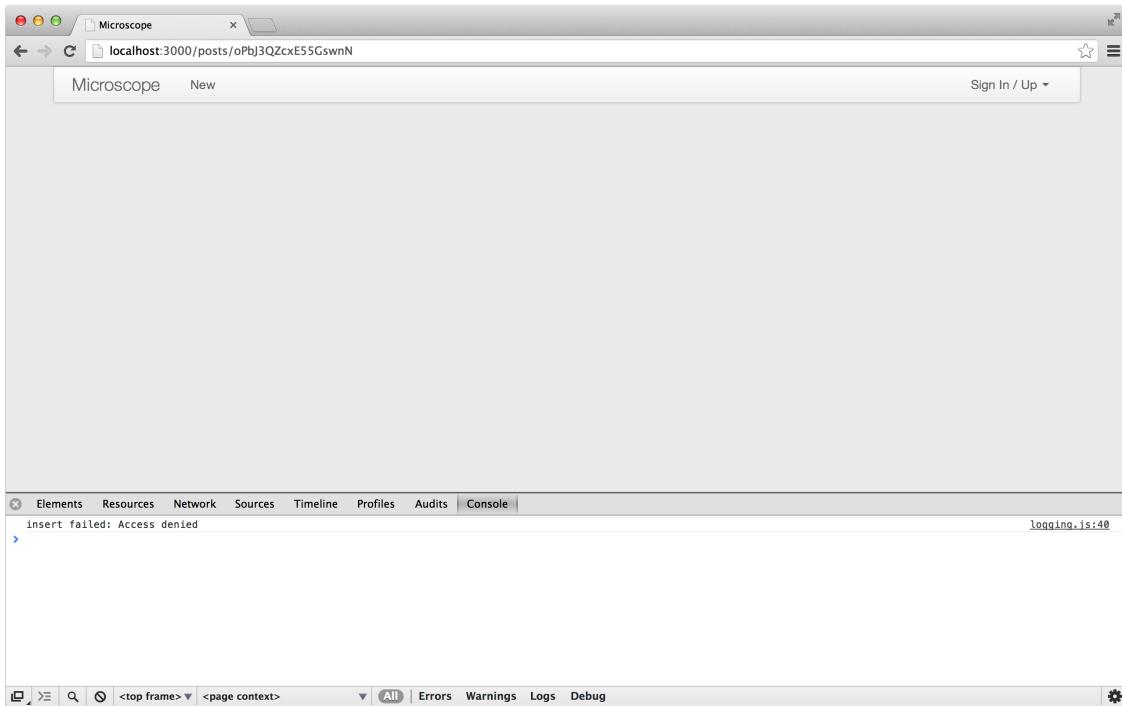
[Ver en GitHub](#)

[Lanzar instancia](#)

Llamamos a `Posts.allow`, que le dice a Meteor que “se trata de un conjunto de circunstancias en las que a los clientes se les permite hacer cosas en la colección de `Posts`”. En este caso, estamos diciendo: “a los clientes se les permite insertar posts siempre y cuando tengan un `userId`”.

El `userId` que realiza la modificación se pasa a las funciones `allow` y `deny` (o devuelve `null` si no hay ningún usuario conectado). Como las cuentas de usuario forman parte del núcleo de Meteor, podemos confiar en que el `userId` siempre será el correcto.

Nos las hemos arreglado para asegurarnos de que un usuario tiene que estar registrado para crear un mensaje. Salimos de la sesión e intentamos crear un post para ver lo que sale por la consola del navegador:



Sin embargo, todavía tenemos que tratar con unas cuantas cosas:

- Los usuarios que no han iniciado sesión aún pueden ver el formulario.
- El post no está vinculado al usuario de ninguna forma.
- Se pueden crear múltiples posts que apunten a la misma URL.

Vamos a corregir estos problemas.

Vamos a empezar por evitar que los usuarios no registrados puedan ver el formulario de envío de posts. Lo haremos a nivel de router, definiendo una acción (*hook*) del router.

Una acción intercepta el proceso de enrutamiento y, potencialmente, cambia la acción que lleva a cabo el router. Puedes pensar en él como en un guardia de seguridad que verifica tus credenciales antes de dejarte entrar.

Lo que tenemos que hacer es comprobar si el usuario está conectado. Si no lo está, mostramos la plantilla `accessDenied` en lugar de la plantilla `postSubmit` (en este momento le diremos al router que no haga nada más). Así que vamos a modificar `router.js`:

```

Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

var requireLogin = function() {
  if (!Meteor.user()) {
    this.render('accessDenied');
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});

```

lib/router.js

Además, tenemos que crear una plantilla para la página de error:

```

<template name="accessDenied">
  <div class="access-denied page jumbotron">
    <h2>Access Denied</h2>
    <p>You can't get here! Please log in.</p>
  </div>
</template>

```

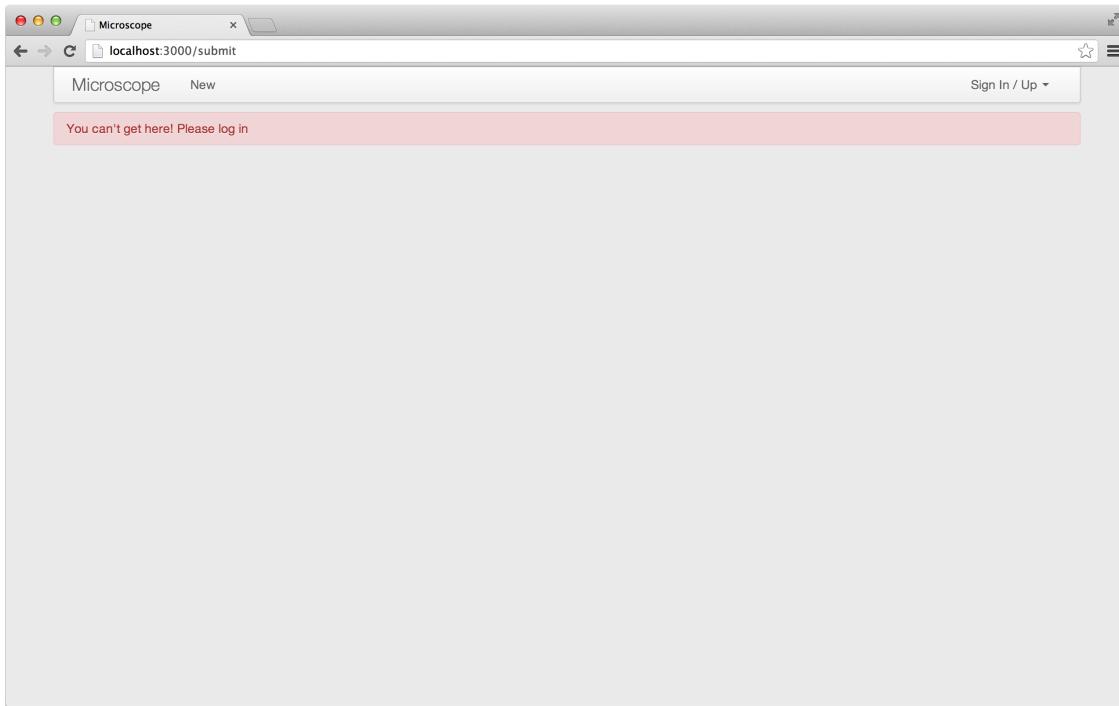
client/templates/includes/access_denied.html

Acceso denegado al envío de posts a usuarios no registrados.

[Ver en GitHub](#)

[Lanzar instancia](#)

Si ahora nos dirigimos a <http://localhost:3000/submit> sin estar registrados, veremos el mensaje de error:



Lo bueno de las acciones del router es que son *reactivas*. Esto significa que no necesitamos pensar en funciones de retorno cuando el usuario se autentica: cuando el estado de autenticación del usuario cambia, la plantilla del Router cambia instantáneamente de `accessDenied` a `postSubmit` sin tener que escribir explícitamente código para manejarlo (y además, esto funciona incluso en las otras pestañas del navegador).

Iniciemos sesión, y vayamos a la página para crear un nuevo post. Ahora actualizar la página en el navegador. Veremos que, por un instante, se ve la plantilla `accessDenied` antes de que aparezca el formulario. Esto es porque Meteor empieza a mostrar las plantillas tan pronto como sea posible, antes de haber hablado con el servidor y comprobado si el usuario existe.

Para evitar este problema (que es uno de los más comunes que nos podemos encontrar cuando tratamos de lidiar con la latencia entre el cliente y el servidor), solo mostraremos una pantalla de espera durante un instante en el que esperamos para ver si el usuario tiene acceso o no.

Después de todo en este momento no sabemos si el usuario tiene acceso y no podemos mostrar ninguna de las plantillas, ya sea la de `accessDenied` o la de `postSubmit` hasta que lo sepamos.

Así que vamos a modificar nuestra acción para añadir la plantilla de espera mientras `Meteor.loggingIn()` sea verdadero en:

```
//...

var requireLogin = function() {
  if (! Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

lib/router.js

Mostrar la pantalla de carga mientras esperamos al login.

[Ver en GitHub](#)

[Lanzar instancia](#)

La forma fácil de evitar que los usuarios lleguen al formulario es esconder el enlace. Podemos hacerlo fácilmente desde `header.html`:

```
//...

<ul class="nav navbar-nav">
  {{#if currentUser}}<li><a href="{{pathFor 'postSubmit'}}>Submit Post</a></li>{{/if}}
</ul>

//...
```

client/templates/includes/header.html

No mostrar el enlace a la página de envío si el usuario

n...

[Ver en GitHub](#)

[Lanzar instancia](#)

El paquete `accounts` nos ofrece el ayudante `currentUser` que es el equivalente a `Meteor.user()` en Spacebars. Puesto que es reactivo, el enlace aparecerá o desaparecerá según el estado del usuario.

Nos las hemos arreglado para asegurar el acceso a la página de entrada de posts, y no permitir crear posts a usuarios no registrados incluso si intentan hacerlo desde la consola. Sin embargo, todavía quedan cosas que debemos mejorar:

- Añadir el timestamp de los posts.
- Asegurarse de que no hay URLs duplicadas.
- Añadir detalles sobre el autor del post (ID, nombre de usuario, etc.)

Podríamos pensar en hacer todo esto en nuestro controlador `submit`. Pero, haciéndolo de esta forma, nos encontraríamos con un montón de problemas.

- Para el timestamp, tendríamos que confiar en la hora de la máquina del usuario.
- Los clientes no conocerán `todas` las URL publicadas. Solo conocen los posts que pueden ver en ese momento (veremos porqué), así que no podemos asegurar desde el lado del cliente que las URLs sean únicas.
- Por último, aunque *podríamos* añadir la información de usuario en el lado del cliente, estaríamos abriendo nuestra aplicación a posibles ataques de usuarios usando la consola del navegador.

Por todas estas razones, es mejor mantener nuestros controladores de eventos simples y, si queremos hacer más inserciones o actualizaciones en las colecciones, debemos usar **métodos**.

Un método en Meteor es una función del lado del servidor que se *llama* desde el lado del cliente. Ya estamos familiarizados con ellos – de hecho, entre bastidores, la inserción, la actualización y el borrado de datos de la colección, son métodos. Vamos a ver cómo crear el nuestro.

Volvamos a `post_submit.js`. En lugar de insertar directamente en la colección `Posts`, vamos a llamar a un método llamado `postInsert`:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

La función `Meteor.call` llama a un método nombrado por su primer argumento. Se pueden proporcionar argumentos a la llamada (en este caso, pasamos el objeto `post` que hemos construido del formulario), y, finalmente, habilitamos un callback, que se ejecutará cuando el método del lado del servidor finalice.

Las funciones de retorno de los métodos Meteor siempre tienen dos argumentos, `error` y `result`. Si por cualquier

razón el argumento `error` existe, avisaremos al usuario (usando `return` para finalizar la función). Si todo ha funcionado bien, redirigiremos al usuario a la página de discusión del post recién creado.

Aprovecharemos esta oportunidad para añadir algo de seguridad a nuestros métodos usando el paquete `audit-argument-checks`.

Este paquete nos permite realizar comprobaciones sobre un objeto JavaScript usando patrones predefinidos. En nuestro caso, lo usaremos para comprobar que el usuario que está invocando el método está correctamente autenticado (asegurándonos que `Meteor.userId()` es de tipo `String`), y que el objeto `postAttributes` pasado como argumento al método contiene las cadenas `title` y `url`, para no terminar insertando cualquier dato extraño en nuestra base de datos.

Vamos a definir el método `postInsert` en nuestro fichero `collections/posts.js`. Eliminaremos el bloque `allow()` del fichero `posts.js` porque usando métodos, Meteor no lo evalúa.

Extenderemos (`extend`) el objeto `postAttributes` con tres propiedades más: el identificador del usuario `_id` y el `username`, además de la fecha y hora `submitted`, antes de insertarlos en nuestra base de datos y devolver el `_id` al cliente (en otras palabras, a la función original que llamó a este método) como un objeto JavaScript.

```
Posts = new Mongo.Collection('posts');

Meteor.methods({
  postInsert: function(postAttributes) {
    check(Meteor.userId(), String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});
```

lib/collections/posts.js

Fíjate que el método `_.extend()` forma parte de la librería **Underscore**, que simplemente nos permite “extender” un objeto con propiedades de otro.

Usando un método para enviar un post.

[Ver en GitHub](#)

[Lanzar instancia](#)

Los métodos Meteor son ejecutados en el servidor, por lo que Meteor supone que son de confianza. Por tanto, los métodos Meteor obvian las llamadas a `allow` y `deny`.

Si quieras ejecutar algún código antes de cada operación de `insert`, `update`, o `remove` *incluso en el lado servidor*, te sugerimos echar un vistazo al paquete **collection-hooks**.

Vamos a hacer una comprobación más antes de dar por bueno nuestro método. Si ya tenemos un post con la misma URL, no vamos a permitir que se añada una segunda vez, por el contrario, redirijamos al usuario al post ya existente.

```
Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});
```

lib/collections/posts.js

Buscamos en nuestra base de datos las URLs duplicadas. Si se encuentra alguna, devolvemos (`return`) el `_id` del post junto con una marca `postExists: true` para informar al cliente sobre esta situación especial.

Y como estamos lanzando una llamada `return`, el método se detiene en este punto sin llegar a ejecutar la sentencia `insert`, evitándonos elegantemente cualquier duplicidad.

Sólo falta usar `postExists` en nuestro ayudante de eventos en el lado del cliente para mostrarnos un mensaje de aviso:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

Forzando la unicidad de las URLs.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora que tenemos una fecha de envío en todos nuestros posts, tiene sentido asegurarnos que se están ordenando usando este atributo. Para ello usaremos el operador `sort` de Mongo que espera un objeto que consta de las claves de ordenación, y un signo que indica si son ascendentes o descendentes:

```
Template.postsList.helpers({
  posts: function() {
    return Posts.find({}, {sort: {submitted: -1}});
  }
});
```

client/templates/posts/posts_list.js

Posts ordenados por fecha de envío.

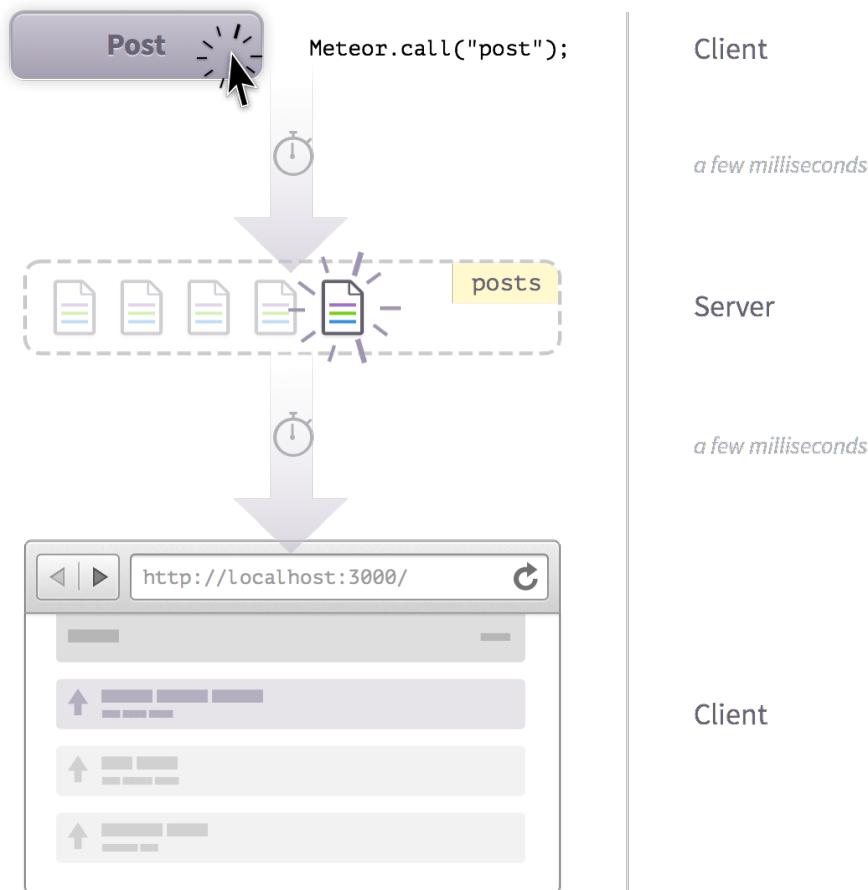
[Ver en GitHub](#)

[Lanzar instancia](#)

Ha costado, pero ¡Finalmente tenemos una interfaz en la que los usuarios introducen posts de forma segura en nuestra aplicación!

Sin embargo, cualquier aplicación que permita a los usuarios crear contenido también debe permitir editarla o borrarla. Eso es de lo que hablaremos en el siguiente capítulo.

En el último capítulo, se introdujo un nuevo concepto en el mundo Meteor: **los métodos**.

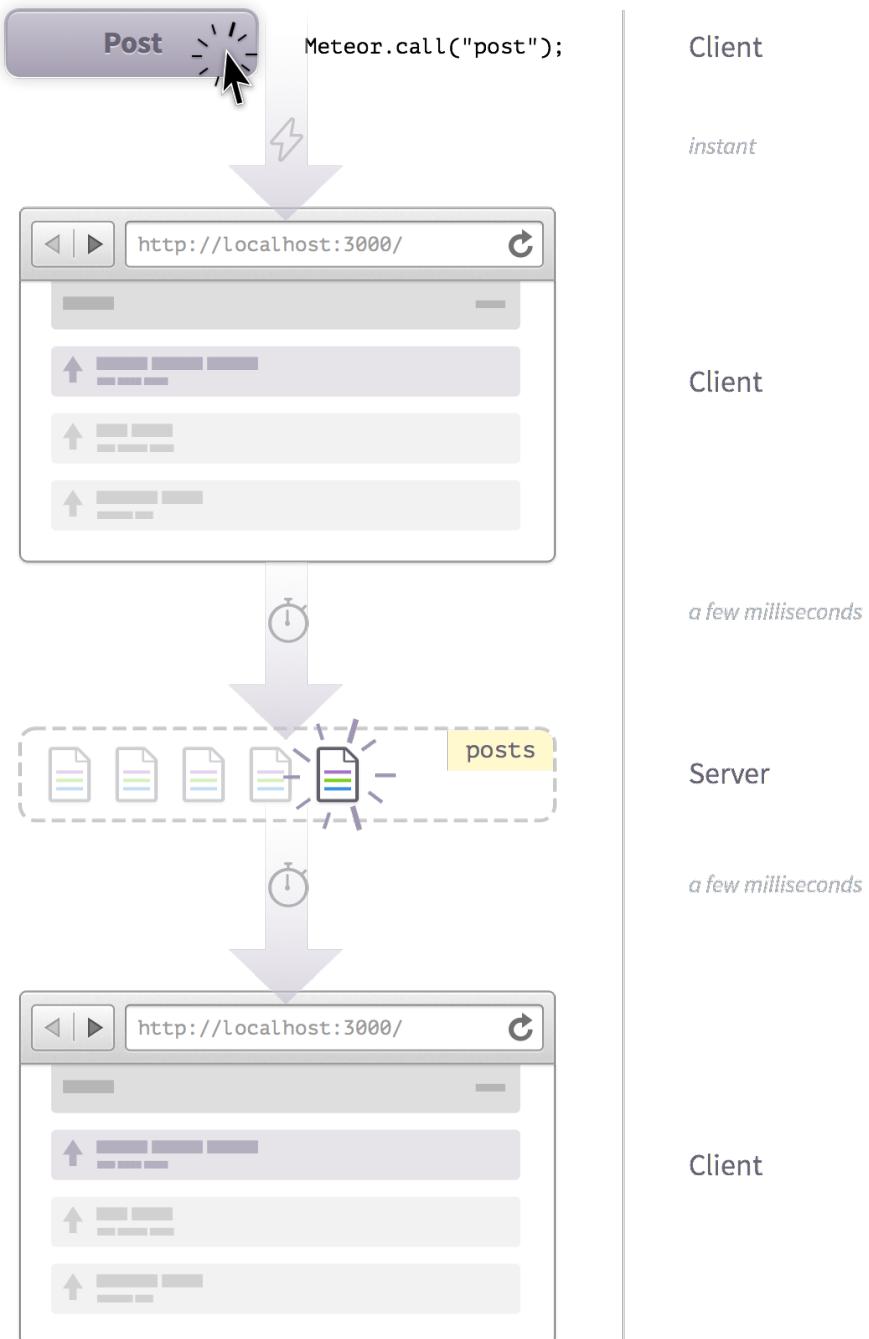


Un método es una forma de ejecutar una serie de comandos en el servidor de una manera estructurada. En nuestro ejemplo, hemos utilizado un método porque queríamos asegurarnos que los nuevos posts se etiqueten con el nombre e identificador de su autor y con la hora actual del servidor.

Sin embargo, tendríamos un problema si Meteor ejecutara los métodos de una forma más básica. Considera la siguiente secuencia de eventos (nota: las marcas de tiempo son valores aleatorios escogidos con fines ilustrativos):

- +0ms: El usuario hace clic en un botón de envío y el navegador lanza una llamada al método.
- +200ms: El servidor realiza cambios en la base de datos Mongo.
- +500ms: El cliente recibe estos cambios y actualiza la interfaz de usuario para reflejarlos.

Si fuera así, habría un breve retraso entre la acción y la visualización de los resultados (que se notaría más o menos dependiendo de lo cerca que estuviésemos del servidor). Esto no se puede permitir en una aplicación web moderna!



Para evitar este problema, Meteor introduce un concepto llamado **Compensación de la Latencia**. Cuando definimos nuestro método `Post`, lo colocamos dentro de un archivo en el directorio `collections/`. Esto implica que estará disponible tanto para el servidor como para el cliente - ¡y se ejecutará al mismo tiempo en ambos contextos!

Cuando se hace una llamada a un método, el cliente no solo envía la llamada al servidor, sino que también *simula* la acción en su propia colección. De esta forma, el flujo de trabajo sería:

- +0ms: El usuario hace clic en un botón de envío y el navegador lanza una llamada al método.
- +0ms: El cliente simula la acción en su propia colección y cambia la interfaz de usuario para reflejar los cambios.
- +200ms: El servidor realiza cambios en la base de datos Mongo..
- +500ms: El cliente recibe esos cambios deshaciendo los que ha simulado y reemplazándolos con los del servidor (que, generalmente son los mismos) y la interfaz de usuario los refleja.

Esto se traduce en que el usuario ve los cambios instantáneamente. Cuando llega la respuesta del servidor unos momentos más tarde, puede haber o no cambios notables. Por tanto, una cosa que tenemos que aprender es que debemos tratar de asegurar de que, en la medida de lo posible, los documentos simulados sean muy parecidos a los reales.

Haciendo un pequeño cambio en el método `post` veremos todo esto en acción. Para ello, usaremos la útil función `Meteor._sleepForMs()` para retrasar la llamada de la función cinco segundos, pero (importante) *solo en el servidor*.

Usaremos `isServer` para preguntar a Meteor si el método se está invocando en el cliente (como un “stub”) o en el servidor. Un **stub** es la simulación del método que ejecuta Meteor en el cliente, mientras el método “real” se está ejecutando en el servidor.

Así que vamos a preguntar a Meteor si se está ejecutando el código en el servidor. Si es así, retrasaremos el proceso cinco segundos y añadiremos la cadena `(server)` al final del título de nuestro post. Si no, añadiremos `(client)`:

```
Posts = new Mongo.Collection('posts');

Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    if (Meteor.isServer) {
      postAttributes.title += "(server)";
      // wait for 5 seconds
      Meteor._sleepForMs(5000);
    } else {
      postAttributes.title += "(client)";
    }

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});
```

collections/posts.js

Si nos detuviéramos aquí, la demostración no sería muy concluyente. En este punto, solo parece que el envío del formulario está siendo pausado por cinco segundos antes de redirigirte a la lista principal de posts, y nada más.

Para entender porqué, volvamos a ver el ayudante de envío:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

Hemos colocado nuestra llamada de enrutamiento `Router.go()` dentro de la función de retorno de la llamada. Lo que significa que el formulario está esperando a que la llamada al método concluya antes de redirigir.

Normalmente, esto es el cauce normal de la acción. Después de todo, no podemos redirigir al usuario antes de saber si el post es válido o no, ya que sería extremadamente confuso redirigir, y, al momento, redirigir de nuevo hacia el formulario de envío original para corregir los datos, todo en cuestión de segundos.

Pero para el propósito de este ejemplo, queremos ver el resultado de nuestras acciones inmediatamente. Por lo que cambiaremos la llamada a Router para redirigir a la ruta `postsList` (no podemos redirigir al post porque no sabemos su identificador fuera del método). Lo sacaremos fuera de la llamada, y veremos que pasa:

```

Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');
    });

    Router.go('postsList');

  }
});

```

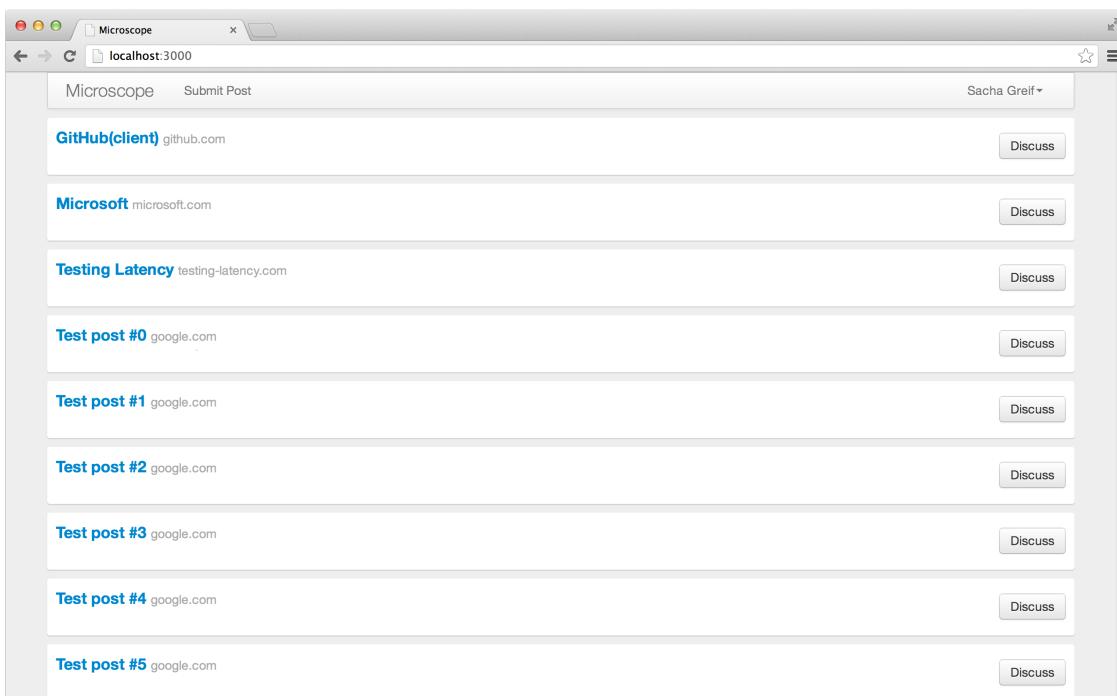
client/templates/posts/post_submit.js

Demostrar el orden en el que aparecen los posts usando
un...

[Ver en GitHub](#)

[Lanzar instancia](#)

Si ahora creamos un nuevo post, vemos claramente la compensación de la latencia. En primer lugar, se inserta el post con el título `(cliente)` (El primer post de la lista, apuntando hacia GitHub):



Cinco segundos más tarde, se reemplaza con el documento real insertado por el servidor:

The screenshot shows a web application window titled "Microscope" with the URL "localhost:3000". The main content area displays a list of documents from a collection named "posts". Each document entry includes the title, URL, and a "Discuss" button. The titles of the documents are: "GitHub(server)" (github.com), "Microsoft" (microsoft.com), "Testing Latency" (testing-latency.com), and "Test post #0" through "Test post #5" (all google.com). The "Test post" entries are clearly labeled as test cases.

Después de todo esto, puedes pensar que todo esto de los métodos, es complicado, pero en realidad pueden ser bastante simples. De hecho, ya hemos visto tres métodos muy sencillos: los métodos `insert`, `update` y `remove` de una colección.

Cuando se define una colección `'posts'`, se están definiendo implícitamente tres métodos: `posts.insert`, `posts.update` y `posts.delete`. En otras palabras, cuando se llama `Posts.insert()` desde el cliente, se llama a un método compensado en latencia que hace dos cosas:

1. Comprueba si podemos hacer el cambio llamando a los callbacks `allow` y `deny` (sin embargo, esto no tiene porque ser así en la simulación).
2. Efectúa, de verdad, el cambio en el almacén de datos subyacente.

Si has estado atento, te habrás dado cuenta de que el método `post` llama a su vez a otro método (`posts.insert`) cada vez que insertamos un nuevo post. ¿Y, cómo es que esto funciona?

Mientras se está ejecutando la simulación (la versión del método en el cliente), ejecutamos un `insert` simulado (lo insertamos en nuestra colección cliente), pero realmente *no* llamamos al `insert` del servidor, porque esperamos que sea la versión de `post` que hay en el servidor la que lo haga.

En consecuencia, cuando el método `post` del servidor llama a `insert` no tiene necesidad de preocuparse por la simulación, y la inserción se realiza sin problemas.

Como anteriormente, no olvides deshacer los cambios antes de avanzar al siguiente capítulo.

Ahora que ya podemos crear posts, el siguiente paso es poder editarlos y borrarlos. Como el código de la IU ha quedado bastante simple, este parece un buen momento para hablar de cómo se gestionan los permisos de usuario con Meteor.

Primero vamos a configurar nuestro router. Añadiremos una ruta para acceder a la página de edición y estableceremos su contexto de datos:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/posts/:_id/edit', {
  name: 'postEdit',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

var requireLogin = function() {
  if (!Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});

lib/router.js
```

Ahora ya nos podemos centrar en la plantilla. Nuestra plantilla `postEdit` tendrá una forma bastante estándar:

```

<template name="postEdit">
  <form class="main form page">
    <div class="form-group">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="{{url}}" placeholder="Your URL" class="form-control"/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="{{title}}" placeholder="Name your post" class="form-control"/>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary submit"/>
    <hr/>
    <a class="btn btn-danger delete" href="#">Delete post</a>
  </form>
</template>

```

client/templates/posts/post_edit.html

Y aquí tenemos el fichero `post_edit.js` que la acompaña:

```

Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    }

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        alert(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },

  'click .delete': function(e) {
    e.preventDefault();

    if (confirm("Delete this post?")) {
      var currentPostId = this._id;
      Posts.remove(currentPostId);
      Router.go('postsList');
    }
  }
});

```

client/templates/posts/post_edit.js

Como podemos ver, la mayoría de cosas ya nos son familiares.

Tenemos dos callbacks de eventos: uno para enviar el formulario `submit`, y `click .delete` para el evento click del

enlace delete.

El callback `delete` es muy simple: elimina el evento predeterminado y después pide confirmación. Si confirmamos, obtenemos el ID del post actual desde el contexto de datos de la plantilla, lo borramos y redirigimos al usuario a la página de inicio.

El callback de actualización es un poco más largo, pero no mucho más complicado. Después de suprimir el evento predeterminado y conseguir el post actual, obtiene los nuevos valores del formulario y los almacena en el objeto `postProperties`.

A continuación, pasa este objeto al método Meteor `Collection.update()` usando el operador `$set` (que reemplaza un conjunto de atributos dejando los demás intactos), y usa un callback para mostrar un error si falla la actualización o envía al usuario a la página del post si la actualización se realiza correctamente.

Deberíamos añadir enlaces a la página de edición de nuestros posts para que los usuarios puedan llegar a ella:

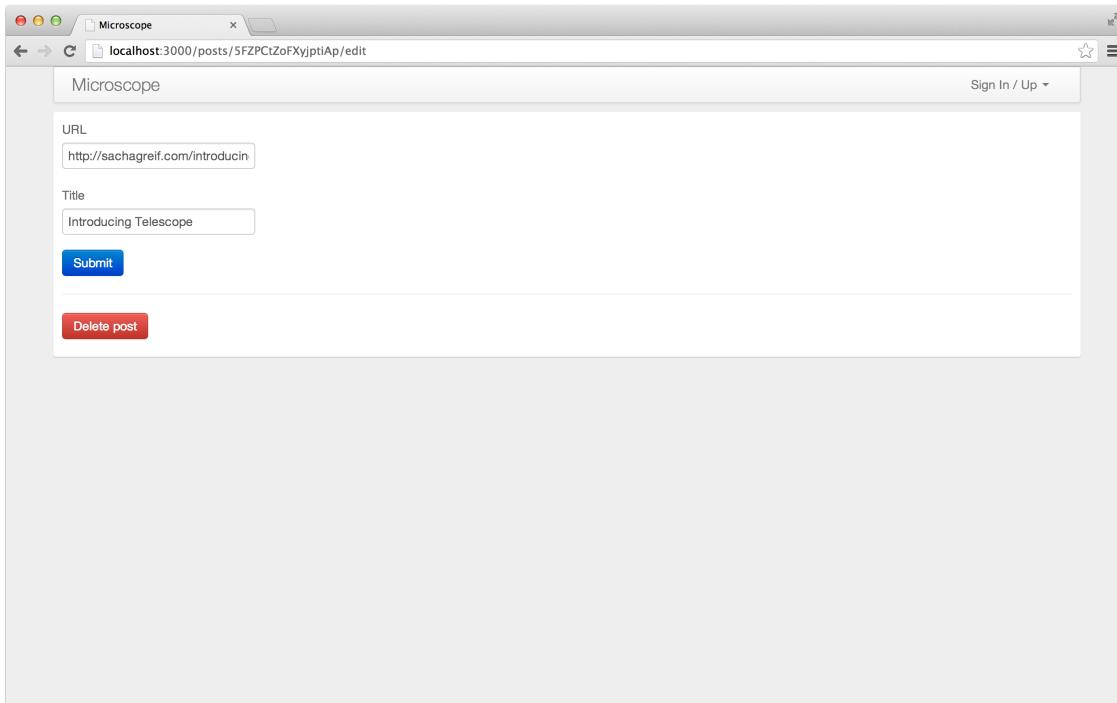
```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}>{{title}}</a><span>{{domain}}</span></h3>
      <p>
        submitted by {{author}}
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}>Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}>{{discuss btn btn-default}} Discuss</a>
  </div>
</template>
```

client/templates/posts/post_item.html

Por supuesto, no queremos que se muestre el enlace para editar un post que no haya sido creado por ese usuario. Aquí es donde entra el ayudante `ownPost`:

```
Template.postItem.helpers({
  ownPost: function() {
    return this.userId === Meteor.userId();
  },
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  }
});
```

client/templates/posts/post_item.js



Añadido el formulario de edición.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ya tenemos nuestro formulario de envío de edición, pero en realidad, todavía no se puede editar nada. ¿Qué es lo que está pasando?

Al haber eliminado el paquete `insecure`, se nos deniegan todas las peticiones de modificación desde el cliente.

Para solucionarlo, estableceremos algunas reglas de permisos. Primero, creamos un nuevo archivo `permissions.js` dentro de `lib`. Esto nos asegura que nuestra lógica de permisos se cargará lo primero (y estará disponible en los dos entornos):

```
// check that the userId specified owns the documents
ownsDocument = function(userId, doc) {
  return doc && doc.userId === userId;
}
```

lib/permissions.js

En el capítulo [Creando posts](#) nos libramos de tener que usar el método `allow()` porque estábamos insertando nuevos posts a través de un método de servidor.

Pero ahora que estamos editando y borrando posts desde el cliente, vamos a necesitar volver a `collections/posts.js` y a añadir un bloque `allow()` :

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  update: function(userId, post) { return ownsDocument(userId, post); },
  remove: function(userId, post) { return ownsDocument(userId, post); },
});

//...
```

lib/collections/posts.js

Añadidos permisos básicos para comprobar el dueño del post.

[Ver en GitHub](#)

[Lanzar instancia](#)

Solo porque podamos editar nuestros propios posts, no significa debamos ser capaces de editar todas las propiedades. Por ejemplo, no queremos que los usuarios puedan crear un post y luego asignárselo a otro usuario.

Utilizaremos el callback `deny()` para asegurarnos de que los usuarios solo puedan editar los atributos especificados:

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  update: ownsDocument,
  remove: ownsDocument
});

Posts.deny({
  update: function(userId, post, fieldNames) {
    // may only edit the following two fields:
    return (_.without(fieldNames, 'url', 'title').length > 0);
  }
});
```

lib/collections/posts.js

Permitir cambios sólo en ciertos campos.

[Ver en GitHub](#)

[Lanzar instancia](#)

Estamos cogiendo el array `fieldNames` que contiene la lista de los campos que quieren modificar, y usamos el método

`without()` de **Underscore** para devolver un sub-array que contiene los campos que no son `url` o `title`.

Si todo va bien, el array debe estar vacío y su longitud debe ser 0. Pero si alguien está tratando de enredar, la longitud del array será mayor que 0, y devolveremos `true` (denegando así la actualización).

Te habrás dado cuenta de que, en el código de edición del post, no comprobamos si hay enlaces duplicados. Esto significa que un usuario podría enviar un enlace y después editarlo y cambiar su URL para saltarse la comprobación. La solución a este problema podría ser utilizar un método Meteor para tratar este formulario de edición, pero dejaremos esto como un ejercicio para el lector.

Para crear un `post`, estamos utilizando un método `postInsert` en el servidor, mientras que para editarlos y eliminarlos, llamamos a `update` y `remove` directamente desde el cliente, limitando el acceso a través de `allow` y `deny`.

¿Cuándo es más adecuado usar uno u otro?

Cuando las cosas son relativamente sencillas se pueden expresar las reglas a través de `allow` y `deny`, por lo general es más fácil de hacer las cosas directamente desde el cliente.

Sin embargo, tan pronto como empecemos a hacer cosas que deberían estar fuera del control del usuario (por ejemplo, el timestamp de un nuevo post o asignarlo al usuario correcto), será mejor que usemos métodos.

Las llamadas a métodos también son apropiadas en algunos otros casos:

- Cuando necesitamos conocer o devolver valores a través de un callback en lugar de esperar a que Meteor propague la sincronización.
- Para consultas pesadas a la base de datos.
- Para resumir o agregar datos (por ejemplo, contadores, promedios, sumas).

Para conocer más a fondo este tema [echa un vistazo a nuestro blog](#).

El sistema de seguridad de Meteor nos permite controlar los cambios en la base de datos sin tener que definir los métodos necesarios para hacerlo.

Nosotros hemos tenido que definir un método `post` específico porque necesitamos hacer tareas adicionales, tales como decorar el post con propiedades adicionales y tomar medidas si la URL del post ya existe.

Por otro lado, tampoco hemos tenido que crear métodos para actualizar y eliminar posts. Solo necesitábamos comprobar si el usuario tenía permiso para hacer la acción, y ha sido muy fácil usando los callbacks `allow` y `deny`.

Usar estos callbacks nos ha permitido ser más declarativos con las modificaciones en la base de datos, definiendo qué tipo de cambios se pueden hacer. El hecho de que estén integrados en el sistema de cuentas es, además, una ventaja añadida.

Podemos definir todos los callbacks `allow` que queramos. Solo es necesario que, *al menos uno* de ellos devuelva `true`, para el cambio actual. De esta forma, cuando se llama a `Posts.insert` desde un navegador (no importa si es desde el código cliente de nuestra aplicación o desde la consola), el servidor llamará a todos los `insert`-permitidos que pueda hasta que encuentre uno que devuelva `true`. Si no encuentra ninguno, no permite la inserción, y se devuelve al cliente un error `403`.

Del mismo modo, podemos definir uno o varios callbacks `deny`. Si *cualquiera* de ellos devuelve `true`, el cambio se cancela y se devuelve un `403`. La lógica de todo esto implica que, para que un `insert` tenga éxito, se ejecutarán uno o varios callbacks `allow` así como *todos los* `deny`.



Callback Execution Order	<code>Posts.deny({ update: function(){...} })</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>
	<code>Posts.deny({ update: function(){...} })</code>	<code>false</code>	<code>false</code>	<code>n/e</code>	<code>false</code>
	<code>Posts.allow({ update: function(){...} })</code>	<code>true</code>	<code>false</code>	<code>n/e</code>	<code>false</code>
	<code>Posts.allow({ update: function(){...} })</code>	<code>n/e</code>	<code>true</code>	<code>n/e</code>	<code>false</code>
Result		✓	✓	✗	✗

En otras palabras, Meteor recorre la lista de callbacks empezando por los `deny`, y luego ejecuta todos los `allow` hasta que uno devuelve `true`.

Un ejemplo práctico de este patrón podría ser, por ejemplo, tener dos callbacks `allow()`, uno comprueba si un post pertenece al usuario actual, y otro si el usuario actual es un administrador. Si es un administrador, se asegura que el

usuario será capaz de actualizar cualquier post, ya que al menos uno de los callbacks devolverá true.

Recuerda que los métodos que provocan cambios en la base de datos (como `.update()`) compensan la latencia, igual que cualquier otro método. Así, por ejemplo, si desde la consola del navegador, intentas eliminar un post que no te pertenece, verás que, por un momento, el post desaparece, porque la colección local pierde el documento, pero luego vuelve a aparecer cuando el servidor nos dice que no, que el documento no ha sido eliminado.

Por supuesto, este comportamiento no es un problema cuando se activa desde la consola (después de todo, si los usuarios juegan con los datos desde la consola, no es problema nuestro lo que ocurra en *sus* navegadores). Sin embargo, necesitas asegurarte de que esto no sucede desde la interfaz de usuario. Por ejemplo, necesitas tomarte la molestia de asegurar que no muestras a los usuarios botones para eliminar documentos que no están autorizados a borrar.

Afortunadamente, no suele requerir demasiado código extra poner el código que define los permisos, compartido entre el cliente y el servidor (por ejemplo, podrías escribir una función `canDeletePost(user, post)` y ponerla en el directorio `/lib`).

Recuerda que el sistema de permisos solo se aplica a cambios en la base de datos iniciados desde el cliente. En el servidor, Meteor asume que se permiten *todas* las operaciones.

Esto significa que si escribes un método `DeletePost` en el lado del servidor, que puede llamarse desde el cliente, todo el mundo podrá borrar cualquier post. Así que, es probable que no quieras hacer esto, a menos que compruebes los permisos de usuario dentro de ese método.

Resulta poco elegante usar un `alert()` para advertir al usuario de que algo ha pasado. Hay que hacerlo mejor.

Vamos a construir un mecanismo de presentación de errores más versátil y que va a hacer mejor el trabajo de informar al usuario sobre lo que está pasando sin tener que romper su flujo de trabajo.

Vamos a implementar un simple sistema que muestre los nuevos errores en la parte de arriba a la derecha de la ventana, de forma similar a la popular aplicación **Growl** de MacOS.

Para empezar, crearemos una colección para almacenar nuestros errores. Puesto que los errores solo son relevantes en la sesión actual y no necesitan ser persistentes, haremos algo nuevo, vamos a crear una *colección local*. Es decir, la colección `Errors` solo existirá *en el navegador*, y no se sincronizará con el servidor.

Para conseguirlo, creamos la colección dentro del directorio `client` (para hacer que sólo esté disponible en la parte cliente), con el nombre de la colección establecido a `null` (ya que los datos nunca se almacenarán en la base de datos del servidor):

```
// Local (client-only) collection
Errors = new Mongo.Collection(null);
```

client/helpers/errors.js

Ahora que hemos creado la colección, podemos agregar una función `throwError`, que usaremos para añadir errores. Al tratarse de una colección local, no tenemos que preocuparnos por definir `allow` o `deny` u otros mecanismos de seguridad, ya que esta colección es “local” para el usuario actual.

```
throwError = function(message) {
  Errors.insert({message: message});
};
```

client/helpers/errors.js

La ventaja de utilizar una colección local para almacenar los errores es que, como todas las colecciones, es reactiva – lo que significa que podemos mostrar errores de la misma forma que mostramos cualquier otro dato de una colección.

Mostraremos los errores en la parte de arriba de nuestro layout:

```
<template name="layout">
<div class="container">
  {{> header}}
  {{> errors}}
  <div id="main">
    {{> yield}}
  </div>
</div>
</template>
```

client/templates/application/layout.html

Ahora vamos a crear la plantilla de error en `errors.html`:

```
<template name="errors">
<div class="errors">
  {{#each errors}}
    {{> error}}
  {{/each}}
</div>
</template>

<template name="error">
<div class="alert alert-danger" role="alert">
  <button type="button" class="close" data-dismiss="alert">&times;</button>
  {{message}}
</div>
</template>
```

client/templates/includes/errors.html

Te habrás dado cuenta de que estamos definiendo dos plantillas en un solo archivo. Hasta ahora hemos tratado de mantener la regla “un archivo, una plantilla”, pero con Meteor podemos poner todas las plantillas en un único fichero y también funcionaría (¡Aunque tendríamos un fichero `main.html` muy confuso!).

En este caso, como las plantillas son muy pequeñas, haremos una excepción y las pondremos en el mismo fichero para que nuestro repositorio quede un poco más limpio.

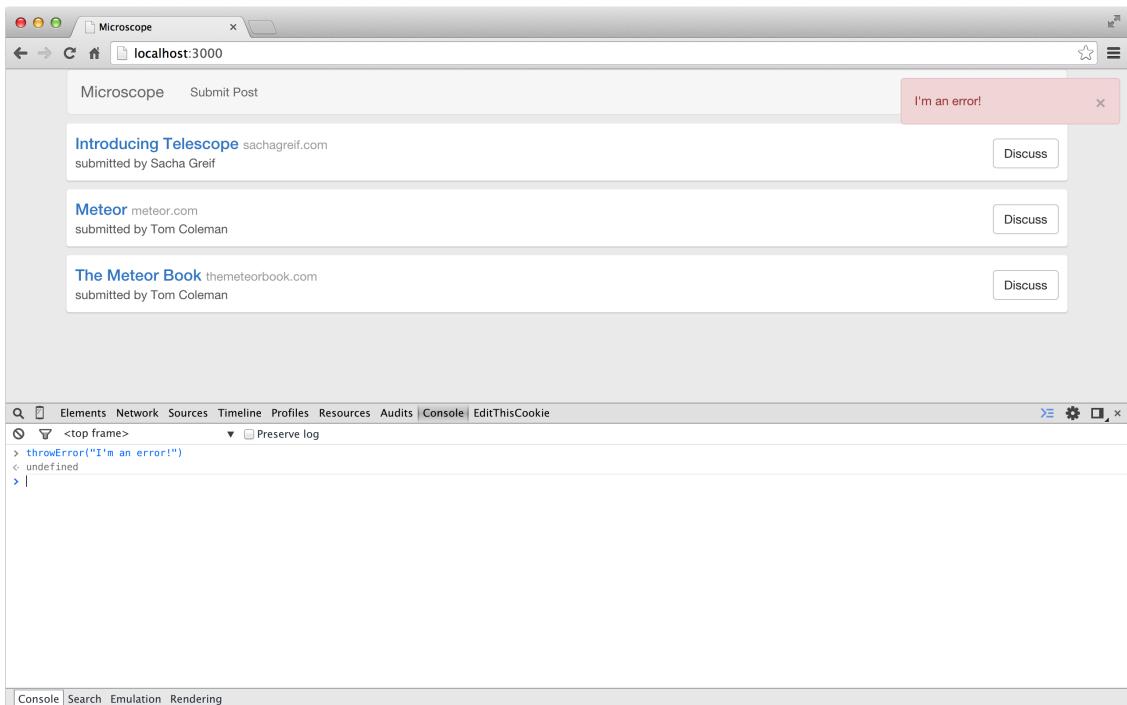
Solo nos falta integrar el ayudante de plantilla y ¡Estará listo!

```
Template.errors.helpers({
  errors: function() {
    return Errors.find();
  }
});
```

client/templates/includes/errors.js

Ya puedes probar nuestros mensajes de error manualmente. Abre una consola en el navegador y escribe:

```
throwError("I'm an error!");
```



Mostrando errores.

[Ver en GitHub](#)

[Lanzar instancia](#)

Llegados a este punto es importante hacer distinción entre los errores a “nivel de aplicación” y los errores a “nivel de código”.

Los errores a **nivel de aplicación** son, generalmente, generados por el usuario, y los usuarios pueden interactuar con ellos. Esto incluye cosas como los errores de validación, errores de permisos, errores de “no encontrado”, etc. Estos son la clase de errores que queremos mostrar al usuario para ayudarle a arreglar el problema que hemos encontrado.

Los errores a **nivel de código**, en cambio, se generan de forma inesperada por fallos en nuestro código, y que probablemente *no* queremos mostrarle al usuario directamente, si no que queremos registrarlos con alguna clase de sistema de registro de errores (como por ejemplo **Kadira**).

En este capítulo, nos centraremos en tratar con el primer tipo de error, no en capturar fallos.

Ya sabemos cómo mostrar los errores, pero todavía tenemos que crearlos antes de poder verlos. Ya hemos

implementado un buen escenario para mostrarlos: el aviso de post duplicado. Sencillamente remplazaremos las llamadas a `alert` en el ayudante de eventos de `postSubmit` con la nueva función `throwError` que acabamos de crear:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return throwError(error.reason);

      // show this result but route anyway
      if (result.postExists)
        throwError('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

Ya que estamos, hacemos lo mismo con el ayudante de `postEdit`:

```
Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        throwError(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },
  //...
});
```

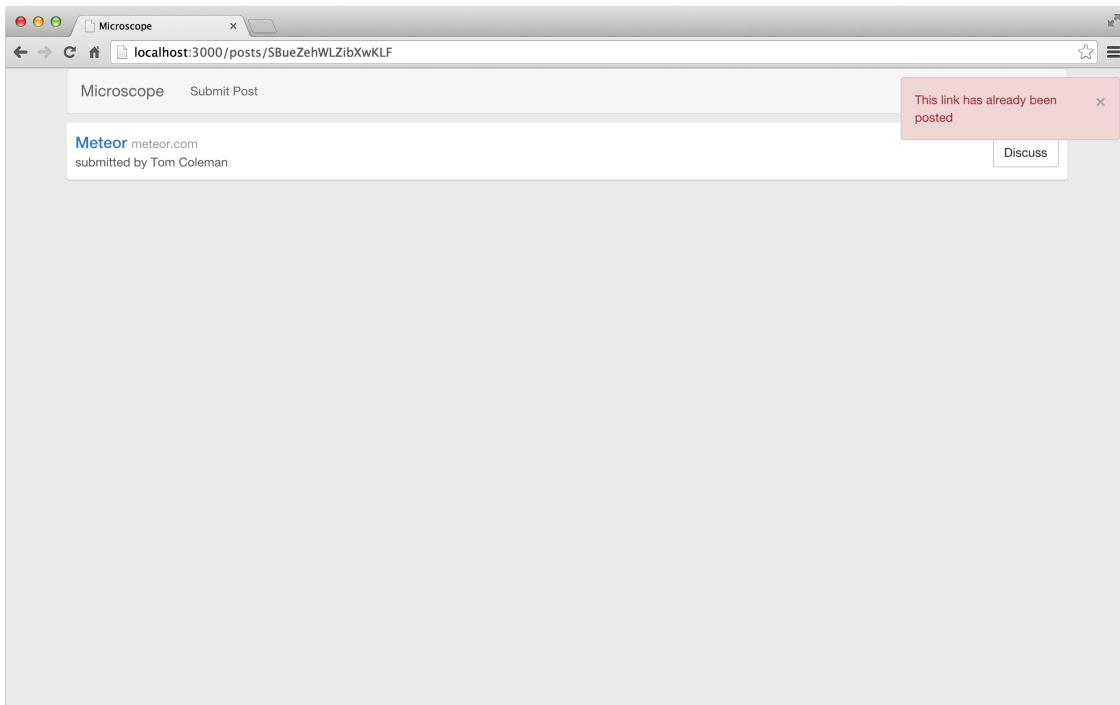
client/templates/posts/post_edit.js

Usando el mecanismo de presentar errores.

[Ver en GitHub](#)

[Lanzar instancia](#)

Vamos a probar: intenta crear un post e introduce la URL `http://meteor.com`. Como esta URL ya existe en un post en nuestros datos de ejemplo, deberíamos ver:



Te habrás dado cuenta de que los mensajes de error desaparecen por sí mismos después de unos segundos. Esto es debido un poco de magia CSS incluida en la hoja de estilos que hemos añadido al principio del libro:

```
@keyframes fadeOut {  
  0% {opacity: 0;}  
  10% {opacity: 1;}  
  90% {opacity: 1;}  
 100% {opacity: 0;}  
}  
  
//...  
  
.alert {  
  animation: fadeOut 2700ms ease-in 0s 1 forwards;  
  //...  
}
```

client/stylesheets/style.css

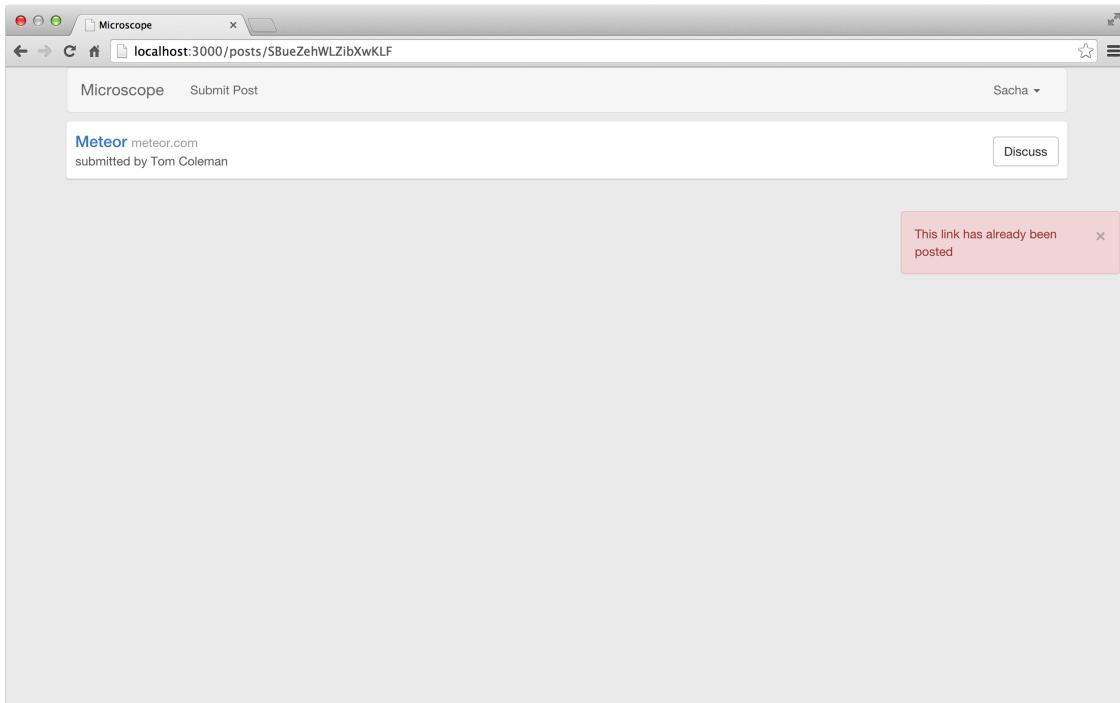
Estamos definiendo una animación CSS `fadeOut` que especifica cuatro keyframes para la propiedad opacidad (al 0%, 10%, 90% y 100% del total de la duración de la animación), y aplicando esta animación a la clase `alert`.

La animación se ejecuta durante 2700 milisegundos en total, usa la ecuación de tiempo `ease-in`, se inicia con un retardo de 0 segundos, se ejecuta una sola vez, y finalmente permanece en el último keyframe una vez que se ha terminado de ejecutar.

Te estarás preguntando porqué estamos usando animaciones basadas en CSS (que son predeterminadas y están fuera del control de nuestra aplicación), en lugar de animaciones controladas por el propio Meteor.

Si bien Meteor proporciona soporte para insertar animaciones, queríamos en este capítulo enfocarnos en los errores. Así que, por ahora, usaremos una “simple” animación CSS y dejaremos esas cosas para el capítulo de las animaciones.

Esto funciona, pero si lanzamos varios errores (enviando el mismo enlace tres veces por ejemplo) notarás que se apilarán uno encima de otro:



Esto pasa porque mientras los elementos `.alert` están desapareciendo *visualmente*, todavía están presentes en el DOM. Necesitamos corregir esto.

Esta es la clase de situaciones en las que Meteor reuce. Puesto que la colección `Errors` es reactiva, ¡Todo lo que necesitamos para deshacernos de estos antiguos errores es eliminarlos de la colección!

Usaremos `Meteor.setTimeout` para especificar una función que se ejecute después de que se expire el tiempo de espera (en este caso, 3000 milisegundos).

```
Template.errors.helpers({
  errors: function() {
    return Errors.find();
  }
});

Template.error.onRendered(function() {
  var error = this.data;
  Meteor.setTimeout(function () {
    Errors.remove(error._id);
  }, 3000);
});
```

client/templates/includes/errors.js

Limpiar errores después de 3 segundos.

[Ver en GitHub](#)

[Lanzar instancia](#)

La llamada a `onRendered` se lanza una vez que nuestra plantilla ha sido renderizada en el navegador. Dentro de esta llamada, `this` hace referencia a la instancia actual de la plantilla, y `this.data` da acceso a los datos del objeto que está siendo renderizado (en nuestro caso, un error).

Aún no hemos impuesto ninguna clase de validación en nuestro formulario. Queremos que los usuarios indiquen una URL y un título para cada post. Vamos a asegurarnos de esto sea así.

Haremos dos cosas para comprobar campos ausentes: primero, añadiremos una clase css especial `has-error` al `div` padre de cualquier campo de formulario. Segundo, mostraremos un mensaje de error justo debajo del campo.

Para comenzar, vamos a preparar nuestra plantilla `postSubmit` para aceptar este nuevo tipo de ayudantes:

```

<template name="postSubmit">
  <form class="main form page">
    <div class="form-group {{errorClass 'url'}}">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="" placeholder="Your URL" class="form-control"/>
        <span class="help-block">{{errorMessage 'url'}}</span>
      </div>
    </div>
    <div class="form-group {{errorClass 'title'}}">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="" placeholder="Name your post" class="form-control"/>
        <span class="help-block">{{errorMessage 'title'}}</span>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary"/>
  </form>
</template>

```

client/templates/posts/post_submit.html

Fíjate que estamos pasando parámetros (`url` y `title` respectivamente) a cada ayudante. Esto nos permite reutilizar el mismo ayudante, modificando su comportamiento según el parámetro.

Ahora la parte divertida: hacer que estos ayudantes hagan realmente algo.

Usaremos la **Session** para almacenar un objeto `postSubmitErrors` que contendrá el potencial mensaje de error. Según el usuario interactúa con el formulario, este objeto irá cambiando, por lo que irá cambiando reactivamente el estilo y contenido del formulario.

Primero, iniciamos el objeto donde se crea la plantilla `postSubmit`. Esto nos asegura que el usuario no ve un mensaje de error antiguo que se haya quedado de una visita anterior a esta página.

Después definimos dos ayudantes de plantilla. Ambos buscarán la propiedad `field` del objeto `Session.get('postSubmitErrors')` (donde `field` será `url` o `title` dependiendo del ayudante desde que el que se llame).

Mientras que `errorMessage` solo devuelve el mensaje en sí mismo, `errorClass` comprueba la presencia de un mensaje y devuelve `has-error` en caso de que exista un mensaje.

```

Template.postSubmit.onCreate(function() {
  Session.set('postSubmitErrors', {});
});

Template.postSubmit.helpers({
  errorMessage: function(field) {
    return Session.get('postSubmitErrors')[field];
  },
  errorClass: function (field) {
    return !!Session.get('postSubmitErrors')[field] ? 'has-error' : '';
  }
});

//...

```

client/templates/posts/post_submit.js

Puedes comprobar que nuestros ayudantes están funcionando correctamente abriendo una consola en el navegador y escribiendo la siguiente línea de código:

```
Session.set('postSubmitErrors', {title: 'Warning! Intruder detected. Now releasing robo-dogs.'});
```

Consola del navegador

The screenshot shows a browser window titled "Microscope" at the URL "localhost:3000/submit". The page contains a form with fields for "URL" and "Title". The "Title" field is highlighted with a red border. Below the form, a message reads "Warning! Intruder detected. Now releasing robo-dogs." A "Submit" button is visible. At the bottom of the browser, the developer tools' "Console" tab is active, displaying the command `Session.set('postSubmitErrors', {title: 'Warning! Intruder detected. Now releasing robo-dogs.'});` which was run in the console.

El próximo paso es enganchar el objeto de sesión `postSubmitErrors` al formulario.

Antes de hacerlo, crearemos una nueva función `validatePost` en el fichero `posts.js` que mire en el objeto `post`, y devuelva un objeto `errors` con cualquier error relevante (que los campos `title` o `url` no están presentes):

```
//...
validatePost = function (post) {
  var errors = {};

  if (!post.title)
    errors.title = "Please fill in a headline";

  if (!post.url)
    errors.url = "Please fill in a URL";

  return errors;
}

//...
```

lib/collections/posts.js

Llamaremos a esta función desde el ayudante de eventos de `postSubmit`:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    var errors = validatePost(post);
    if (errors.title || errors.url)
      return Session.set('postSubmitErrors', errors);

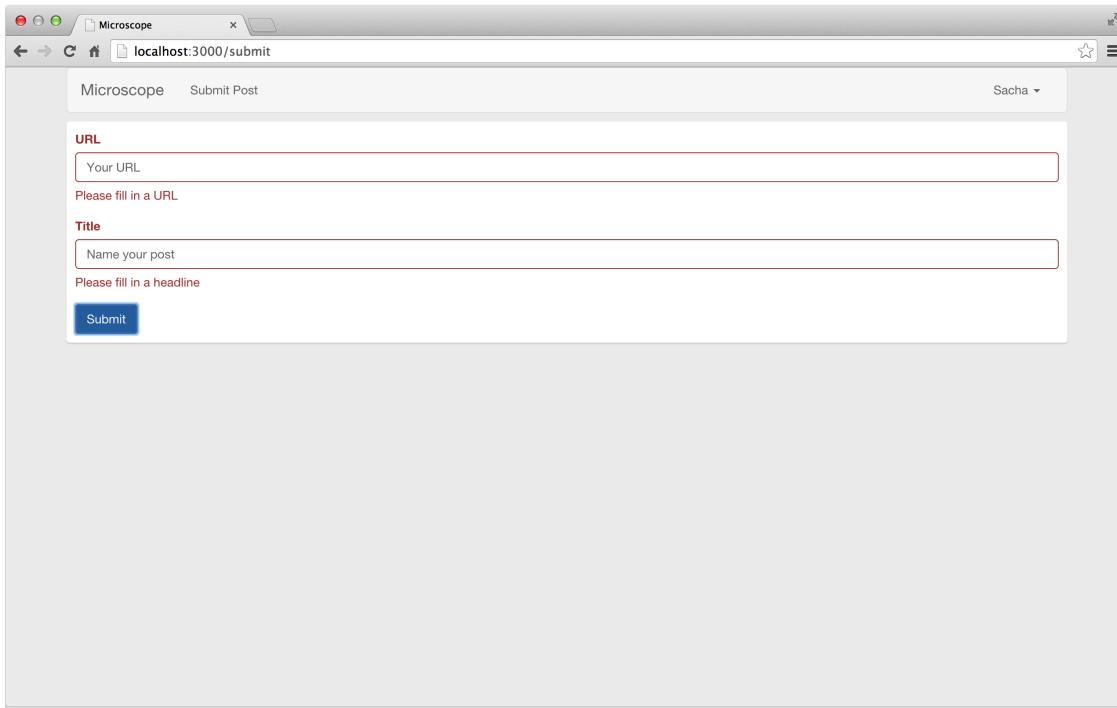
    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return throwError(error.reason);

      // show this result but route anyway
      if (result.postExists)
        throwError('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

Fíjate que estamos usando `return` para abortar la ejecución del ayudante si hay errores, no porque queramos devolver ningún valor concreto.



¿Ya hemos terminado?. Estamos validando la presencia de una URL y un título en el lado del cliente, ¿Pero que pasa con el servidor? Después de todo, alguien podría añadir un post llamando manualmente al método `postInsert` desde la consola del navegador.

Aunque pienses que no necesitamos mostrar ningún mensaje de error en el servidor, podemos hacer uso de la función `validatePost`. Solo que esta vez la llamaremos desde el método `postInsert` también, y no sólo desde el ayudante de eventos:

```

Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var errors = validatePost(postAttributes);
    if (errors.title || errors.url)
      throw new Meteor.Error('invalid-post', "You must set a title and URL for your post");

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});

```

lib/collections/posts.js

De nuevo, los usuarios no deberían nunca ver este mensaje “You must set a title and URL for your post”. Sólo se mostrará si alguien quiere saltarse los controles que hemos dispuesto concienzudamente, y usa la consola del navegador.

Para probarlo, abre una consola del navegador y prueba a enviar un post sin URL:

```
Meteor.call('postInsert', {url: '', title: 'No URL here!'});
```

Si hemos hecho bien nuestro trabajo, obtendrás un montón de código extraño junto con un mensaje “You must set a title and URL for your post”.

Validar el contenido de la post al enviar.

[Ver en GitHub](#)

[Lanzar instancia](#)

Pare redondear las cosas, aplicaremos la misma validación a nuestro formulario de edición de posts. El código es muy similar. Primero, la plantilla:

```
<template name="postEdit">
  <form class="main form page">
    <div class="form-group {{errorClass 'url'}}">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="{{url}}" placeholder="Your URL" class="form-control"/>
        <span class="help-block">{{errorMessage 'url'}}
      </div>
    </div>
    <div class="form-group {{errorClass 'title'}}">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="{{title}}" placeholder="Name your post" class="form-control"/>
        <span class="help-block">{{errorMessage 'title'}}
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary submit"/>
    <hr/>
    <a class="btn btn-danger delete" href="#">Delete post</a>
  </form>
</template>
```

client/templates/posts/post_edit.html

Luego los ayudantes de plantilla:

```

Template.postEdit.onCreate(function() {
  Session.set('postEditErrors', {});
});

Template.postEdit.helpers({
  errorMessage: function(field) {
    return Session.get('postEditErrors')[field];
  },
  errorClass: function (field) {
    return !!Session.get('postEditErrors')[field] ? 'has-error' : '';
  }
});

Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    }

    var errors = validatePost(postProperties);
    if (errors.title || errors.url)
      return Session.set('postEditErrors', errors);

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        throwError(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },
  'click .delete': function(e) {
    e.preventDefault();

    if (confirm("Delete this post?")) {
      var currentPostId = this._id;
      Posts.remove(currentPostId);
      Router.go('postsList');
    }
  }
});

```

client/templates/posts/post_edit.js

Tal y como hicimos en el formulario de envío de posts, queremos validar nuestros posts en el servidor. Excepto que recordarás que no estamos usando un método para editar los posts, si no una llamada `update` directamente desde el lado cliente.

Esto significa que tenemos que añadir una nueva regla `deny`:

```
//...

Posts.deny({
  update: function(userId, post, fieldNames, modifier) {
    var errors = validatePost(modifier.$set);
    return errors.title || errors.url;
  }
});

//...
```

lib/collections/posts.js

Fíjate que el argumento `post` se refiere al *post existente*. En este caso, queremos validar la *actualización*, que es por lo que estamos llamando a `validatePost` con el contenido del modificador de la propiedad `$set` (como en `Posts.update({$set: {title: ..., url: ...}})`).

Esto funciona porque `modifier.$set` contiene las mismas dos propiedades `title` y `url` que el objeto `post`. Por supuesto, esto quiere decir que cualquier actualización parcial que afecte solo a `title` o `url` fallará, pero en la práctica no debería ser una complicación.

Te habrás dado cuenta de que esta es nuestra segunda llamada `deny`. Cuando añadimos múltiples llamadas `deny`, la operación fallará si uno de ellos devuelve `true`. En este caso, esto significa que el `update` solo será satisfactorio si estamos modificando el `title` y la `url`, y ninguna de ellas esté vacía.

Validar el contenido del post al
editar.

[Ver en GitHub](#)

[Lanzar instancia](#)

Durante nuestro trabajo en los errores, hemos construido un modelo reutilizable, ¿por qué no ponerlo dentro de un paquete y compartirlo con el resto de la comunidad Meteor?

Para empezar, tenemos que asegurarnos de que tenemos una cuenta de desarrollador Meteor. Puedes hacerte una en [meteor.com](#), pero es muy probable que ya lo hayas hecho ¡cuando te registraste para el libro! En cualquier caso, deberías saber cuál es tu nombre de usuario, porque lo utilizaremos bastante durante este capítulo.

En este capítulo usaremos `tmeasday` como usuario – puedes cambiarlo por el tuyo.

En primer lugar, necesitamos crear una estructura de carpetas para nuestro paquete. Podemos usar el comando `meteor create --package tmeasday:errors` para conseguirla. Fíjate que Meteor ha creado una carpeta llamada `packages/tmeasday:errors/`, con algunos ficheros dentro. Comenzaremos por editar `package.js`, el archivo que informa a Meteor de cómo debe utilizar el paquete, y los símbolos y funciones que exporta.

```
Package.describe({
  name: "tmeasday:errors",
  summary: "A pattern to display application errors to the user",
  version: "1.0.0"
});

Package.onUse(function(api, where) {
  api.versionsFrom('0.9.0');

  api.use(['minimongo', 'mongo-livedata', 'templating'], 'client');

  api.addFiles(['errors.js', 'errors_list.html', 'errors_list.js'], 'client');

  if (api.export)
    api.export('Errors');
});

packages/tmeasday:errors/package.js
```

Cuando desarrollamos un paquete para su uso en el mundo-real, es una buena práctica llenar la sección `Package.describe` con la URL del repositorio Git (como por ejemplo, <https://github.com/tmeasday/meteor-errors.git>). De esta manera, los usuarios pueden acceder al código fuente, y (si usas GitHub) ver el README del paquete en Atmosphere.

Vamos a añadir al paquete los tres archivos que se pasan en la llamada a `add_files`. Podemos usar los mismos que tenemos para Microscope, haciendo solo, unos pequeños cambios para los espacios de nombres y para dejar la API un poco más limpia:

```
Errors = {
  // Local (client-only) collection
  collection: new Mongo.Collection(null),

  throw: function(message) {
    Errors.collection.insert({message: message, seen: false})
  }
};
```

```
packages/tmeasday:errors/errors.js
```

```
<template name="meteorErrors">
<div class="errors">
  {{#each errors}}
    {{> meteorError}}
  {{/each}}
</div>
</template>

<template name="meteorError">
<div class="alert alert-danger" role="alert">
  <button type="button" class="close" data-dismiss="alert">&times;</button>
  {{message}}
</div>
</template>
```

```
packages/tmeasday:errors/errors_list.html
```

```
Template.meteorErrors.helpers({
  errors: function() {
    return Errors.collection.find();
  }
});

Template.meteorError.rendered = function() {
  var error = this.data;
  Meteor.setTimeout(function () {
    Errors.collection.remove(error._id);
  }, 3000);
};
```

```
packages/tmeasday:errors/errors_list.js
```

Vamos a probar el paquete localmente con Microscope para asegurarnos de que nuestros cambios funcionan. Para enlazar el paquete en nuestro proyecto, ejecutaremos `meteor add tmeasday:errors`. A continuación, debemos eliminar los archivos a los que reemplaza el nuevo paquete:

```
rm client/helpers/errors.js
rm client/templates/includes/errors.html
rm client/templates/includes/errors.js
```

```
Eliminando los archivos antiguos
```

Otra cosa que debemos hacer son algunos pequeños cambios en el código de la aplicación para que use la API correcta:

```
  {{> header}}
  {{> meteorErrors}}
```

```
client/templates/application/layout.html
```

```
Meteor.call('postInsert', post, function(error, id) {  
  if (error) {  
    // display the error to the user  
    Errors.throw(error.reason);  
}
```

client/templates/posts/post_submit.js

```
Posts.update(currentPostId, {$set: postProperties}, function(error) {  
  if (error) {  
    // display the error to the user  
    Errors.throw(error.reason);  
  
    // show this result but route anyway  
  if (result.postExists)  
    Errors.throw('This link has already been posted');
```

client/templates/posts/post_edit.js

Creado y enlazado un paquete
básico.

[Ver en GitHub](#)

[Lanzar instancia](#)

Una vez hechos estos cambios, deberíamos ver el mismo comportamiento que teníamos con el código sin empaquetar.

El primer paso en el desarrollo de un paquete es probarlo contra una aplicación, pero el siguiente es escribir un conjunto de tests que evalúen adecuadamente el comportamiento del paquete. Meteor incluye Tinytest, que permite ejecutar este tipo de pruebas de forma fácil y, de esta forma, tener la conciencia tranquila cuando compartimos el paquete con los demás.

Vamos a crear un archivo que usa Tinytest para ejecutar tests contra el código de los errores.

```

Tinytest.add("Errors - collection", function(test) {
  test.equal(Errors.collection.find({}).count(), 0);

  Errors.throw('A new error!');
  test.equal(Errors.collection.find({}).count(), 1);

  Errors.collection.remove({});
});

Tinytest.addAsync("Errors - template", function(test, done) {
  Errors.throw('A new error!');
  test.equal(Errors.collection.find({}).count(), 1);

  // render the template
  UI.insert(UI.render(Template.meteorErrors), document.body);

  Meteor.setTimeout(function() {
    test.equal(Errors.collection.find({}).count(), 0);
    done();
  }, 3500);
});

```

packages/tmeasday:errors/errors_tests.js

Con estos tests comprobamos que las funciones básicas de `Meteor.Errors` funcionan correctamente, así como que el código `mostrado` en la plantilla sigue funcionando bien.

No vamos a cubrir los aspectos específicos sobre cómo escribir tests de paquetes (porque la API todavía no está acabada y podría cambiar mucho), pero viendo el código, puedes hacerte una idea cómo funciona.

Para decirle a Meteor que ejecute los tests, añadimos este código a `package.js`

```

Package.onTest(function(api) {
  api.use('tmeasday:errors', 'client');
  api.use(['tinytest', 'test-helpers'], 'client');

  api.addFiles('errors_tests.js', 'client');
});

```

packages/tmeasday:errors/package.js

Tests añadidos al paquete.

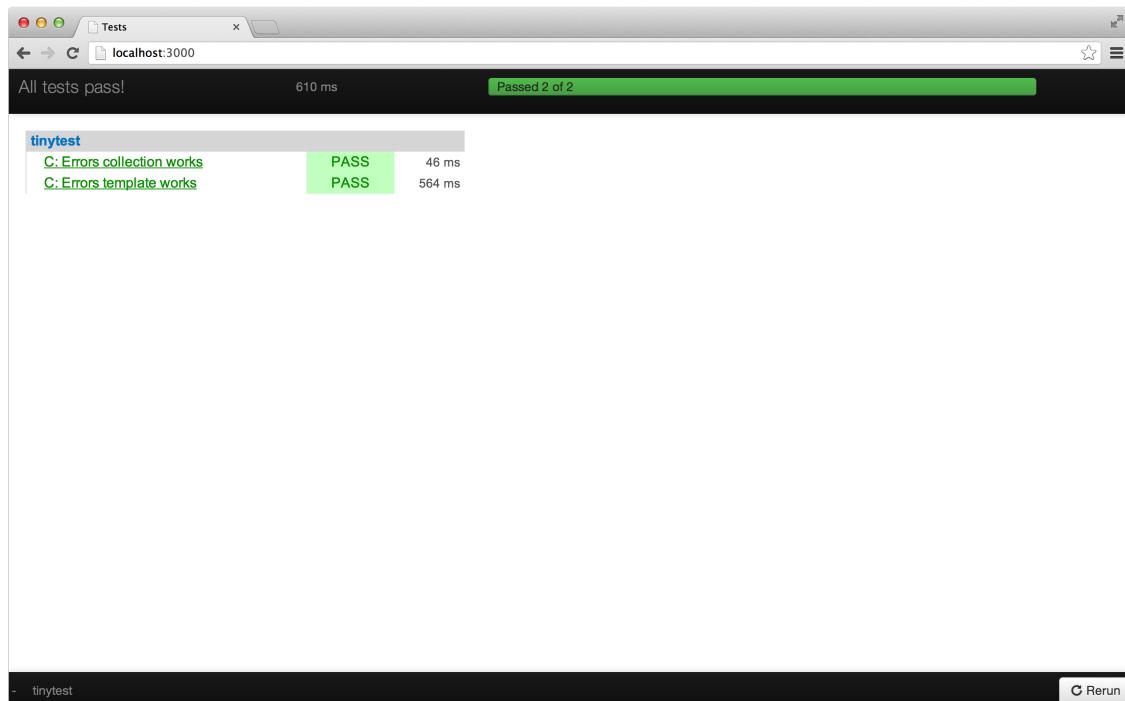
[Ver en GitHub](#)

[Lanzar instancia](#)

Ya podemos ejecutar los tests con:

meteor test-packages tmeasday:errors

Terminal



Ahora, queremos liberar el paquete y ponerlo a disposición de todo el mundo. Para ello tendremos que subirlo al servidor de paquetes de Meteor y, hacerlo miembro del repositorio Atmosphere.

Afortunadamente, es muy fácil. Entramos en el directorio del paquete, y ejecutamos `meteor publish --create`:

```
cd packages/tmeasday:errors
meteor publish --create
```

Terminal

Ahora que hemos publicado el paquete, podemos eliminarlo del proyecto y luego añadirlo de nuevo directamente:

```
rm -r packages/errors
meteor add tmeasday:errors
```

Terminal (ejecutar desde el directorio raíz de la aplicación)

Paquete eliminado del árbol de desarrollo.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora debemos ver a Meteor descargar nuestro paquete por primera vez. ¡Bien hecho!

Como de costumbre, asegúrate de deshacer los cambios antes de continuar (o mantenerlos, teniéndolos en cuenta en el resto del libro).

El objetivo de un sitio de noticias es crear una comunidad de usuarios, y será difícil hacerlo sin que puedan a hablar unos con otros. En este capítulo, vamos a agregar los comentarios.

Empezaremos creando una nueva colección para almacenar los comentarios.

```
Comments = new Mongo.Collection('comments');
```

```
lib/collections/comments.js
```

```

// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // create two users
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachaId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sacha = Meteor.users.findOne(sachaId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sacha._id,
    author: sacha.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: new Date(now - 7 * 3600 * 1000)
  });

  Comments.insert({
    postId: telescopeId,
    userId: tom._id,
    author: tom.profile.name,
    submitted: new Date(now - 5 * 3600 * 1000),
    body: 'Interesting project Sacha, can I get involved?'
  });

  Comments.insert({
    postId: telescopeId,
    userId: sacha._id,
    author: sacha.profile.name,
    submitted: new Date(now - 3 * 3600 * 1000),
    body: 'You sure can Tom!'
  });

  Posts.insert({
    title: 'Meteor',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://meteor.com',
    submitted: new Date(now - 10 * 3600 * 1000)
  });

  Posts.insert({
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteorbook.com',
    submitted: new Date(now - 12 * 3600 * 1000)
  });
}

```

server/fixtures.js

No olvidemos que debemos publicar y suscribir la nueva colección:

```
Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function() {
  return Comments.find();
});
```

server/publications.js

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('posts'), Meteor.subscribe('comments')];
  }
});
```

lib/router.js

Añadidos comentarios a la colección, pub/sub y datos de
p...

[Ver en GitHub](#)

[Lanzar instancia](#)

Ten en cuenta que para que se carguen los nuevos datos de prueba, es necesario ejecutar `meteor reset`. Después de la restauración, ¡no olvides crear una nueva cuenta y volver a entrar!

En primer lugar, hemos creado un par de usuarios (inventados), insertándolos en la base de datos y usando los `ids` para seleccionarlos después en la base de datos. Luego añadimos un comentario de cada usuario al primer post, enlazando el comentario al post (con `postId`), y el usuario (con `userId`). Además, añadimos la fecha y el cuerpo de cada comentario, junto un campo denormalizado denominado `author`.

Además, hemos extendido nuestro router para que espere a un array que contiene las dos colecciones, `comentarios` y `posts`.

Está bien tener comentarios en la base de datos, pero habrá que mostrarlos en la página de discusión. Este proceso ya nos debe ser familiar:

```
<template name="postPage">
<div class="post-page page">
  {{> postItem}}
  <ul class="comments">
    {{#each comments}}
      {{> commentItem}}
    {{/each}}
  </ul>
</div>
</template>
```

client/templates/posts/post_page.html

```
Template.postPage.helpers({
  comments: function() {
    return Comments.find({postId: this._id});
  }
});
```

client/templates/posts/post_page.js

Ponemos el bloque `{{#each comments}}` dentro de la plantilla del post, por lo que `this` es un post para el ayudante `comments`. Para encontrar los comentarios adecuados, buscamos los que están vinculados a ese post a través de `postId`.

Con todo lo que hemos aprendido acerca de ayudantes y plantillas, sabemos que mostrar un comentario es bastante sencillo. Vamos a crear un nuevo directorio `comments` dentro de `templates` para almacenar toda la información acerca de los comentarios, y una nueva plantilla `commentItem` dentro:

```
<template name="commentItem">
<li>
  <h4>
    <span class="author">{{author}}</span>
    <span class="date">on {{submittedText}}</span>
  </h4>
  <p>{{body}}</p>
</li>
</template>
```

client/templates/comments/comment_item.html

Vamos a crear rápidamente un ayudante de plantilla para dar a nuestra fecha de envío `submitted` un formato más amigable:

```
Template.commentItem.helpers({
  submittedText: function() {
    return this.submitted.toString();
  }
});
```

client/templates/comments/comment_item.js

A continuación, vamos a mostrar el número de comentarios de cada post:

```

<template name="postItem">
<div class="post">
  <div class="post-content">
    <h3><a href="{{url}}>{{title}}</a><span>{{domain}}</span></h3>
    <p>
      submitted by {{author}},
      <a href="{{pathFor 'postPage'}}>{{commentsCount}} comments</a>
      {{#if ownPost}}<a href="{{pathFor 'postEdit'}}>Edit</a>{{/if}}
    </p>
  </div>
  <a href="{{pathFor 'postPage'}}>Discuss</a>
</div>
</template>

```

client/templates/posts/post_item.html

Y añadimos el ayudante `commentsCount` a `post_item.js`:

```

Template.postItem.helpers({
  ownPost: function() {
    return this.userId === Meteor.userId();
  },
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  },
  commentsCount: function() {
    return Comments.find({postId: this._id}).count();
  }
});

```

client/templates/posts/post_item.js

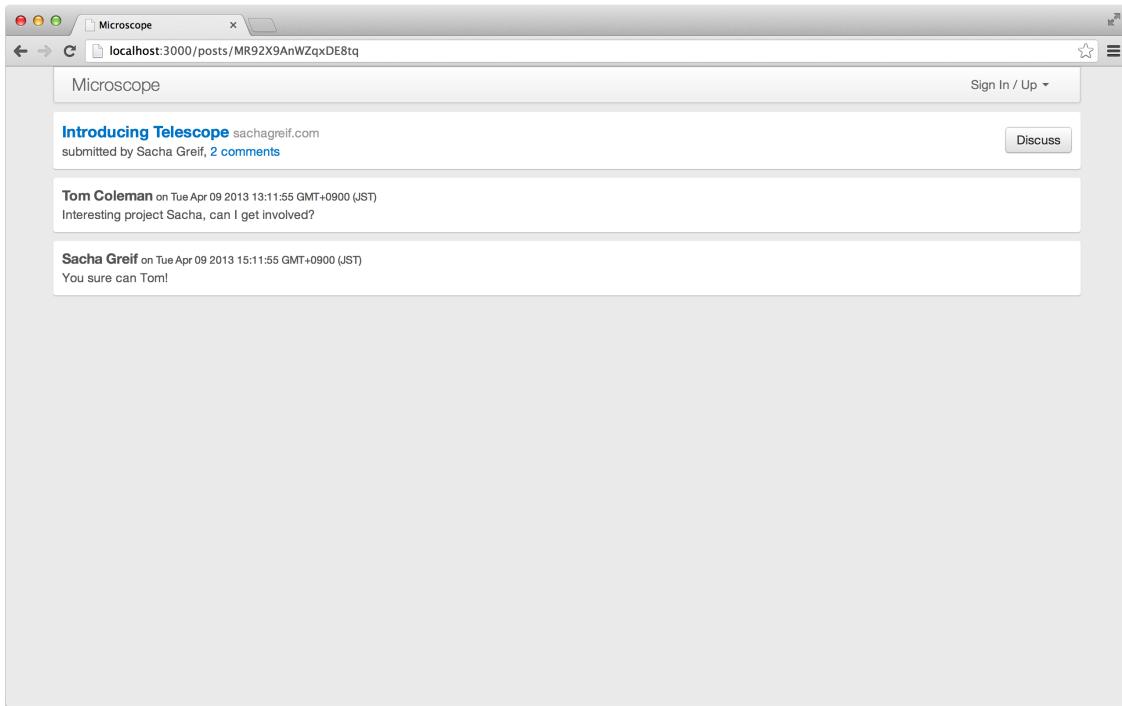
Mostrar los comentarios en

`postPage`.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora deberíamos ser capaces de mostrar nuestros comentarios de prueba y ver algo como esto:



Vamos a añadir una forma de que los usuarios puedan hacer nuevos comentarios. El proceso será bastante similar a como ya hemos hecho para permitir a los usuarios crear nuevos posts.

Empezaremos añadiendo un área de envío en la parte inferior de cada post:

```
<template name="postPage">
  <div class="post-page page">
    {{> postItem}}
    <ul class="comments">
      {{#each comments}}
        {{> commentItem}}
      {{/each}}
    </ul>
    {{#if currentUser}}
      {{> commentSubmit}}
    {{else}}
      <p>Please log in to leave a comment.</p>
    {{/if}}
  </div>
</template>
```

client/templates/posts/post_page.html

Y a continuación, crear la plantilla del formulario para los comentarios:

```

<template name="commentSubmit">
  <form name="comment" class="comment-form form">
    <div class="form-group {{errorClass 'body'}}">
      <div class="controls">
        <label for="body">Comment on this post</label>
        <textarea name="body" id="body" class="form-control" rows="3"></textarea>
        <span class="help-block">{{errorMessage 'body'}}</span>
      </div>
    </div>
    <button type="submit" class="btn btn-primary">Add Comment</button>
  </form>
</template>

```

client/templates/comments/comment_submit.html

Para enviar comentarios, llamaremos a un método `comment` en el fichero `comment_submit.js` que funciona de forma similar a lo que hicimos para al enviar posts:

```

Template.commentSubmit.onCreate(function() {
  Session.set('commentSubmitErrors', {});
});

Template.commentSubmit.helpers({
  errorMessage: function(field) {
    return Session.get('commentSubmitErrors')[field];
  },
  errorClass: function (field) {
    return !!Session.get('commentSubmitErrors')[field] ? 'has-error' : '';
  }
});

Template.commentSubmit.events({
  'submit form': function(e, template) {
    e.preventDefault();

    var $body = $(e.target).find('[name=body]');
    var comment = {
      body: $body.val(),
      postId: template.data._id
    };

    var errors = {};
    if (!comment.body) {
      errors.body = "Please write some content";
      return Session.set('commentSubmitErrors', errors);
    }

    Meteor.call('commentInsert', comment, function(error, commentId) {
      if (error){
        throwError(error.reason);
      } else {
        $body.val('');
      }
    });
  }
});

```

client/templates/comments/comment_submit.js

Al igual que anteriormente establecimos un método `post` en el servidor, vamos a hacer lo mismo para crear comentarios, comprobar que todo está bien, y finalmente insertar el nuevo comentario dentro de su colección.

```

Comments = new Mongo.Collection('comments');

Meteor.methods({
  commentInsert: function(commentAttributes) {
    check(this.userId, String);
    check(commentAttributes, {
      postId: String,
      body: String
    });

    var user = Meteor.user();
    var post = Posts.findOne(commentAttributes.postId);

    if (!post)
      throw new Meteor.Error('invalid-comment', 'You must comment on a post');

    comment = _.extend(commentAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    return Comments.insert(comment);
  }
});

```

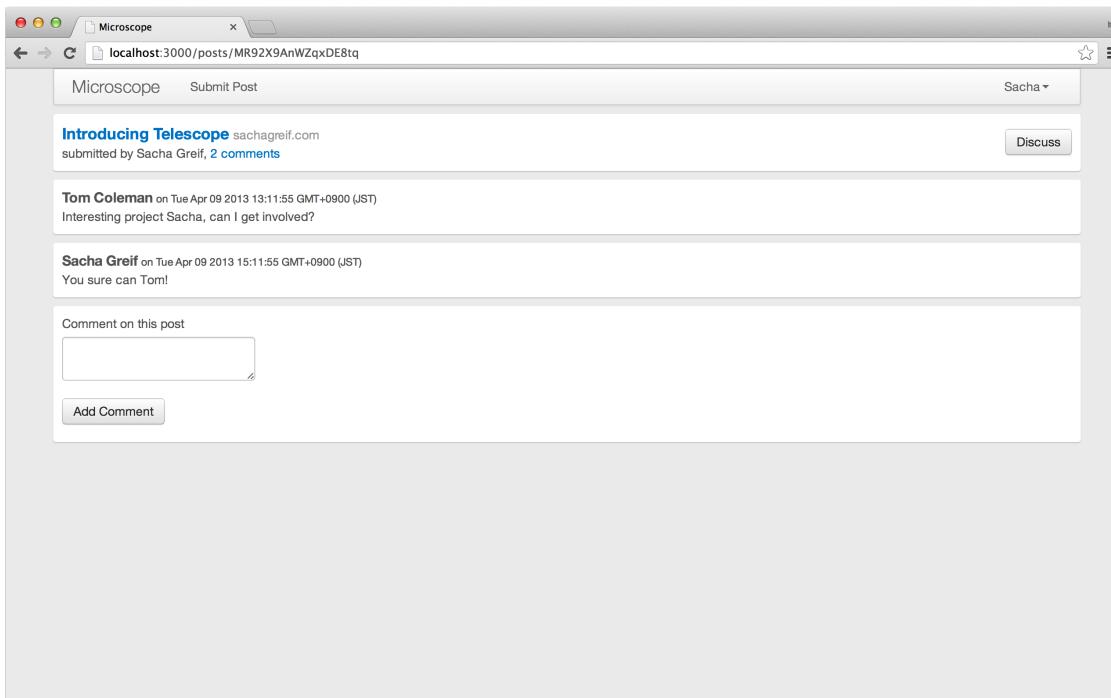
lib/collections/comments.js

Creado el formulario de envío de comentarios.

[Ver en GitHub](#)

[Lanzar instancia](#)

Comprobamos que el usuario está conectado, que el comentario tiene cuerpo, y que está vinculado a un post.



Tal como están las cosas, publicamos todos los comentarios de todos los posts a todos los clientes conectados. ¿No estaremos derrochando recursos? Después de todo, en un momento dado solo usamos un pequeño subconjunto de estos datos. Así que vamos a mejorar nuestras publicaciones y suscripciones para controlar exactamente qué comentarios se publican.

Si lo pensamos bien, el único momento en el que necesitamos suscribirnos a la publicación `comments` es cuando un usuario accede a la página de un post individual, y solo hay que cargar el subconjunto de comentarios relacionados con ese post en particular.

El primer paso va a ser cambiar la forma de suscribirse a los comentarios. Hasta ahora, nos hemos estado suscribiendo a nivel `router`, lo que significa que cargamos todos nuestros datos una vez cuando este se inicializa.

Pero ahora queremos que nuestra suscripción dependa de un parámetro de ruta, y, obviamente, este parámetro puede cambiar en cualquier momento. Así que tendremos que cambiar nuestro código de suscripción desde el nivel de `router` al nivel de `ruta`.

Esto tiene otra consecuencia: en vez de cargar nuestros datos cuando se inicializa la aplicación, ahora los cargaremos cada vez que llegamos a una ruta concreta. Esto significa que ahora tendremos *tiempos de carga* mientras navegamos por la aplicación. Esto es un inconveniente inevitable a no ser que queramos cargar siempre todos los datos.

Primero, dejaremos de pre-cargar todos los comentarios en el bloque `configure`, eliminando la línea `Meteor.subscribe('comments')` (dicho de otra manera, volvemos a lo que teníamos anteriormente):

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return Meteor.subscribe('posts');
  }
});
```

lib/router.js

Y añadiremos una nueva función `waitOn` a nivel de `ruta` en la ruta `postPage`:

```
//...
Router.route('/posts/:_id', {
  name: 'postPage',
  waitOn: function() {
    return Meteor.subscribe('comments', this.params._id);
  },
  data: function() { return Posts.findOne(this.params._id); }
});
```

Estamos pasando `this.params._id` como argumento a la suscripción. Así que, utilicemos esa nueva información para asegurarnos que limitamos el conjunto de datos a los comentarios que pertenecen al post actual:

```
Meteor.publish('posts', function() {
  return Posts.find();
});

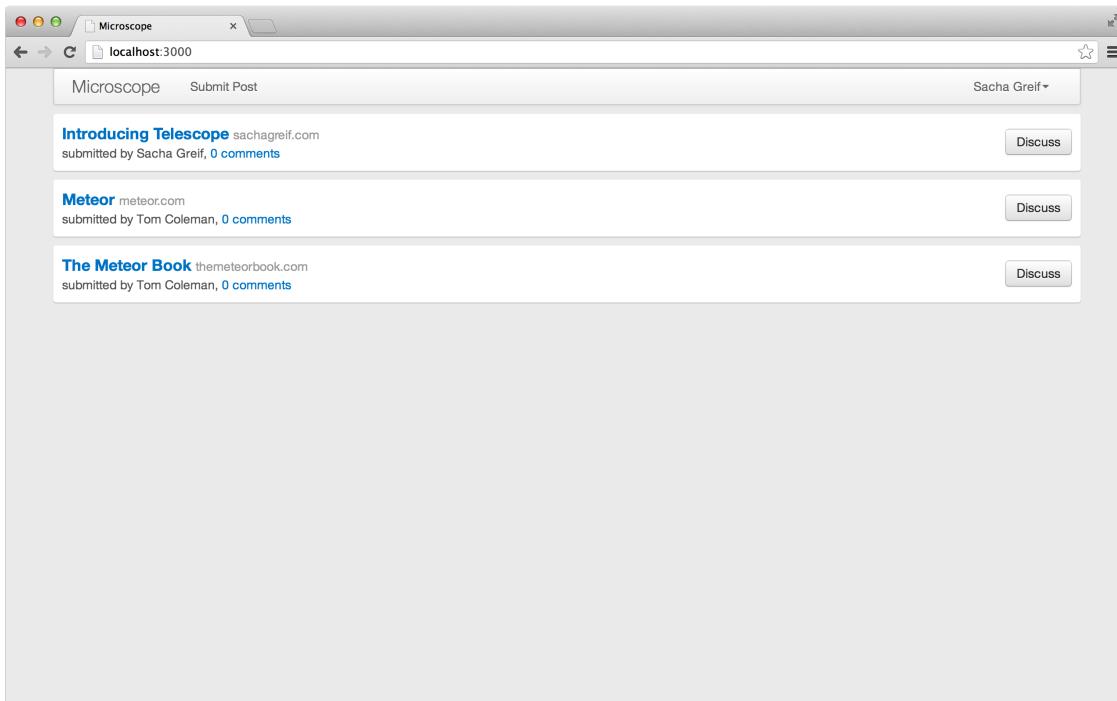
Meteor.publish('comments', function(postId) {
  check(postId, String);
  return Comments.find({postId: postId});
});
```

server/publications.js

Creado un mecanismo simple de publicación/suscripción par...

[Ver en GitHub](#)[Lanzar instancia](#)

Solo hay un problema: cuando volvemos a la página principal, todos nuestros mensajes tienen 0 comentarios:



La razón de que esto ocurra está bien clara: solo cargaremos comentarios en la ruta `postPage`, así que cuando llamamos a `Comments.find({postId: this._id})` en nuestro ayudante `commentsCount` del gestor `client/views/posts/post_item.js`, Meteor no encuentra los datos necesarios en el lado del cliente para devolver un resultado.

La mejor manera de resolver esto es *denormalizar* el número de comentarios dentro del post (si no sabes lo que significa denormalizar, no te preocupes, lo veremos en [el próximo capítulo](#)). Aunque, como veremos, hay que añadir un poco de complejidad a nuestro código, a cambio, mejoramos la velocidad al no tener que publicar todos los comentarios de la base de datos solo para contarlos.

Lo conseguiremos añadiendo una propiedad `commentsCount` a la estructura de datos del post (y restableceremos Meteor con `meteor reset` - no olvides volver a crear una cuenta de usuario):

```

// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // create two users
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachaId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sacha = Meteor.users.findOne(sachaId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sacha._id,
    author: sacha.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: new Date(now - 7 * 3600 * 1000),
    commentsCount: 2
  });

  Comments.insert({
    postId: telescopeId,
    userId: tom._id,
    author: tom.profile.name,
    submitted: new Date(now - 5 * 3600 * 1000),
    body: 'Interesting project Sacha, can I get involved?'
  });

  Comments.insert({
    postId: telescopeId,
    userId: sacha._id,
    author: sacha.profile.name,
    submitted: new Date(now - 3 * 3600 * 1000),
    body: 'You sure can Tom!'
  });

  Posts.insert({
    title: 'Meteor',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://meteor.com',
    submitted: new Date(now - 10 * 3600 * 1000),
    commentsCount: 0
  });

  Posts.insert({
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteorbook.com',
    submitted: new Date(now - 12 * 3600 * 1000),
    commentsCount: 0
  });
}

```

server/fixtures.js

Como de costumbre cuando actualizamos el fichero de fixtures, deberás ejecutar `meteor reset` para inicializar la base de datos y asegurarnos que se ejecutan de nuevo los fixtures.

Luego, nos aseguramos de que todos los nuevos posts empiezan con 0 comentarios:

```
....  
  
var post = _.extend(postAttributes, {  
  userId: user._id,  
  author: user.username,  
  submitted: new Date(),  
  commentsCount: 0  
});  
  
var postId = Posts.insert(post);  
  
....
```

lib/collections/posts.js

Y entonces actualizamos `commentsCount` cuando hacemos un nuevo comentario usando el operador `$inc` de Mongo (que incrementa campos numéricos):

```
....  
  
comment = _.extend(commentAttributes, {  
  userId: user._id,  
  author: user.username,  
  submitted: new Date()  
});  
  
// update the post with the number of comments  
Posts.update(comment.postId, {$inc: {commentsCount: 1}});  
  
return Comments.insert(comment);  
  
....
```

lib/collections/comments.js

Finalmente, tenemos que eliminar el ayudante `commentsCount` de `client/templates/posts/post_item.js`, ya que el campo está disponible directamente en el post.

Denormalizando el número de
comentarios.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora que los usuarios pueden hablar entre sí, sería una lástima que se perdieran los nuevos comentarios de otros usuarios. En el siguiente capítulo veremos cómo implementar notificaciones!

Denormalizar los datos significa no almacenar esos datos de una manera “normal”. En otras palabras, significa tener múltiples copias de la misma porción de datos.

En el capítulo anterior, denormalizamos la cantidad de comentarios dentro del objeto post para evitar tener que cargar todos los comentarios todo el tiempo. Teniendo en cuenta el modelado de datos, esto es redundante, ya que en su lugar podríamos simplemente contar el número correcto de comentarios en cualquier momento para averiguar su valor (dejando de lado las consideraciones de rendimiento).

Denormalizar a menudo significa un trabajo extra para el desarrollador. En nuestro ejemplo, cada vez que agregamos o eliminamos un comentario además tenemos que acordarnos de actualizar el post en cuestión para asegurarnos de que el campo `commentsCount` siga siendo correcto. Esto es exactamente la razón por la cual las bases de datos relacionales como MySQL desaprueban esta técnica.

De todas maneras, la técnica normal también tiene sus desventajas: sin una propiedad `commentsCount`, necesitaríamos enviar *todos* los comentarios todo el tiempo tan sólo para poder contarlos, que es lo que estábamos haciendo en un principio. Denormalizar permite evitar esto último.

Sería posible crear una publicación especial que solo envíe la cantidad de comentarios que nos interesan (por ejemplo, la cantidad de comentarios en los posts que actualmente estamos viendo, haciendo más consultas al servidor).

Pero vale la pena considerar si la complejidad de dicha publicación superaría o no las dificultades creadas al denormalizar...

Por supuesto, dichas consideraciones son específicas para cada aplicación: si estás escribiendo código donde la integridad de datos es fundamental, entonces evitar inconsistencias en los datos es de lejos más importante y de mayor prioridad que cualquier mejora de rendimiento.

Si tienes experiencia con Mongo, podrás haberte sorprendido al ver que hemos creado una segunda colección solo para los comentarios: ¿por qué no simplemente integrar los comentarios en una lista dentro del documento post?

Resulta que muchas de las herramientas que Meteor nos da, trabajan mucho mejor operando a un nivel de colección. Por ejemplo:

1. El ayudante `{}#each{}` es muy eficiente cuando itera sobre un cursor (el resultado de `collection.find()`). Pero no lo es cuando itera sobre un array de objetos dentro un documento más grande.
2. `allow` y `deny` operan a nivel de documento, por consiguiente, facilita la tarea de asegurarse que cualquier modificación de un comentario individual es correcta. Esto sería mucho más complejo si operara a nivel de post.
3. DDP opera a nivel de los atributos “top-level” de un documento. Esto significa que si `comments` fuese una

propiedad de un `post`, cada vez que un comentario fuese creado en un post, el servidor debería enviar toda la lista de comentarios actualizada para ese post a cada uno de los clientes conectados.

4. Publicaciones y suscripciones son mucho más fáciles de controlar a nivel de documentos. Por ejemplo, si quisieramos paginar comentarios en un post sería muy difícil a menos que los comentarios estuviesen en su propia colección.

Mongo sugiere incrustar documentos para reducir la cantidad de consultas necesarias para buscar los documentos. De todos modos, esto es un problema mínimo cuando se tiene en cuenta la arquitectura de Meteor: la mayor parte del tiempo estamos consultando comentarios en el `cliente`, donde el acceso a la base de datos prácticamente no tiene coste.

Hay buenos argumentos sobre por qué *no deberías* denormalizar tus datos. Para ver un buen caso contra la denormalización, recomendamos [Por qué nunca deberías usar MongoDB](#) por Sarah Mei.

Ahora que los usuarios pueden comentar los posts de otros usuarios, sería bueno hacerles saber que alguien ha comenzado una conversación.

Para ello, notificaremos al autor, de que ha habido un comentario en su post, y le proporcionaremos un enlace para poder comentar.

En este tipo de funcionalidad es en la que Meteor brilla. Como por defecto, Meteor trabaja en tiempo real, vamos a poder mostrar notificaciones instantáneamente. No necesitamos esperar a que el usuario actualice la página, podemos mostrar nuevas notificaciones sin tener que escribir ningún código especial.

Crearemos una notificación cuando alguien comente uno de nuestros posts. En el futuro, las notificaciones podrían extenderse para cubrir muchos otros escenarios, pero, por ahora será suficiente con esto para mantener a los usuarios informados sobre lo que está pasando.

Vamos a crear la colección `Notifications` y la función `createCommentNotification` que insertará una notificación para cada comentario que se haga en uno de nuestros posts.

Puesto que estamos actualizando las notificaciones desde el lado del cliente, necesitamos asegurarnos que nuestra llamada `allow` es a prueba de balas. Por lo que deberemos comprobar que:

- El usuario que hace la llamada `update` es el dueño de la notificación modificada.
- El usuario solo está intentando modificar un solo campo.
- El campo a modificar es la propiedad `read` de nuestra notificación.

```
Notifications = new Mongo.Collection('notifications');

Notifications.allow({
  update: function(userId, doc, fieldNames) {
    return ownsDocument(userId, doc) &&
      fieldNames.length === 1 && fieldNames[0] === 'read';
  }
});

createCommentNotification = function(comment) {
  var post = Posts.findOne(comment.postId);
  if (comment.userId !== post.userId) {
    Notifications.insert({
      userId: post.userId,
      postId: post._id,
      commentId: comment._id,
      commenterName: comment.author,
      read: false
    });
  }
};
```

lib/collections/notifications.js

Al igual que con los posts o los comentarios, esta colección estará compartida por clientes y servidor. Como tendremos que actualizar las notificaciones cuando un usuario las haya visto, permitimos hacer `update` siempre que se trate de los datos del propio usuario.

También creamos una función que mira qué post está comentando el usuario, averigua qué usuario debe ser notificado e inserta una nueva notificación.

Ya tenemos un método en el servidor para crear comentarios, por lo que podemos ampliarlo para que llame a nuestra nueva función. Para guardar el `_id` del nuevo comentario en una variable, cambiamos `return Comments.insert(comment);`, por `comment._id = Comments.insert(comment)` y llamamos a la función `createCommentNotification`:

```
Comments = new Mongo.Collection('comments');

Meteor.methods({
  commentInsert: function(commentAttributes) {
    //...

    comment = _.extend(commentAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    // update the post with the number of comments
    Posts.update(comment.postId, {$inc: {commentsCount: 1}});

    // create the comment, save the id
    comment._id = Comments.insert(comment);

    // now create a notification, informing the user that there's been a comment
    createCommentNotification(comment);

    return comment._id;
  }
});
```

lib/collections/comments.js

Tenemos que publicar las notificaciones:

```
Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function(postId) {
  check(postId, String);
  return Comments.find({postId: postId});
});

Meteor.publish('notifications', function() {
  return Notifications.find();
});
```

server/publications.js

Y suscribirnos en el cliente:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('posts'), Meteor.subscribe('notifications')]
  }
});
```

lib/router.js

Añadida la colección de
comentarios.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora podemos seguir y añadir una lista de notificaciones a nuestra cabecera:

```
<template name="header">
<nav class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="{{pathFor 'postsList'}}>Microscope</a>
  </div>
  <div class="collapse navbar-collapse" id="navigation">
    <ul class="nav navbar-nav">
      {{#if currentUser}}
        <li>
          <a href="{{pathFor 'postSubmit'}}>Submit Post</a>
        </li>
        <li class="dropdown">
          {{> notifications}}
        </li>
      {{/if}}
    </ul>
    <ul class="nav navbar-nav navbar-right">
      {{> loginButtons}}
    </ul>
  </div>
</nav>
</template>
```

client/templates/includes/header.html

Y crear las plantillas `notifications` y `notificationItem` (que pondremos en el archivo `notifications.html`):

```

<template name="notifications">
  <a href="#" class="dropdown-toggle" data-toggle="dropdown">
    Notifications
    {{#if notificationCount}}
      <span class="badge badge-inverse">{{notificationCount}}</span>
    {{/if}}
    <b class="caret"></b>
  </a>
  <ul class="notification dropdown-menu">
    {{#if notificationCount}}
      {{#each notifications}}
        {{> notificationItem}}
      {{/each}}
    {{else}}
      <li><span>No Notifications</span></li>
    {{/if}}
  </ul>
</template>

<template name="notificationItem">
  <li>
    <a href="{{notificationPostPath}}">
      <strong>{{commenterName}}</strong> commented on your post
    </a>
  </li>
</template>

```

client/templates/notifications/notifications.html

Podemos ver que para cada notificación, tendremos un enlace al post que ha sido comentado junto con el usuario que lo ha hecho.

A continuación, hay que asegurarse de que se selecciona la lista de notificaciones correcta desde nuestro ayudante, y actualizar las notificaciones como “leídas” cuando el usuario hace clic en el enlace al que apuntan.

```

Template.notifications.helpers({
  notifications: function() {
    return Notifications.find({userId: Meteor.userId(), read: false});
  },
  notificationCount: function(){
    return Notifications.find({userId: Meteor.userId(), read: false}).count();
  }
});

Template.notificationItem.helpers({
  notificationPostPath: function() {
    return Router.routes.postPage.path({_id: this.postId});
  }
});

Template.notificationItem.events({
  'click a': function() {
    Notifications.update(this._id, {$set: {read: true}});
  }
});

```

client/templates/notifications/notifications.js

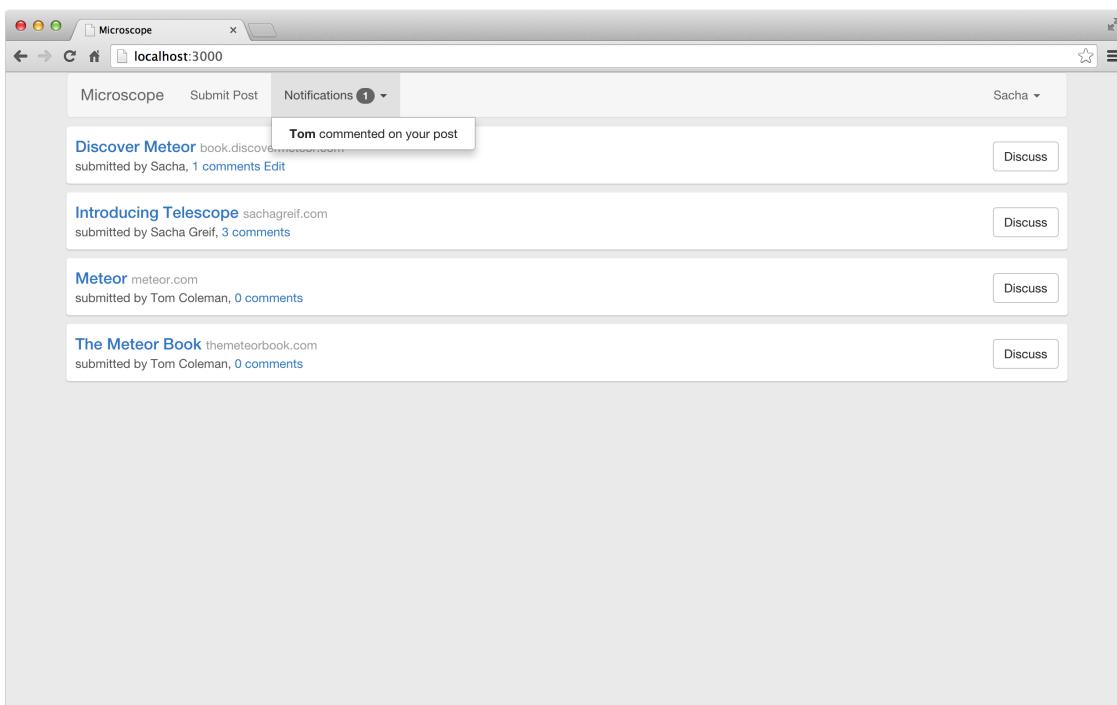
Mostrar las notificaciones en la cabecera.

[Ver en GitHub](#)

[Lanzar instancia](#)

Como podemos ver, las notificaciones no son muy diferentes de los errores, y su estructura es muy similar. Solo hay una diferencia clave: hemos creado una colección sincronizada cliente-servidor. Esto significa que nuestras notificaciones son persistentes y, siempre y cuando se utilice la misma cuenta de usuario, persistirán en distintos navegadores y dispositivos.

Abre un segundo navegador, crea una nueva cuenta de usuario, y añade un comentario en un post del usuario anterior. Deberías ver algo así:



Las notificaciones van bien. Sin embargo, hay un pequeño problema: nuestras notificaciones son públicas.

Si ejecutamos el siguiente comando en la consola del segundo navegador:

```
Notifications.find().count();  
1
```

Consola del navegador

El nuevo usuario (el que ha *comentado*) no debería tener notificaciones. Las que vemos son las de los demás usuarios.

A parte de los posibles problemas de privacidad, simplemente no podemos permitirnos el lujo de cargar las notificaciones de todos los usuarios. Con un sitio lo suficientemente grande, esto podría sobrecargar la memoria disponible en el navegador y empezar a causar graves problemas de rendimiento.

Resolveremos este problema mediante las **publicaciones**. Podemos usar las publicaciones para especificar qué parte de nuestra colección queremos compartir con el navegador.

Para lograrlo, tenemos que cambiar `Notifications.find()`. Es decir, tenemos que devolver el cursor que corresponde a las notificaciones del usuario actual.

Hacer esto es bastante sencillo puesto que la función `publish` tiene el `_id` del usuario actual disponible en `this.userId`:

```
Meteor.publish('notifications', function() {
  return Notifications.find({userId: this.userId, read: false});
});
```

server/publications.js

Sincronizar solo las notificaciones relevantes al usuario.

[Ver en GitHub](#)

[Lanzar instancia](#)

Si ahora se busca en las consolas de los dos navegadores, deberíamos ver dos colecciones distintas de notificaciones:

```
> Notifications.find().count();
1
```

Consola del navegador (usuario 1)

```
> Notifications.find().count();
0
```

Consola del navegador (usuario 2)

De hecho, la lista de notificaciones cambiará si accedes y sales de la aplicación. Esto se debe a que las publicaciones se repiten automáticamente cada vez que cambia el estado del usuario.

Nuestra aplicación es cada vez más funcional, y a medida que cada vez más usuarios entran y empiezan a publicar enlaces, corremos el riesgo de acabar con una página de inicio demasiado larga. Vamos a abordar este problema en el próximo capítulo: la paginación.

No es común tener que escribir código de seguimiento de dependencias por ti mismo, pero para comprender el concepto, es verdaderamente útil seguir el camino de cómo funciona el flujo de dependencias.

Imagina que quisiéramos saber a cuántos amigos del usuario actual de Facebook le ha gustado cada post en Microscope. Supongamos que ya hemos trabajado en los detalles de cómo autenticar el usuario con Facebook, hacer las llamadas necesarias a la API, y procesar los datos relevantes. Ahora tenemos una función asíncrona en el lado del cliente que devuelve el número de “me gusta”: `getFacebookLikeCount(user, url, callback)`.

Lo importante a recordar sobre una función de esta naturaleza es que *no es reactiva* ni funciona en tiempo real. Hará una petición HTTP a Facebook, enviando algunos datos, y obtendremos el resultado en la aplicación a través de una llamada asíncrona. Pero la función no se va a volver a ejecutar por sí sola cuando haya un cambio en Facebook, ni nuestra UI va a cambiar cuando los datos lo hagan.

Para solucionar esto, podemos comenzar utilizando `setInterval` para llamar a la función cada ciertos segundos:

```
currentLikeCount = 0;
Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId).url,
      function(err, count) {
        if (!err)
          currentLikeCount = count;
      });
  }
}, 5 * 1000);
```

Cada vez que usemos esa variable `currentLikeCount`, obtendremos el número correcto con un margen de error de cinco segundos. Ahora podemos usar esa variable en un ayudante:

```
Template.postItem.likeCount = function() {
  return currentLikeCount;
}
```

Sin embargo, nada le dice todavía a nuestra plantilla que se redibuje cuando cambie `currentLikeCount`. Si bien la variable ahora está en pseudo tiempo real (se cambia a sí misma), no es *reactiva* y por lo tanto todavía no puede comunicarse correctamente con el resto del ecosistema de Meteor.

La reactividad de Meteor es mediada por *dependencias*, estructuras de datos que rastrean una serie de computaciones.

Como vimos anteriormente, una computación es una parte de código que usa datos reactivos. En nuestro caso, hay una computación que ha sido implícitamente creada por la plantilla `postItem`, y cada ayudante en el manejador de esa plantilla está funcionando dentro de esa computación.

Puedes pensar de la computación como una parte del código que se “preocupa” por los datos reactivos. Cuando los datos cambien, esta computación será informada (a través de `invalidate()`), y es la computación la que debe decidir si algo debe hacerse.

Para transformar nuestra variable `currentLikeCount` en una fuente de datos reactiva, necesitamos rastrear todas las computaciones que la usan como una dependencia. Esto requiere trasformarla de una variable a una función (que devolverá un valor):

```
var _currentLikeCount = 0;
var _currentLikeCountListeners = new Tracker.Dependency();

currentLikeCount = function() {
  _currentLikeCountListeners.depend();
  return _currentLikeCount;
}

Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId),
      function(err, count) {
        if (!err && count !== _currentLikeCount) {
          _currentLikeCount = count;
          _currentLikeCountListeners.changed();
        }
      });
  }
}, 5 * 1000);
```

Lo que hemos hecho es configurar una dependencia `_currentLikeCountListeners`, que rastreará todas las computaciones en las cuales se utilice `currentLikeCount()`. Cuando el valor de `_currentLikeCount` cambie, podemos llamar a la función `changed()` en esa dependencia, que invalida todas las computaciones realizadas.

Estas computaciones pueden entonces seguir adelante y evaluar los cambios caso por caso.

Parece un montón de código para una única fuente de datos reactiva, y tienes razón, por lo que Meteor proporciona algunas herramientas de serie para hacerlo un poco más sencillo. (normalmente, no se usan computaciones directamente, si no sencillamente auto ejecuciones). Hay un paquete llamado `reactive-var` que hace exactamente lo que hacemos con la función `currentLikeCount()`. Por lo que si lo añadimos:

```
meteor add reactive-var
```

Podremos simplificar nuestro código un poco:

```
var currentLikeCount = new ReactiveVar();

Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId),
      function(err, count) {
        if (!err) {
          currentLikeCount.set(count);
        }
      });
  }
}, 5 * 1000);
```

Ahora para usarlo, llamaremos a `currentLikeCount.get()` en nuestro ayudante y debería funcionar como antes. Hay también otro paquete `reactive-dict`, que proporciona un almacén de datos clave-valor reactivo (exactamente igual que la `Session`), que podría ser útil también.

Angular es una librería de renderizado reactivo del lado del cliente, desarrollada por la buena gente de Google. Solo se puede comparar el seguimiento de dependencias de Meteor con el de Angular de un modo ilustrativo, ya que sus enfoques son bastante diferentes.

Hemos visto que el modelo de Meteor usa bloques de código llamados computaciones. Estas son seguidas por fuentes de datos “reactivas” (funciones) que se ocupan de invalidarlos cuando sea apropiado. Así, la fuente de datos informa *explícitamente* todas sus dependencias cuando necesita llamar a `invalidate()`. Nótese que a pesar de que esto sucede generalmente cuando los datos cambian, la fuente de los datos puede además decidir ejecutar la invalidación por otras razones.

Además, por más que usualmente las computaciones simplemente se reejecutan cuando son invalidadas, se pueden configurar para que se comporten como uno quiera. Esto nos da un alto nivel de control sobre la reactividad.

En Angular, la reactividad es medida por el objeto `scope`. Un scope, o alcance, puede ser pensado como un simple objeto de JavaScript con algunos métodos especiales.

Cuando se desea depender reactivamente de un valor dentro del scope, se llama a `scope.$watch`, declarando la expresión en la que uno está interesado (por ejemplo, qué partes del scope te importan) y una función que se ejecutará cada vez que esa expresión cambie. Así, se puede declarar explícitamente qué hacer cada vez que ese valor sea modificado.

Volviendo a nuestro ejemplo con Facebook, escribiríamos:

```
$rootScope.$watch('currentLikeCount', function(likeCount) {
  console.log('Current like count is ' + likeCount);
});
```

Por supuesto, es tan raro tener que configurar computaciones en Meteor, como tener que invocar a `$watch` explícitamente en Angular, ya que las directivas `ng-model` y las `{}{{expressions}}` automáticamente se ocupan de re-renderizarse cuando haya un cambio.

Cuando dicho valor reactivo sea cambiado, `scope.$apply()` debe ser llamado. Esto vuelve a evaluar cada “watcher” del scope, pero solo llama a las funciones de aquellos “watchers” que contengan valores modificados.

Entonces, `scope.$apply()` es similar a `dependency.changed()`, excepto que actúa al nivel del scope, en lugar de darle el control al desarrollador para decirle precisamente cuáles funciones deberían ser re-evaluadas. Con eso aclarado, esta pequeña falta de control le da a Angular la habilidad de ser muy inteligente y eficiente, ya que determina precisamente qué debe ser vuelto a evaluar.

Con Angular, nuestra función `getFacebookLikeCount()` habría sido algo así:

```
Meteor.setInterval(function() {
  getFacebookLikeCount(Meteor.user(), Posts.find(postId),
    function(err, count) {
      if (!err) {
        $rootScope.currentLikeCount = count;
        $rootScope.$apply();
      }
    });
}, 5 * 1000);
```

Decididamente, Meteor se ocupa de la parte más pesada por nosotros y nos deja beneficiarnos de la reactividad sin demasiado trabajo. Pero tal vez, aprender estos patrones será de ayuda si alguna vez necesitas ir más allá.

Nuestra aplicación va tomando forma y podemos esperar un gran éxito cuando todo el mundo la conozca.

Así que quizás debamos pensar un poco sobre cómo afectará al rendimiento el gran número de nuevos posts que vamos a recibir.

Hemos visto antes cómo una colección en el cliente puede contener un subconjunto de los datos en el servidor y lo hemos usado para nuestras notificaciones y comentarios.

Ahora pensemos en que todavía estamos publicando todos nuestros posts de una sola vez a todos los usuarios conectados. Si se publicaran miles de enlaces, esto sería un problema. Para solucionarlo tenemos que paginar nuestros posts.

En primer lugar, vamos a cargar los suficientes posts para que la paginación tenga sentido:

```
// Fixture data
if (Posts.find().count() === 0) {

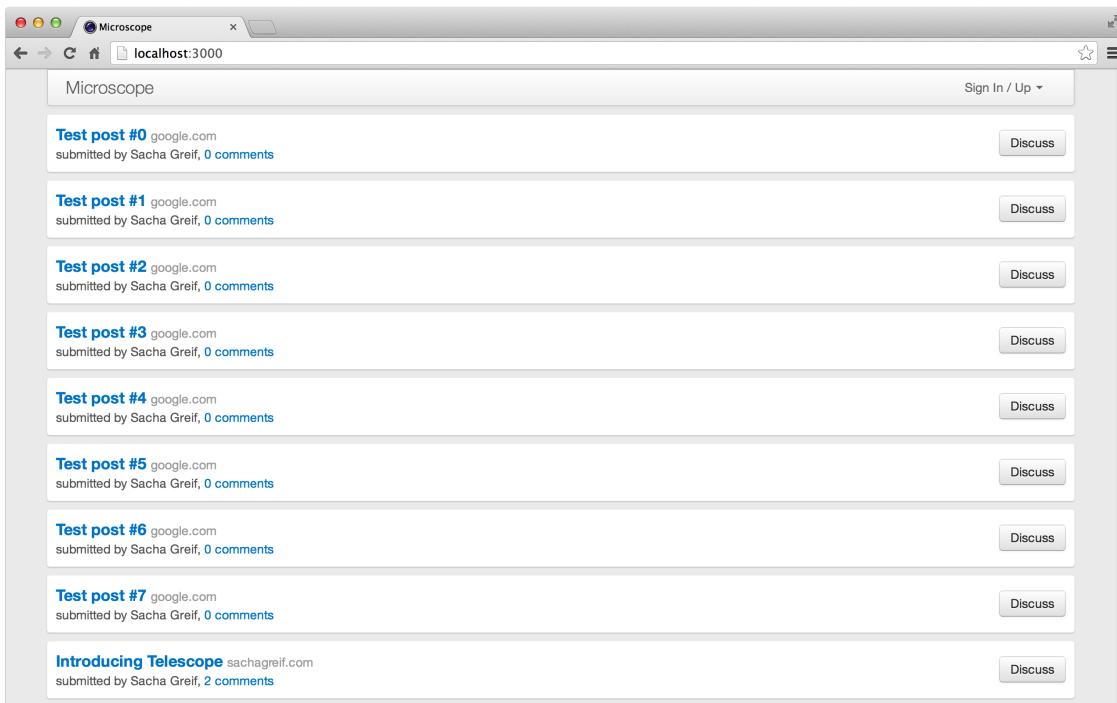
    //...

    Posts.insert({
        title: 'The Meteor Book',
        userId: tom._id,
        author: tom.profile.name,
        url: 'http://themeteorbook.com',
        submitted: new Date(now - 12 * 3600 * 1000),
        commentsCount: 0
    });

    for (var i = 0; i < 10; i++) {
        Posts.insert({
            title: 'Test post #' + i,
            author: sacha.profile.name,
            userId: sacha._id,
            url: 'http://google.com/?q=test-' + i,
            submitted: new Date(now - i * 3600 * 1000),
            commentsCount: 0
        });
    }
}
```

server/fixtures.js

Después de ejecutar `meteor reset` e iniciar la aplicación de nuevo, deberíamos ver algo como esto:



Añadidos suficientes posts para hacer necesaria la pagina...

[Ver en GitHub](#)

[Lanzar instancia](#)

Vamos a implementar una paginación de estilo “infinito”. Lo que queremos decir con esto es que primero mostramos, por ejemplo, 10 posts, con un enlace de “cargar más” en la parte inferior. Al hacer clic en este enlace se cargarán 10 más, y así *hasta el infinito y más allá*. Esto significa que podemos controlar todo nuestro sistema de paginación con un solo parámetro que representa el número de posts que mostraremos en pantalla.

Vamos a necesitar entonces una forma de pasar este parámetro al servidor para que sepa la cantidad de mensajes que debe enviar al cliente. Se da la circunstancia de que ya estamos suscritos a la publicación `posts` en el router, así que vamos a aprovecharlo y a dejar que el router maneje también la paginación.

La forma más fácil de configurar esto es hacer que el parámetro límite forme parte de la ruta, quedando las URL de la forma `http://localhost:3000/25`. Una ventaja añadida de utilizar la URL en vez de otros métodos es que si estamos viendo 25 posts y resulta que se recarga la página por error, todavía seguiremos viendo 25 posts.

Para hacer esto correctamente, tenemos que cambiar la forma en que nos suscribimos a los posts. Al igual que hicimos en el capítulo *Comentarios*, tendremos que mover nuestro código de suscripción desde el nivel de router al nivel de ruta.

Parece demasiado para hacerlo todo de una sola vez, pero se verá más claro escribiendo el código.

En primer lugar, vamos a dejar de suscribirnos a la publicación `posts` en el bloque `Router.configure()`. Simplemente

elimina `Meteor.subscribe('posts')`, dejando solo la suscripción `notifications`:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('notifications')]
  }
});
```

lib/router.js

A continuación, añadiremos el parámetro `postsLimit` al path de la ruta. Si añadimos un `?` después del parámetro, lo hacemos opcional. De forma que nuestra ruta no solo coincidirá con `http://localhost:3000/50`, sino también con `http://localhost:3000`.

```
//...
Router.route('/postsLimit?', {
  name: 'postsList',
});
```

//...

lib/router.js

Es importante señalar que un path de la forma `/:parameter?` coincide con todos los path posibles. Dado que cada ruta se analiza en orden secuencial para comprobar si coincide con la ruta actual, tenemos que asegurarnos que organizamos bien nuestras rutas con el fin de disminuir la especificidad.

En otras palabras, las rutas más específicas como `/posts/:_id` deben ir primero, y nuestra ruta `postsList` debería ir **en la parte inferior** del grupo de rutas para que todo coincida correctamente.

Es el momento de abordar el difícil problema de suscribirse y encontrar los datos correctos. Tenemos que lidiar con el caso en el que el parámetro `postsLimit` no está presente, por lo que vamos a asignarle un valor predeterminado. Usaremos “5”, que nos dará suficiente espacio para jugar con la paginación.

```
//...
Router.route('/postsLimit?', {
  name: 'postsList',
  waitOn: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
  }
});
```

//...

lib/router.js

Ya te habrás dado cuenta de que estamos pasando un objeto JavaScript (`{sort: {submitted: -1}, limit: postsLimit}`) junto

con el nombre de nuestra publicación `posts`. Este objeto servirá como parámetro `options` en la llamada a `Posts.find()` en el lado del servidor. Vamos a cambiar nuestro código en el servidor para implementarlo:

```
Meteor.publish('posts', function(options) {
  check(options, {
    sort: Object,
    limit: Number
  });
  return Posts.find({}, options);
});

Meteor.publish('comments', function(postId) {
  check(postId, String);
  return Comments.find({postId: postId});
});

Meteor.publish('notifications', function() {
  return Notifications.find({userId: this.userId});
});
```

server/publications.js

Nuestro código de publicaciones está diciendo al servidor que puede confiar en cualquier objeto JavaScript enviado por el cliente (en nuestro caso, `{limit: postsLimit}`) para servir como opciones para `find()`. Esto hace posible que los usuarios envíen cualquier opción a través de la consola del navegador.

En nuestro caso, esto es relativamente inofensivo, ya que todo lo que un usuario podría hacer es reordenar los mensajes de manera diferente, o cambiar el límite (que es lo que queremos hacer). ¡De todas formas una aplicación del mundo real debería probablemente limitar el límite!

Afortunadamente, usando `check()` sabemos que los usuarios no podrán inyectar opciones adicionales (como la opción `fields`, que en algunos casos podría exponer datos privados en los documentos).

De todas formas, un patrón más seguro para asegurarnos el control de nuestros datos podría ser pasar los parámetros de forma individual en lugar de todo el objeto:

```
Meteor.publish('posts', function(sort, limit) {
  return Posts.find({}, {sort: sort, limit: limit});
});
```

Ahora que nos suscribimos a nivel de ruta, tiene sentido establecer el contexto de datos en ese mismo lugar. Vamos a desviarnos un poco de nuestro patrón anterior y hacer que la función `data` devuelva un objeto JavaScript en lugar de simplemente devolver un cursor. Esto nos permite crear un contexto de datos con nombre que llamaremos `posts`.

Lo que significa es que en lugar de disponer implícitamente de los datos en `this` dentro de la plantilla, estará disponible también como `posts`. Aparte de este pequeño elemento, el código debe sernos familiar:

```
//...

Router.route('/postsLimit?', {
  name: 'postsList',
  waitOn: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
  },
  data: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return {
      posts: Posts.find({}, {sort: {submitted: -1}, limit: limit})
    };
  }
});

//...
```

lib/router.js

Ahora que hemos establecido el contexto de datos a nivel de router podemos deshacernos del ayudante de plantilla `posts` del archivo `posts_list.js` borrando el contenido de este archivo.

Y como hemos llamado `posts` al contexto de datos (igual que en el ayudante), ¡ni siquiera necesitamos tocar la plantilla `postsList`!

Recapitulemos. Así es como ha quedado nuestro `router.js`:

```

Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('notifications')]
  }
});

Router.route('/posts/:_id', {
  name: 'postPage',
  waitOn: function() {
    return Meteor.subscribe('comments', this.params._id);
  },
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/posts/:_id/edit', {
  name: 'postEdit',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

Router.route('/:postsLimit?', {
  name: 'postsList',
  waitOn: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
  },
  data: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return {
      posts: Posts.find({}, {sort: {submitted: -1}, limit: limit})
    };
  }
});

var requireLogin = function() {
  if (!Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});

```

lib/router.js

Aumentada la ruta `postsList` para que tenga un límite.

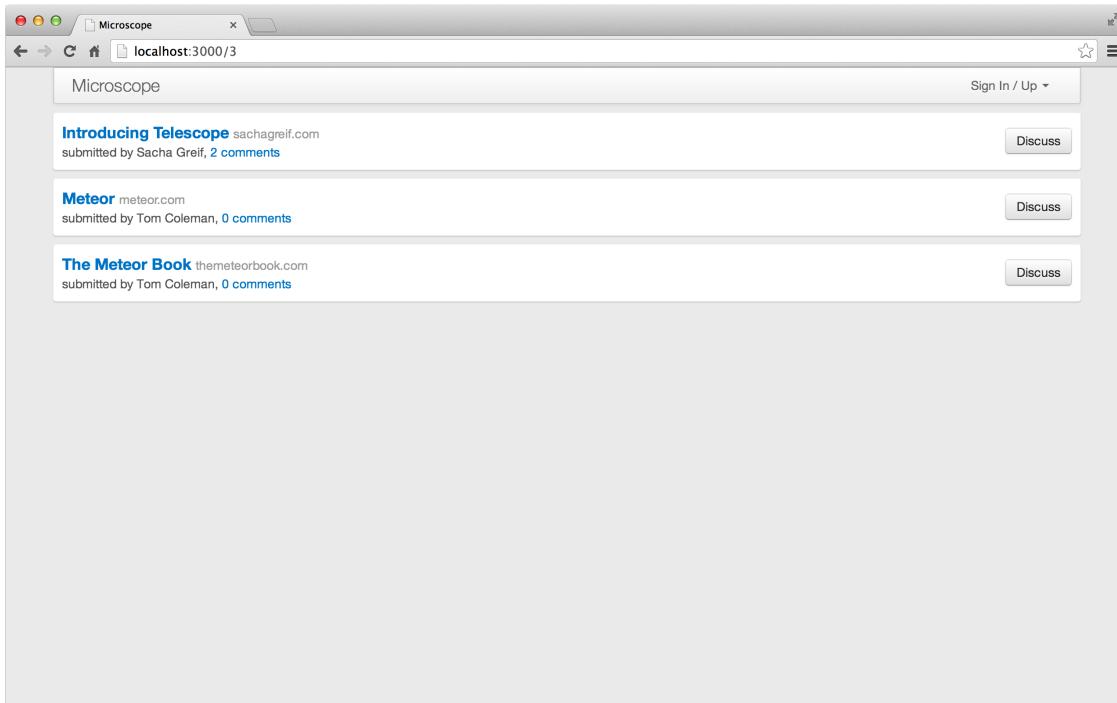
[Ver en GitHub](#)

[Lanzar instancia](#)

Vamos a probar nuestro nuevo sistema de paginación. Ahora podemos mostrar un número arbitrario de posts en la

página principal simplemente cambiando el parámetro en la URL. Por ejemplo, intenta acceder a

<http://localhost:3000/3> . Deberías ver algo como esto:



¿Por qué usamos el enfoque “paginación infinita” en lugar de mostrar páginas sucesivas con 10 posts cada uno, como hace Google para en sus resultados de búsqueda? En realidad se debe al paradigma de tiempo real que utiliza Meteor.

Imaginemos que paginamos nuestra colección `Posts` utilizando el patrón de resultados Google, y que estamos en la página 2, que muestra los mensajes de 10 a 20. ¿Qué pasa si otro usuario elimina una de las 10 entradas anteriores?

Como nuestra aplicación es en tiempo real, nuestra base de datos cambiaría. El post 10 se convertiría en el 9, y desaparecería de nuestra vista y ahora veríamos el 11. El resultado sería que el usuario vería aparecer y desaparecer posts sin razón aparente.

Incluso si toleramos esta peculiaridad, la paginación tradicional también es difícil de implementar por razones técnicas.

Volvamos a nuestro ejemplo anterior. Estamos publicando los posts 10-20 de la colección `Posts`, pero ¿cómo los encontramos en el cliente? No se pueden seleccionar los mensajes 10 a 20 porque solo hay diez posts en total en el conjunto de datos del lado del cliente.

Una solución sería publicar esos 10 mensajes en el servidor y, a continuación, hacer un `Posts.find()` en el lado del cliente para recoger todos los posts publicados.

Esto funciona si solo tienes una suscripción. Pero, ¿y si tenemos más de una?

Digamos que una suscripción pide los posts 10 a 20, y otra 30 a 40. Ahora tenemos 20 mensajes cargados del lado del cliente en total, y no hay forma de saber cuáles pertenecen a cada suscripción.

Por todas estas razones, la paginación tradicional simplemente no tiene mucho sentido cuando se trabaja con Meteor.

Te habrás dado cuenta de que repetimos dos veces la línea `var limit = parseInt(this.params.postsLimit) || 5;`. Además, codificar el número “5” no es lo ideal. No te preocupes, no es el fin del mundo, pero como siempre es mejor seguir el principio DRY (Don’t Repeat Yourself), vamos a ver cómo podemos refactorizar un poco las cosas.

Introduciremos un nuevo aspecto de Iron Router, los *controladores de ruta*. Un controlador de ruta es simplemente una forma de agrupar en un paquete reutilizable, características de enrutamiento que puede heredar cualquier ruta. Ahora solo lo utilizaremos para una sola ruta, pero en el próximo capítulo veremos que esta característica es muy útil.

```

//...

PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.postsLimit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  data: function() {
    return {posts: Posts.find({}, this.findOptions())};
  }
});

//...

Router.route('/:postsLimit?', {
  name: 'postsList'
});

//...

```

lib/router.js

Vamos a verlo paso a paso. En primer lugar, creamos nuestro controlador extendiendo `RouteController`. A continuación, establecemos la propiedad `template` tal y como hicimos antes, y luego una nueva propiedad `increment`.

A continuación, definimos una nueva función `postsLimit` que devolverá el límite actual, y una función `findOptions` que devolverá un objeto con las opciones. Esto puede parecer un paso extra, pero vamos a hacer uso de él en el futuro.

A continuación, definimos nuestras funciones `waitOn` y de `data` igual que antes, excepto que ahora usan la nueva función `findOptions`.

Como nuestro controlador se llama `PostsListController` y nuestra ruta se llama `postsList`, Iron Router usará el controlador automáticamente. Así que sólo necesitamos eliminar `waitOn` y `data` de la definición de la ruta (ya que es el controlador quien los gestiona ahora). Si necesitáramos utilizar un controlador con un nombre diferente, lo podemos hacer usando la opción `controller` (veremos un ejemplo de esto en el siguiente capítulo).

postLists refactorizado en un controlador de rutas

[Ver en GitHub](#)

[Lanzar instancia](#)

Ya tenemos funcionando la paginación. Solo hay un problema: no hay manera de utilizarla realmente si no escribimos en la URL manualmente. Desde luego, no parece una gran experiencia de usuario, así que vamos a arreglarlo.

Lo que queremos hacer es bastante simple. Vamos a añadir un botón “Load more” en la parte inferior de nuestra lista de

posts, que incrementará en 5 el número de posts que se muestran. Así que si estamos en la URL `http://localhost:3000/5`, haciendo clic en “Load more” debería llevarnos a `http://localhost:3000/10`. Si has llegado hasta aquí, ¡confiamos en que podrás manejar un poco de aritmética!

Al igual que antes, vamos a añadir nuestra lógica de paginación en la ruta. ¿Recuerdas que nombramos explícitamente el contexto de datos en lugar de usar un cursor anónimo? Bueno, pues no hay ninguna regla que diga que la función `data` solo puede pasar cursores, de modo que usaremos la misma técnica para generar la URL del botón “Load more”.

```
//...

PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.postsLimit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.postsLimit();
    var nextPath = this.route.path({postsLimit: this.postsLimit() + this.increment});
    return {
      posts: this.posts(),
      nextPath: hasMore ? nextPath : null
    };
  }
});

//...
```

lib/router.js

Echemos un vistazo en profundidad a este pequeño truco de magia que hemos puesto en el router. Recuerda que la ruta `postsList` (que se hereda del controlador `PostsListController` con el que estamos trabajando) toma un parámetro `postsLimit`.

Cuando alimentamos `{postsLimit: this.postsLimit() + this.increment}` a `this.route.path()`, le estamos diciendo a la ruta `postsList` que construya su propio path utilizando ese objeto JavaScript como contexto de datos.

En otras palabras, es exactamente lo mismo que usar el ayudante Spacebars `{{pathFor 'postsList'}}` , salvo que reemplazamos el `this` implícito por nuestro propio contexto de datos a medida.

Estamos cogiendo ese path y añadiéndolo al contexto de datos de nuestra plantilla, pero sólo si hay más posts que mostrar. La forma de hacerlo es un poco complicada.

Sabemos que `this.limit()` devuelve el número actual de posts que nos gustaría mostrar, que puede ser el valor de la URL actual, o el valor por defecto (5) si la URL no contiene ningún parámetro.

Por otro lado, `this.posts` se refiere al cursor actual, de modo que `this.posts.count()` es el número de mensajes que

hay en el cursor.

Así que lo que estamos diciendo es que si pedimos `n` posts y obtenemos `n`, seguiremos mostrando el botón “Load more”. Pero si pedimos `n` y tenemos menos de `n`, significa que hemos llegado al límite y deberíamos dejar de mostrarlo.

Con todo esto, nuestro sistema falla en un caso: cuando el número de posts en nuestra base de datos es *exactamente* `n`. Si eso ocurre, el cliente pedirá `n` posts y obtendrá `n` por lo que seguirá mostrando el botón “Load more”, sin darse cuenta de que ya no quedan más elementos.

Lamentablemente, no hay soluciones sencillas para este problema, así que por ahora vamos a tener que conformarnos con esto.

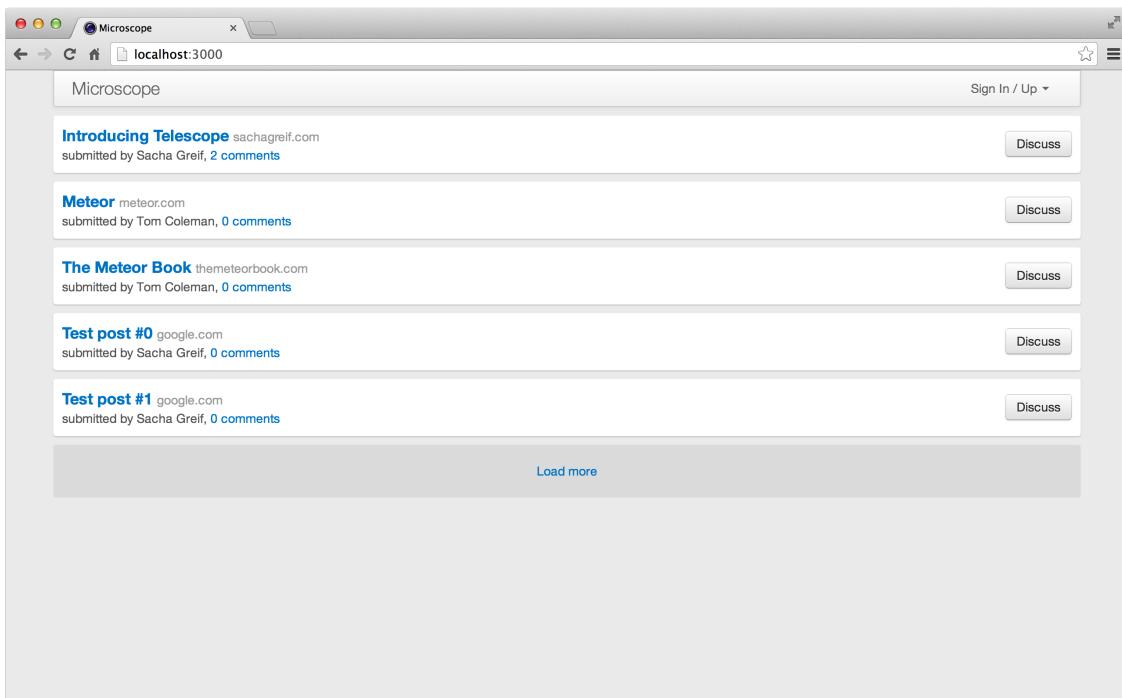
Todo lo que queda por hacer es añadir el botón “Load more” en la parte inferior de nuestra lista de posts, asegurándonos de mostrarlo solo si tenemos más posts que cargar:

```
<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}>Load more</a>
    {{/if}}
  </div>
</template>
```

client/templates/posts/posts_list.html

Así es como se debería ver la lista ahora:



Añadido nextPath() al controlador para desplazarnos por
l...

[Ver en GitHub](#)

[Lanzar instancia](#)

La paginación funciona correctamente, pero tiene una peculiaridad algo molesta: cada vez que se hace clic en “Load more” y el router pide más posts, nos envía a la plantilla de la carga mientras esperamos los nuevos datos. El resultado es que cada vez, nos envía a la parte superior de la página y tenemos que desplazarnos hasta el final para reanudar la navegación.

Así que primero, tenemos que decirle a Iron Router que no espere (`waitOn`) la suscripción. En su lugar, definiremos nuestras suscripciones en el hook `subscriptions`.

También estamos pasando una variable `ready` que hace referencia a `this.postsSub.ready` como parte de nuestro contexto de datos, que informará a la plantilla cuando se haya terminado de cargar la suscripción a los posts.

```
//...  
  
PostsListController = RouteController.extend({  
  template: 'postsList',  
  increment: 5,  
  postsLimit: function() {  
    return parseInt(this.params.postsLimit) || this.increment;  
  },  
  findOptions: function() {  
    return {sort: {submitted: -1}, limit: this.postsLimit()};  
  },  
  subscriptions: function() {  
    this.postsSub = Meteor.subscribe('posts', this.findOptions());  
  },  
  posts: function() {  
    return Posts.find({}, this.findOptions());  
  },  
  data: function() {  
    var hasMore = this.posts().count() === this.postsLimit();  
    var nextPath = this.route.path({postsLimit: this.postsLimit() + this.increment});  
    return {  
      posts: this.posts(),  
      ready: this.postsSub.ready,  
      nextPath: hasMore ? nextPath : null  
    };  
  }  
});  
  
//...
```

lib/router.js

Comprobaremos esta variable `ready` en la plantilla para mostrar un spinner al final de la lista de posts mientras estemos cargando el nuevo conjunto de posts:

```

<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}>Load more</a>
    {{else}}
      {{#unless ready}}
        {{> spinner}}
      {{/unless}}
    {{/if}}
  </div>
</template>

```

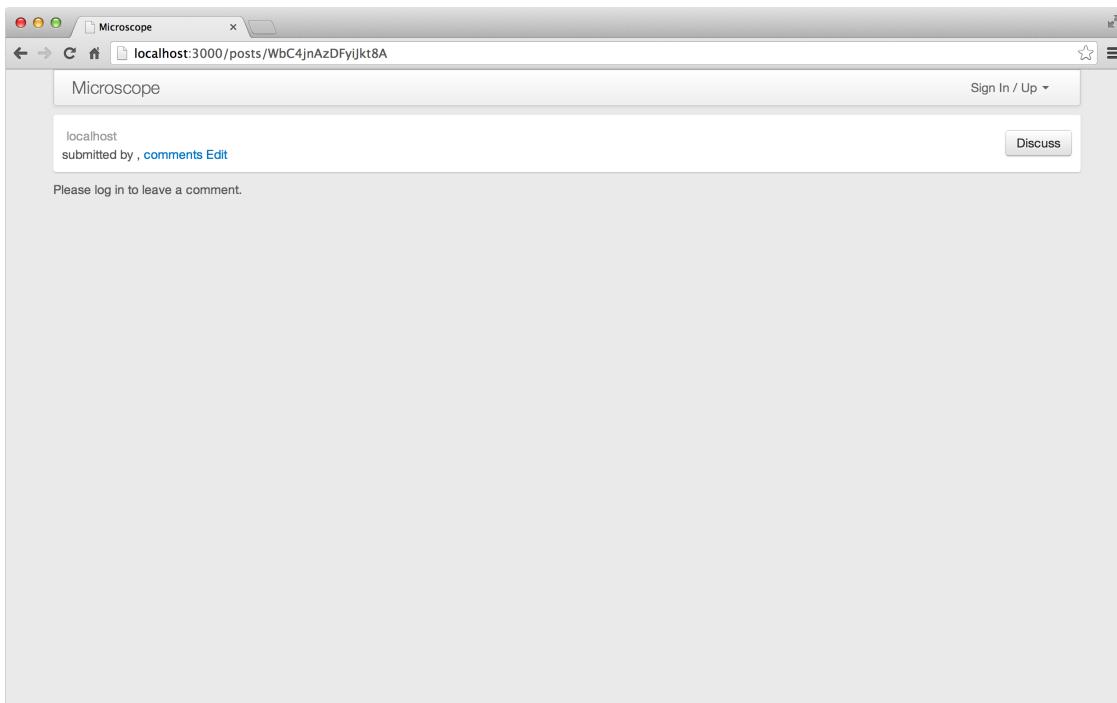
client/templates/posts/posts_list.html

Añadir un spinner para hacer la paginación más atractiva.

[Ver en GitHub](#)

[Lanzar instancia](#)

Por defecto, cargamos los cinco últimos posts, pero ¿qué pasa si vamos a la página de un post individual?



Si lo pruebas, se mostrará un error “no encontrado”. En realidad, tiene sentido: le hemos dicho al router que se suscriba a la publicación `posts` cuando carga la ruta `postsList`, pero no le hemos dicho qué debe hacer con la ruta `postpage`.

Pero hasta el momento, lo único que sabemos es suscribirnos a una lista de los `n` últimos posts. ¿Cómo pedimos al

servidor un solo post? Te contaré un pequeño secreto: ¡podemos usar más de una publicación para cada colección!

Así que para volver a ver los posts perdidos, crearemos una nueva y separada publicación `singlePost` que solo publica un post, identificado por `_id`.

```
Meteor.publish('posts', function(options) {
  return Posts.find({}, options);
});

Meteor.publish('singlePost', function(id) {
  check(id, String);
  return Posts.find(id);
});

//...
```

server/publications.js

Ahora, vamos a suscribirnos a los posts correctos en el lado del cliente. Ya estamos suscritos a la publicación `comments` en la función `waitOn` de la ruta `postPage`, por lo que simplemente podemos añadir ahí la suscripción a `singlePost`. Sin olvidarnos de añadir la suscripción a la ruta `postEdit`, que también necesita los mismos datos:

```
//...

Router.route('/posts/:_id', {
  name: 'postPage',
  waitOn: function() {
    return [
      Meteor.subscribe('singlePost', this.params._id),
      Meteor.subscribe('comments', this.params._id)
    ];
  },
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/posts/:_id/edit', {
  name: 'postEdit',
  waitOn: function() {
    return Meteor.subscribe('singlePost', this.params._id);
  },
  data: function() { return Posts.findOne(this.params._id); }
});

//...
```

lib/router.js

Usando una sola suscripción a los posts para
asegurarnos...

[Ver en GitHub](#)

[Lanzar instancia](#)

Terminada la paginación, nuestra aplicación ya no sufre de problemas de escalado, y los usuarios pueden contribuir con muchos más enlaces que antes. ¿No estaría bien tener una forma de clasificarlos? Si no lo sabías, ¡este es precisamente el

tema del siguiente capítulo!

Ahora que nuestro sitio es cada vez más popular, empieza a ser complicado buscar los mejores posts. Lo que necesitamos es algún tipo de sistema de clasificación para ordenarlos.

Podríamos construir un sistema de clasificación complejo con karma, basado en tiempo, y muchas otras cosas (la mayoría de las cuales se implementan en **Telescope**, el hermano mayor de Microscope). Nosotros vamos a mantener las cosas sencillas y ordenaremos los posts por el número de votos que reciban.

Vamos a empezar proporcionando a los usuarios una manera de votar los posts.

Vamos a guardar una lista de upvoters en cada post para que sepamos dónde mostrar el botón upvote a los usuarios, así como para evitar que la gente vote varias veces el mismo post.

Vamos a publicar las listas de upvoters a todos los usuarios, por lo que automáticamente tendrán a su disposición los datos a través de la consola de navegador.

Este es el tipo de problema de privacidad que puede surgir por la forma en la que trabajan las colecciones. Por ejemplo, ¿queremos que la gente pueda averiguar quién ha votado sus posts? En nuestro caso, no tendría ninguna consecuencia, pero es importante por lo menos reconocer el problema.

También vamos a denormalizar el número total de upvoters de un post para que sea más fácil recuperar esa cifra. Así que vamos a añadir dos atributos a nuestros posts, `upvoters` y `votes`. Vamos a empezar añadiéndolos a nuestros datos de prueba:

```
// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // create two users
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachaId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sacha = Meteor.users.findOne(sachaId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sacha._id,
    author: sacha.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: new Date(now - 7 * 3600 * 1000),
    commentsCount: 2,
    upvoters: [],
    votes: 0
  });
}
```

```

Comments.insert({
  postId: telescopeId,
  userId: tom._id,
  author: tom.profile.name,
  submitted: new Date(now - 5 * 3600 * 1000),
  body: 'Interesting project Sacha, can I get involved?'
});

Comments.insert({
  postId: telescopeId,
  userId: sacha._id,
  author: sacha.profile.name,
  submitted: new Date(now - 3 * 3600 * 1000),
  body: 'You sure can Tom!'
});

Posts.insert([
  {
    title: 'Meteor',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://meteor.com',
    submitted: new Date(now - 10 * 3600 * 1000),
    commentsCount: 0,
    upvoters: [],
    votes: 0
  }
]);

Posts.insert([
  {
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteorbook.com',
    submitted: new Date(now - 12 * 3600 * 1000),
    commentsCount: 0,
    upvoters: [],
    votes: 0
  }
]);

for (var i = 0; i < 10; i++) {
  Posts.insert({
    title: 'Test post #' + i,
    author: sacha.profile.name,
    userId: sacha._id,
    url: 'http://google.com/?q=test-' + i,
    submitted: new Date(now - i * 3600 * 1000 + 1),
    commentsCount: 0,
    upvoters: [],
    votes: 0
  });
}
}

```

server/fixtures.js

Como de costumbre, ejecutamos `meteor reset` y creamos una nueva cuenta de usuario. Ahora nos aseguraremos que inicializamos las dos nuevas propiedades cuando se crean los posts:

```

//...

var postWithSameLink = Posts.findOne({url: postAttributes.url});
if (postWithSameLink) {
  return {
    postExists: true,
    _id: postWithSameLink._id
  }
}

var user = Meteor.user();
var post = _.extend(postAttributes, {
  userId: user._id,
  author: user.username,
  submitted: new Date(),
  commentsCount: 0,
  upvoters: [],
  votes: 0
});

var postId = Posts.insert(post);

return {
  _id: postId
};

//...

```

collections/posts.js

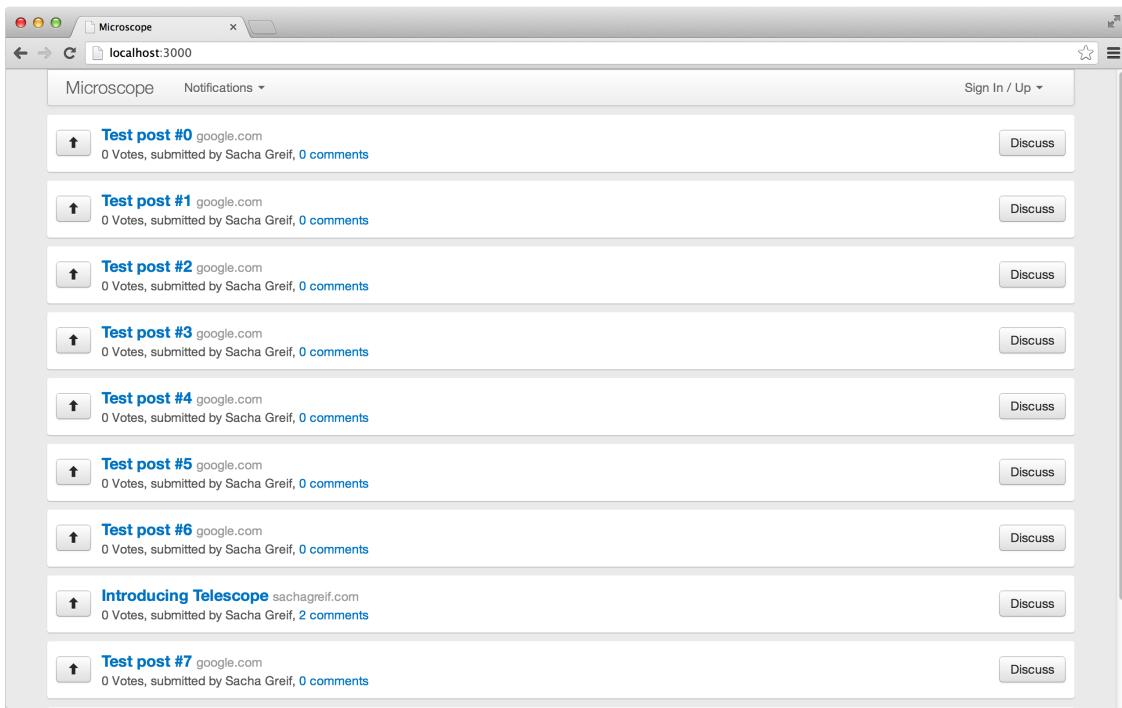
Lo primero es añadir un botón upvote a nuestros posts y mostrar el contador de votos en los metadatos del post:

```

<template name="postItem">
  <div class="post">
    <a href="#" class="upvote btn btn-default">↑</a>
    <div class="post-content">
      <h3><a href="{{url}}>{{title}}</a><span>{{domain}}</span></h3>
      <p>
        {{votes}} Votes,
        submitted by {{author}},
        <a href="{{pathFor 'postPage'}}>{{commentsCount}} comments</a>
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}>Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  </div>
</template>

```

client/templates/posts/post_item.html



Después, llamaremos a un método en el servidor cuando el usuario haga clic en el botón:

```
//...

Template.postItem.events({
  'click .upvote': function(e) {
    e.preventDefault();
    Meteor.call('upvote', this._id);
  }
});
```

client/templates/posts/post_item.js

Finalmente, volvemos a `lib/collections/posts.js` para añadir el método de servidor que actualizará los votos:

```

//...

Meteor.methods({
  post: function(postAttributes) {
    //...
  },

  upvote: function(postId) {
    check(this.userId, String);
    check(postId, String);

    var post = Posts.findOne(postId);
    if (!post)
      throw new Meteor.Error('invalid', 'Post not found');

    if (_.include(post.upvoters, this.userId))
      throw new Meteor.Error('invalid', 'Already upvoted this post');

    Posts.update(post._id, {
      $addToSet: {upvoters: this.userId},
      $inc: {votes: 1}
    });
  }
});

//...

```

lib/collections/posts.js

Algoritmo básico de voto.

[Ver en GitHub](#)

[Lanzar instancia](#)

El método es bastante sencillo. Hacemos algunas comprobaciones para garantizar que el usuario ha iniciado sesión y que el post realmente existe. Después de corroborar que el usuario no ha votado ya este post, incrementamos el total de votos y añadimos al usuario a la lista de upvoters.

Este último paso es muy interesante. Hemos utilizado un par de operadores de Mongo que son muy útiles: `$addToSet` agrega un elemento a una lista siempre y cuando este no exista ya en ella, y `$inc` simplemente incrementa un entero.

Si el usuario no está conectado, o ya ha votado uno de los posts, no podrá votar el post. Para reflejar esto en nuestra interfaz de usuario, usaremos un ayudante para añadir condicionalmente una clase CSS para que deshabilite el botón upvote.

```

<template name="postItem">
  <div class="post">
    <a href="#" class="upvote btn btn-default {{upvotedClass}}>↑</a>
    <div class="post-content">
      //...
    </div>
  </template>

```

client/templates/posts/post_item.html

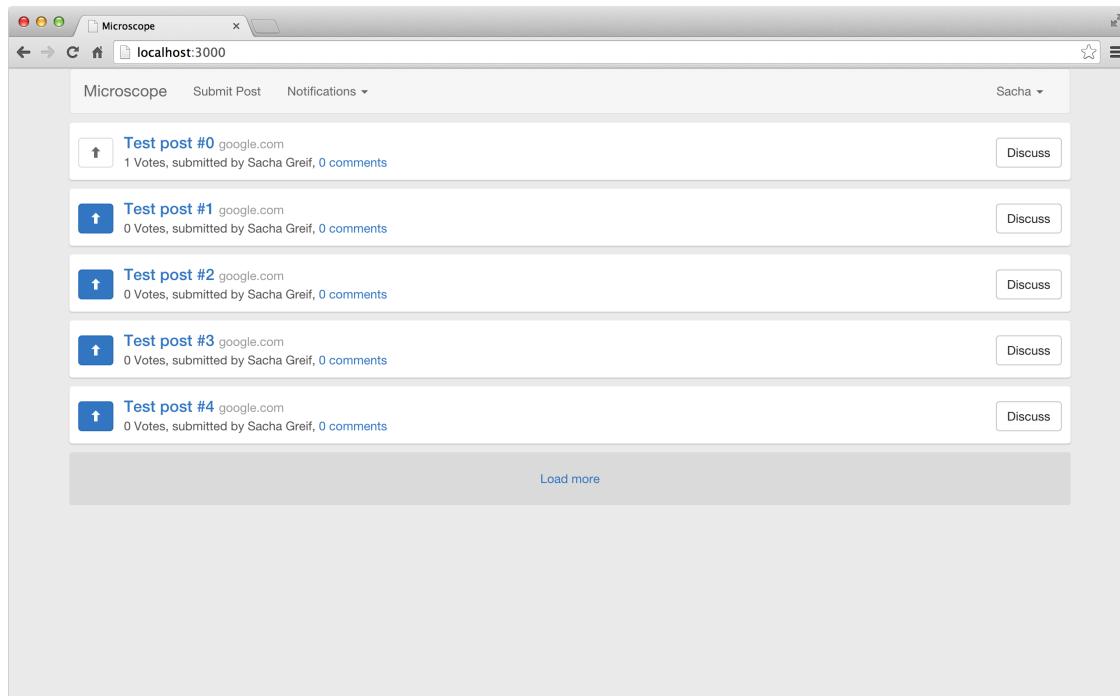
```

Template.postItem.helpers({
  ownPost: function() {
    //...
  },
  domain: function() {
    //...
  },
  upvotedClass: function() {
    var userId = Meteor.userId();
    if (userId && !_.include(this.upvoters, userId)) {
      return 'btn-primary upvotable';
    } else {
      return 'disabled';
    }
  }
});

Template.postItem.events({
  'click .upvotable': function(e) {
    e.preventDefault();
    Meteor.call('upvote', this._id);
  }
});

```

Creamos el ayudante `upvotedClass` para cambiar la clase `.upvote` por `.upvotable`, así que no podemos olvidar hacerlo también en el controlador de eventos. `client/views/posts/post_item.js`:



Deshabilitado el botón upvote si el usuario no ha
accedido...

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora nos damos cuenta que los posts con un solo voto están etiquetados como “1 votes”, por lo que vamos a pararnos a pluralizar las etiquetas correctamente. La pluralización puede ser un proceso complicado, pero por ahora vamos a hacerlo de una manera bastante simple. Crearemos un nuevo ayudante Spacebars que podemos utilizar desde cualquier lugar:

```
Template.registerHelper('pluralize', function(n, thing) {  
  // fairly stupid pluralizer  
  if (n === 1) {  
    return '1 ' + thing;  
  } else {  
    return n + ' ' + thing + 's';  
  }  
});
```

client/helpers/spacebars.js

Los ayudantes que hemos creado con anterioridad siempre han estado relacionados con la plantilla a la que se aplican. Pero usando `Template.registerHelper`, hemos creado un ayudante *global* que se puede utilizar dentro de cualquier plantilla:

```
<template name="postItem">  
  
//...  
  
<p>  
  {{pluralize votes "Vote"}},  
  submitted by {{author}},  
  <a href="{{pathFor 'postPage'}}>{{pluralize commentsCount "comment"}}</a>  
  {{#if ownPost}}<a href="{{pathFor 'postEdit'}}>Edit</a>{{/if}}  
</p>  
  
//...  
  
</template>
```

client/templates/posts/post_item.html

The screenshot shows a list of test posts on a website. Each post has a blue upvote icon, the title 'Test post #n', the URL 'google.com', the number of votes (0), the submitter ('Sacha Greif'), and the number of comments (0 or 2). A 'Discuss' button is also present next to each post.

Post	Title	URL	Votes	Submitter	Comments	Action
0	Test post #0	google.com	0	Sacha Greif	0	Discuss
1	Test post #1	google.com	0	Sacha Greif	0	Discuss
2	Test post #2	google.com	1	Sacha Greif	0	Discuss
3	Test post #3	google.com	0	Sacha Greif	0	Discuss
4	Test post #4	google.com	0	Sacha Greif	0	Discuss
5	Test post #5	google.com	0	Sacha Greif	0	Discuss
6	Test post #6	google.com	0	Sacha Greif	0	Discuss
7	Introducing Telescope	sachagreif.com	0	Sacha Greif	2	Discuss
8	Test post #7	google.com	0	Sacha Greif	0	Discuss

Añadida una función para pluralizar textos.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora ya deberíamos ver “1 Vote”.

Nuestro código para el método `upvote` en `collections/posts.js` parece bueno, pero todavía podemos hacerlo mejor. En él, hacemos dos llamadas a Mongo: una para obtener el post y otra para actualizarlo.

Veamos como con esta aproximación, tenemos dos problemas. En primer lugar, es ineficaz consultar la base de datos dos veces. Pero lo más importante es que se introduce una condición de carrera. Estamos siguiendo el siguiente algoritmo:

1. Coger el post desde la base de datos.
2. Comprobar si el usuario lo ha votado.
3. Si no, añadir un voto.

¿Qué pasa si el mismo usuario vota el mismo post entre los pasos 1 y 3? Nuestro código abre la puerta al usuario a votar dos veces el mismo post. Afortunadamente, Mongo nos permite ser más inteligentes y combinar las consultas 1 y 3 en una sola:

```
//...

Meteor.methods({
  post: function(postAttributes) {
    //...
  },

  upvote: function(postId) {
    check(this.userId, String);
    check(postId, String);

    var affected = Posts.update({
      _id: postId,
      upvoters: {$ne: this.userId}
    }, {
      $addToSet: {upvoters: this.userId},
      $inc: {votes: 1}
    });

    if (! affected)
      throw new Meteor.Error('invalid', "You weren't able to upvote that post");
  }
});

//...
```

collections/posts.js

Mejorado el algoritmo de voto.

[Ver en GitHub](#)

[Lanzar instancia](#)

Lo que estamos diciendo es “encuentra todos los posts con este `id` que todavía no hayan sido votados por este usuario, y actualízalos”. Si el usuario aún no ha votado el post con esta `id`, lo encontrará, pero si ya lo ha hecho, la consulta no coincidirá con ningún documento, y por lo tanto no ocurrirá nada.

Digamos que tratas de engañarnos y pones uno de tus posts el primero de la lista ajustando su número de votos desde la consola del navegador:

```
> Posts.update(postId, {$set: {votes: 10000}});
```

Consola del navegador

(Donde `postId` es el id de uno de tus posts)

Este descarado intento de jugar con el sistema sería capturado por nuestro callback `deny()` (en `collections/posts.js`, ¿recuerdas?) e inmediatamente denegada.

Pero si miras detenidamente, podrás ver la compensación de latencia en acción. Puede ser rápido, pero el post saltará brevemente al principio de la lista antes de volver de nuevo a su posición.

¿Qué está pasando? En su colección de `Posts` locales, la actualización se ha aplicado sin incidentes. Esto sucede al instante, por lo que el mensaje se va al principio de la lista. Mientras tanto, en el servidor, se deniega la actualización, de forma que un poco más tarde (milisegundos si estás ejecutando Meteor en tu propia máquina), el servidor devuelve un error, obligando a la colección local a revertirse.

El resultado final: mientras esperamos a que el servidor responda, la interfaz de usuario no puede dejar de confiar en la colección local. Tan pronto como el servidor deniega la modificación, las interfaces de usuario se adaptan para reflejarlo.

Ahora que disponemos de una puntuación para cada post en función del número de votos, vamos a mostrar una lista de los mejores. Para ello, vamos a ver cómo gestionamos dos suscripciones separadas contra la colección de posts, y, de paso hacer nuestra plantilla `postsList` un poco más general.

Para empezar, queremos tener *dos* suscripciones, una para cada tipo de ordenación. El truco aquí es suscribirse a la misma publicación, ¡solo que con diferentes argumentos!

También crearemos dos nuevas rutas denominadas `newPosts` y `bestPosts`, accesibles desde las direcciones `/new` y `/best` respectivamente (junto con `/new/5` y `/best/5` para la paginación, por supuesto).

Para ello, vamos a extender nuestro `PostsListController` en dos controladores distintos: `NewPostsListController` y `BestPostsListController`. Esto nos permitirá reutilizar las mismas opciones de ruta, tanto para las rutas `home` y `newPosts`, quedándonos un solo `NewPostsListController` del que heredar. Y, además, todo esto es una buena ilustración de lo flexible que puede ser Iron Router.

Vamos a reemplazar la propiedad de ordenación `{submitted: -1}` en `PostsListController` por `this.sort`, que será proporcionado por `NewPostsListController` y `BestPostsListController`:

```

//...

PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: this.sort, limit: this.postsLimit()};
  },
  subscriptions: function() {
    this.postsSub = Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.postsLimit();
    return {
      posts: this.posts(),
      ready: this.postsSub.ready,
      nextPath: hasMore ? this.nextPath() : null
    };
  }
});

NewPostsController = PostsListController.extend({
  sort: {submitted: -1, _id: -1},
  nextPath: function() {
    return Router.routes.newPosts.path({postsLimit: this.postsLimit() + this.increment})
  }
});

BestPostsController = PostsListController.extend({
  sort: {votes: -1, submitted: -1, _id: -1},
  nextPath: function() {
    return Router.routes.bestPosts.path({postsLimit: this.postsLimit() + this.increment})
  }
});

Router.route('/', {
  name: 'home',
  controller: NewPostsController
});

Router.route('/new/:postsLimit?', {name: 'newPosts'});

Router.route('/best/:postsLimit?', {name: 'bestPosts'});

```

lib/router.js

Ten en cuenta que ahora que tenemos más de una ruta, sacamos la lógica para `nextPath` de `PostsListController` y la ponemos en `NewPostsController` y `BestPostsController`, ya que el path será diferente en uno y otro caso.

Además, cuando ordenamos por votos, establecemos un segundo orden por fecha y por `_id` para garantizar que el orden está completamente especificado.

Una vez listos los nuevos controladores podemos deshacernos de la ruta anterior a `postsList`. Simplemente eliminamos el siguiente código:

```
Router.route('/:postsLimit?', {
  name: 'postsList'
})
```

lib/router.js

Vamos a añadir los enlaces en la cabecera:

```
<template name="header">
<nav class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="{{pathFor 'home'}}">Microscope</a>
  </div>
  <div class="collapse navbar-collapse" id="navigation">
    <ul class="nav navbar-nav">
      <li>
        <a href="{{pathFor 'newPosts'}}">New</a>
      </li>
      <li>
        <a href="{{pathFor 'bestPosts'}}">Best</a>
      </li>
      {{#if currentUser}}
        <li>
          <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
        </li>
        <li class="dropdown">
          {{> notifications}}
        </li>
      {{/if}}
    </ul>
    <ul class="nav navbar-nav navbar-right">
      {{> loginButtons}}
    </ul>
  </div>
</nav>
</template>
```

client/templates/includes/header.html

Finalmente, también necesitamos actualizar el controlador de eventos para borrar posts:

```
'click .delete': function(e) {
  e.preventDefault();

  if (confirm("Delete this post?")) {
    var currentPostId = this._id;
    Posts.remove(currentPostId);
    Router.go('home');
  }
}
```

client/templates/posts/posts_edit.js

Con todo listo, ahora obtenemos lista de posts mejorada:

The screenshot shows a web application interface titled "Microscope". At the top, there's a navigation bar with links for "Microscope", "Submit Post", and "Notifications". A user profile for "Sacha Greif" is visible. Below the navigation, a list of posts is displayed in a grid format. Each post card includes an upvote icon, the post title, the source URL, the submitter, the number of votes, the number of comments, and a "Discuss" button.

Post	Source	Submitted By	Votes	Comments	Action
Introducing Telescop	sachagreif.com	Sacha Greif	3	2	Discuss
Meteor	meteor.com	Tom Coleman	2	0	Discuss
The Meteor Book	themeteorbook.com	Tom Coleman	2	0	Discuss
Test post #0	google.com	Sacha Greif	1	0	Discuss
Test post #1	google.com	Sacha Greif	0	0	Discuss
Test post #2	google.com	Sacha Greif	0	0	Discuss
Test post #3	google.com	Sacha Greif	0	0	Discuss
Test post #4	google.com	Sacha Greif	0	0	Discuss
Test post #5	google.com	Sacha Greif	0	0	Discuss

Añadidas rutas para las listas de posts y páginas para
mo...

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora que tenemos dos listas, puede resultar difícil saber cuál de ellas estamos viendo. Así que vamos a revisar nuestra cabecera para que sea más evidente. Vamos a crear el gestor `header.js` y un ayudante auxiliar que use la ruta actual y una o más rutas con nombre para activar una clase en nuestros elementos de navegación:

La razón por la que queremos permitir varias rutas es que tanto la ruta `home` como la ruta `newPosts` (que se corresponden con las nuevas URLs `/` y `/new`) devuelven la misma plantilla, lo que significa que nuestro `activeRouteClass` debe ser lo suficientemente inteligente como para activar la etiqueta `` en ambos casos.

```

<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'home'}}>Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav">
        <li class="{{activeRouteClass 'home' 'newPosts'}}">
          <a href="{{pathFor 'newPosts'}}>New</a>
        </li>
        <li class="{{activeRouteClass 'bestPosts'}}">
          <a href="{{pathFor 'bestPosts'}}>Best</a>
        </li>
        {{#if currentUser}}
          <li class="{{activeRouteClass 'postSubmit'}}">
            <a href="{{pathFor 'postSubmit'}}>Submit Post</a>
          </li>
          <li class="dropdown">
            {{> notifications}}
          </li>
        {{/if}}
      </ul>
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>

```

client/templates/includes/header.html

```

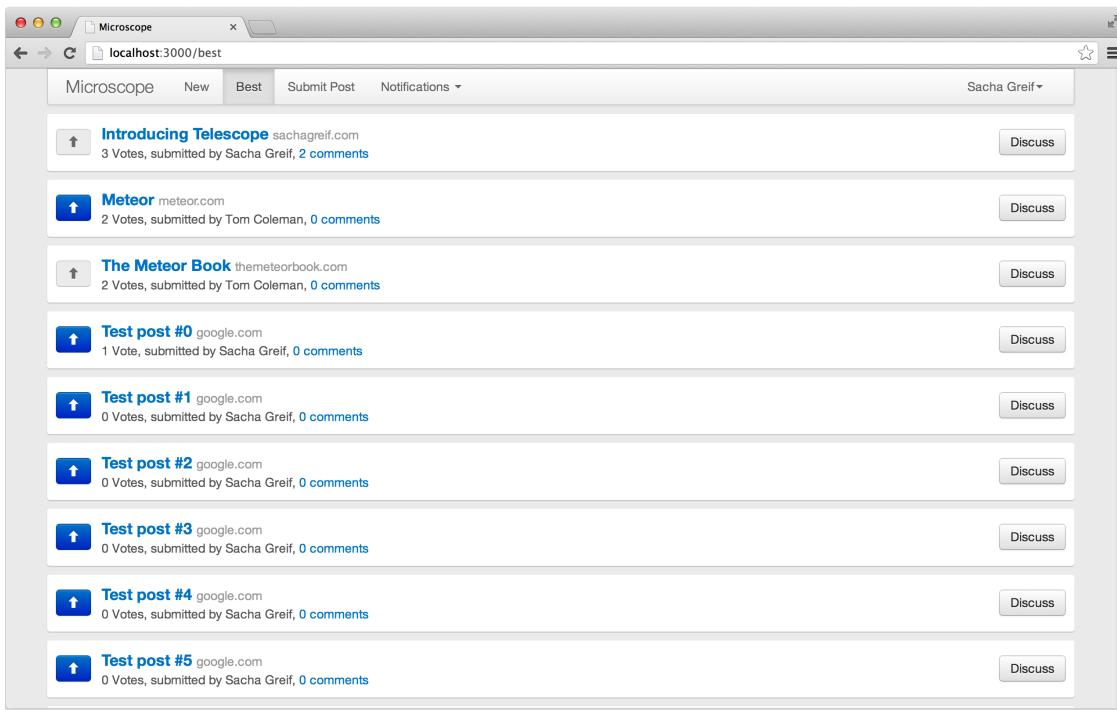
Template.header.helpers({
  activeRouteClass: function(/* route names */) {
    var args = Array.prototype.slice.call(arguments, 0);
    args.pop();

    var active = _.any(args, function(name) {
      return Router.current() && Router.current().route.getName() === name
    });

    return active && 'active';
  }
});

```

client/templates/includes/header.js



Hasta ahora no hemos usado este modelo en concreto, pero al igual que para cualquier otro tag de Spacebars, los tags de los ayudantes de plantilla también pueden tomar argumentos.

Y aunque por supuesto, se pueden pasar argumentos específicos a la función, también se pueden pasar un número indeterminado de ellos y recuperarlos mediante una llamada al objeto `arguments` dentro de la función.

En este último caso, es probable que queramos convertir el objeto `arguments` a un array JavaScript y luego llamar a `pop()` sobre él para deshacernos del hash que añade Spacebars.

Para cada elemento, el ayudante `activeRouteClass` toma una lista de nombres de ruta, y luego utiliza el ayudante `any()` de Underscore para ver si las rutas pasan la prueba (es decir, que su correspondiente URL sea igual a la actual). Si cualquiera de las rutas se corresponde con la actual, `any()` devolverá `true`.

Por último, estamos aprovechando el patrón `boolean && myString` de JavaScript con el que `false && myString` devuelve `false`, pero los `true && myString` devuelven `myString`.

Añadidas clases activas a la cabecera.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora que los usuarios pueden votar en tiempo real, podemos ver cómo saltan los posts hacia arriba o abajo según cambia su clasificación. Pero ¿no sería más agradable si hubiera una manera de suavizar estos cambios con algunas animaciones?

A estas alturas ya deberías conocer bastante bien cómo interactúan las suscripciones y las publicaciones. Así que vamos a deshacernos de las ruedas de entrenamiento y examinar algunos escenarios más avanzados.

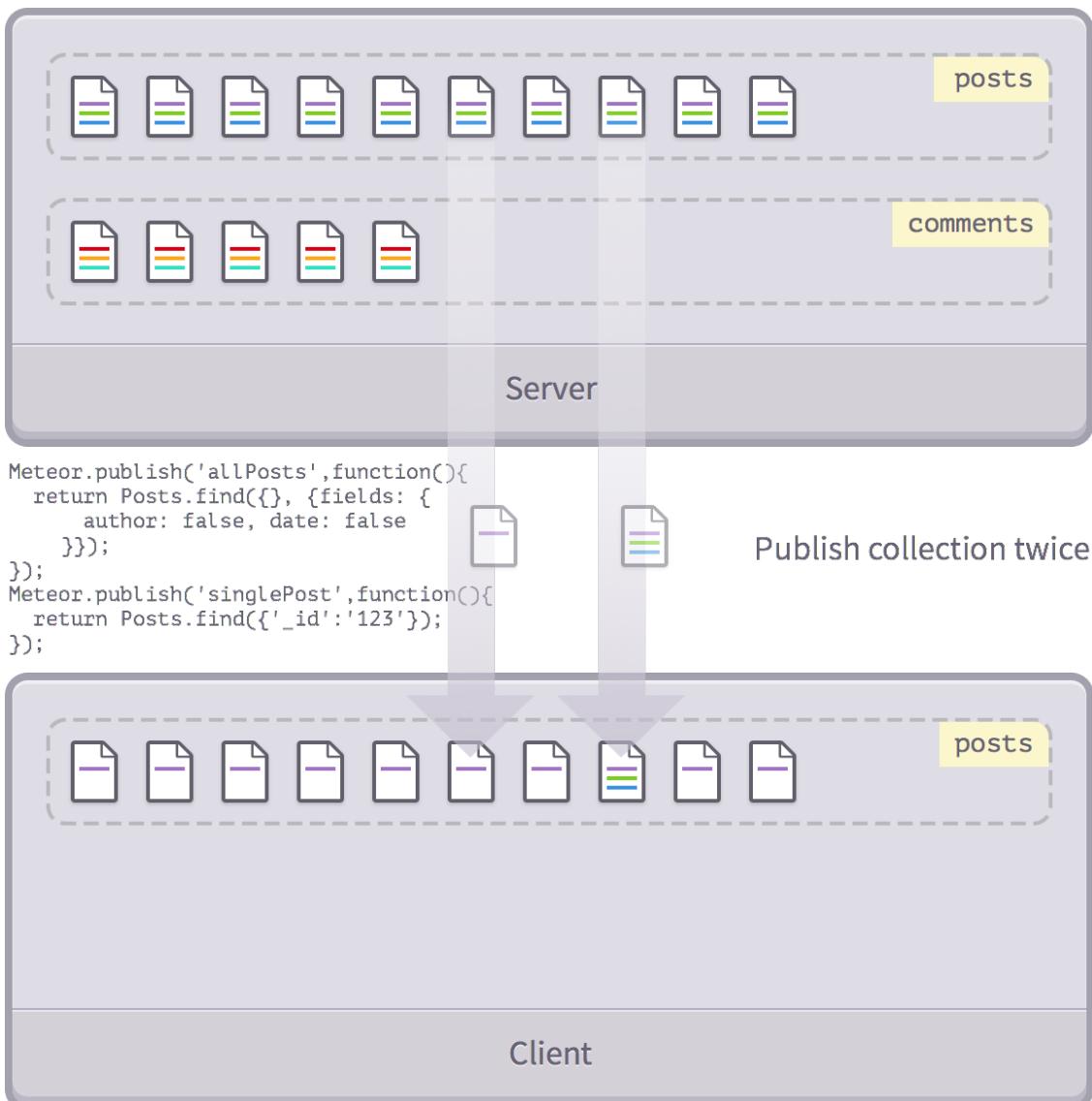
En [la primera sidebar](#), vimos algunos de los patrones más comunes de publicación y suscripción, y aprendimos cómo la función `_publishCursor` las hace muy fáciles de usar en nuestras aplicaciones.

Primero, vamos a recapitular exactamente qué hace por nosotros la función `_publishCursor`: lo que hace es tomar todos los documentos que coinciden con un cursor dado, y los envía a la colección del cliente *del mismo nombre*. Ten en cuenta que el nombre de la *publicación* no está involucrado.

Esto significa que podemos tener *más de una publicación* de cualquier colección, que enlaza al cliente y al servidor

Ya hemos encontrado este patrón en el [capítulo paginación](#), cuando publicamos un subconjunto paginado de todos los posts, además del post actual.

Otro caso de uso similar es publicar una *overview* de un gran conjunto de documentos, así como también los detalles completos de un único ítem:



```

Meteor.publish('allPosts', function() {
  return Posts.find({}, {fields: {title: true, author: true}});
});

Meteor.publish('postDetail', function(postId) {
  return Posts.find(postId);
});

```

Ahora cuando el cliente se suscriba a esas dos publicaciones, su colección 'posts' se rellena desde dos fuentes: una lista de títulos y nombres de autor de la primera suscripción, y los detalles completos de un único post de la segunda.

Tal vez te hayas dado cuenta de que el post publicado por `postDetail` se publica también desde `allPosts` (aunque solo con un subconjunto de sus propiedades). Sin embargo, Meteor se hace cargo de la superposición fusionando los campos y asegurando que no haya duplicados.

Esto es genial, porque ahora cuando renderizemos la lista de los resúmenes de los posts, estaremos lidiando con objetos de datos que tienen lo suficiente para mostrar lo que necesitamos. Y cuando renderizemos la página de un sólo post, también lo tenemos. Por supuesto, tenemos que ocuparnos que el cliente no espere que todos los campos estén disponibles en todos los posts – ¡esto es un error común!

Ten en cuenta que no estamos limitados a variar las propiedades de los documentos. Podríamos también, publicar las mismas propiedades en ambas publicaciones, pero ordenar los items de otra manera.

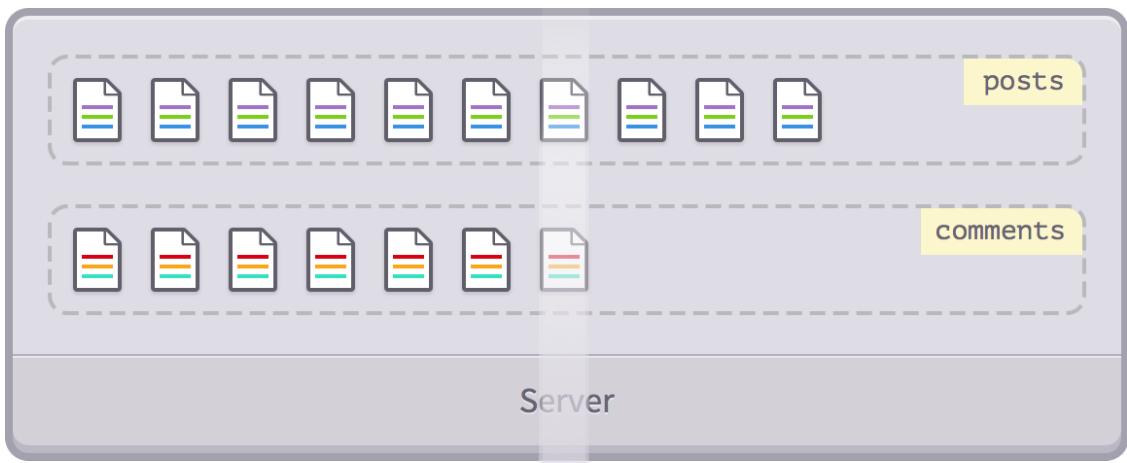
```
Meteor.publish('newPosts', function(limit) {  
  return Posts.find({}, {sort: {submitted: -1}, limit: limit});  
});  
  
Meteor.publish('bestPosts', function(limit) {  
  return Posts.find({}, {sort: {votes: -1, submitted: -1}, limit: limit});  
});
```

server/publications.js

Acabamos de ver cómo se puede publicar una sola colección más de una vez. Resulta que se puede lograr un resultado muy similar con otro patrón: creando una única publicación, pero suscribiéndonos a ella varias veces.

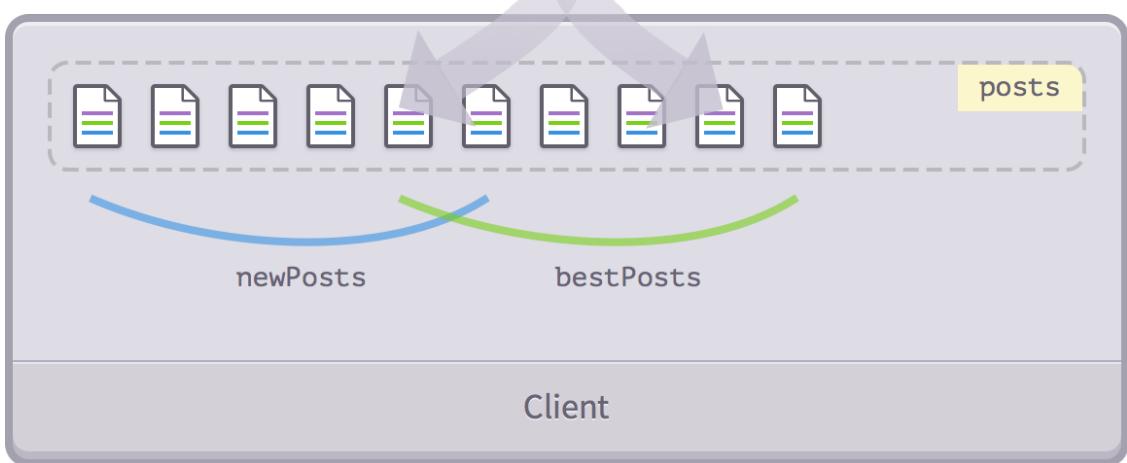
En Microscope, nos suscribimos a la publicación de `posts` varias veces, pero Iron Router activa y desactiva cada suscripción por nosotros. Aun así, no hay ninguna razón por la cual no podamos suscribirnos muchas veces *simultáneamente*.

Por ejemplo, digamos que queremos cargar los posts más recientes y los mejores en la memoria al mismo tiempo:



```
newPostsHandle = Meteor.subscribe('posts',
{submitted: -1}, 10);
bestPostsHandle = Meteor.subscribe('posts',
{baseScore: -1, submitted: -1}, 10)
```

Subscribing twice



Estamos estableciendo una sola publicación:

```
Meteor.publish('posts', function(options) {
  return Posts.find({}, options);
});
```

Luego, nos suscribimos a esta publicación múltiples veces. De hecho, esto es más o menos exactamente lo que estamos haciendo en Microscope:

```
Meteor.subscribe('posts', {submitted: -1, limit: 10});
Meteor.subscribe('posts', {baseScore: -1, submitted: -1, limit: 10});
```

Entonces, ¿qué está pasando exactamente? Cada navegador abre dos suscripciones diferentes, cada uno se conecta a la *misma* publicación en el servidor.

Cada suscripción ofrece diferentes argumentos para esa publicación, pero fundamentalmente, cada vez que un conjunto de documentos (diferente) se saca de la colección `posts`, se envía por el canal a la colección del lado del cliente.

Puedes incluso suscribirte dos veces a la misma publicación con *los mismos argumentos!* Es difícil pensar en escenarios donde esto sea de utilidad, ¡pero esta flexibilidad puede que sea útil algún día!

A diferencia de las bases de datos relacionales tradicionales como MySQL que hacen uso de *joins*, las bases de datos NoSQL como Mongo se basan en la *denormalización* e *incrustación*. Vamos a ver cómo funciona en el contexto de Meteor.

Veamos un ejemplo concreto. Hemos añadido comentarios a nuestros posts, y hasta ahora, hemos publicado los comentarios en la página de un único post.

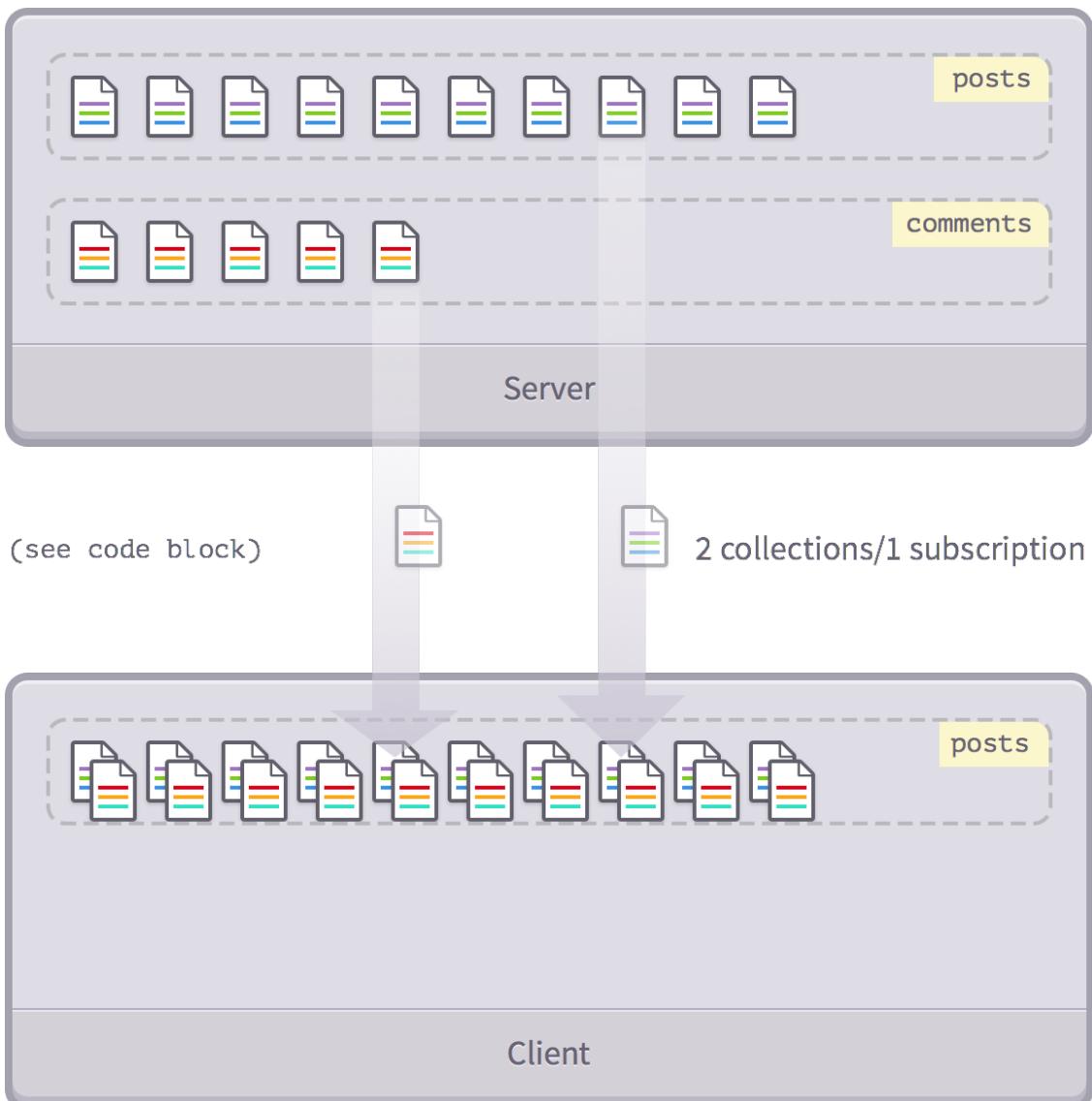
Sin embargo, supongamos que queremos mostrar *todos* los comentarios en los posts en la página principal (teniendo en cuenta que estos posts van a cambiar a medida que paginamos a través de ellos). Este caso de uso presenta una buena razón para insertar comentarios en los posts, y de hecho es lo que nos empuja a desnormalizar la *colección* de comentarios.

Por supuesto que siempre se pueden insertar los comentarios en los posts, deshaciéndonos de la colección de `comments` por completo. Pero, como hemos visto anteriormente en el capítulo *Desnormalización*, al hacerlo estaríamos perdiendo algunos beneficios adicionales de trabajar con colecciones separadas.

Pero resulta que hay un truco que hace posible embeber nuestros comentarios, preservando colecciones separadas.

Supongamos que, junto con la lista de la página principal de posts, queremos suscribirnos a una lista de los mejores 2 comentarios para cada uno de ellos.

Lograr esto con una publicación de comentarios independiente sería difícil, sobre todo si la relación de posts se limita de alguna manera (por ejemplo, los 10 más recientes). Tendríamos que crear una publicación que se pareciera a algo como esto:



```
Meteor.publish('topComments', function(topPostIds) {
  return Comments.find({postId: topPostIds});
});
```

Esto sería un problema desde el punto de vista de rendimiento, ya que habría que eliminar y volver a establecer la publicación cada vez que cambiara la lista de `topPostIds`.

Existe una manera de evitar esto. Acabamos de utilizar el hecho de que no solo podemos tener más de una publicación por colección, sino que también podemos tener más de una colección por publicación:

```

Meteor.publish('topPosts', function(limit) {
  var sub = this, commentHandles = [], postHandle = null;

  // send over the top two comments attached to a single post
  function publishPostComments(postId) {
    var commentsCursor = Comments.find({postId: postId}, {limit: 2});
    commentHandles[postId] =
      Mongo.Collection._publishCursor(commentsCursor, sub, 'comments');
  }

  postHandle = Posts.find({}, {limit: limit}).observeChanges({
    added: function(id, post) {
      publishPostComments(id);
      sub.added('posts', id, post);
    },
    changed: function(id, fields) {
      sub.changed('posts', id, fields);
    },
    removed: function(id) {
      // stop observing changes on the post's comments
      commentHandles[id] && commentHandles[id].stop();
      // delete the post
      sub.removed('posts', id);
    }
  });
}

sub.ready();

// make sure we clean everything up (note '_publishCursor'
// does this for us with the comment observers)
sub.onStop(function() { postHandle.stop(); });
});
```

Tengamos en cuenta que no estamos devolviendo nada en esta publicación, le enviamos mensajes manualmente a la `sub` nosotros mismos (a través de `.added()` y sus amigos). Así que no necesitamos que `_publishCursor` lo haga mediante la devolución de un cursor.

Ahora, cada vez que publiquemos un post también publicaremos automáticamente los dos primeros comentarios adjuntos. ¡Y todo con una sola llamada a una suscripción!

Aunque Meteor aún no hace este enfoque muy sencillo, también se puede utilizar el paquete `publish-with-relations` de Atmosphere, cuyo objetivo es hacer que este patrón sea más fácil de usar.

¿Qué más puede darnos nuestro nuevo conocimiento sobre la flexibilidad de las suscripciones? Bueno, si no usamos `_publishCursor`, no tendremos la restricción de que la fuente de la colección en el servidor necesita tener el mismo nombre que la colección de destino en el cliente.



Una razón por la que no queríamos hacer esto es la *Herencia de Tabla Simple*

Supongamos que quisiéramos referenciar varios tipos de objetos desde nuestros posts, cada uno alojado en campos comunes pero ligeramente diferentes en contenido. Por ejemplo, podríamos estar creando un motor de blogging al estilo de Tumblr en el que cada post posee el habitual ID, un timestamp, y el título. Pero también puede tener imágenes, videos, links o simplemente texto.

Podríamos guardar todos estos objetos en una colección llamada `'resources'` (recursos), usando un atributo `type` que indique qué tipo de objeto son (`video`, `image`, `link`, etc.).

Y aunque tendríamos una sola colección `resources` en el servidor, podríamos transformar esa única colección en múltiples colecciones en el cliente, como `Videos`, `Images`, etc., con el siguiente trozo de magia:

```
Meteor.publish('videos', function() {
  var sub = this;

  var videosCursor = Resources.find({type: 'video'});
  Mongo.Collection._publishCursor(videosCursor, sub, 'videos');

  // _publishCursor doesn't call this for us in case we do this more than once.
  sub.ready();
});
```

Le estamos diciendo a `_publishCursor` que publique nuestros videos (como hacer un return) como lo haría el cursor, pero en lugar de publicar la colección `resources` en el cliente, publicamos de `resources` a `videos`.

Otra idea similar es usar `publish` para una colección en el lado del cliente donde *no hay ninguna colección en el lado servidor!*. Por ejemplo, podrías obtener datos de un servicio de terceros, y publicarlos como si fuera una colección en el cliente.

Gracias a la flexibilidad de la API de publicación, las posibilidades son ilimitadas.

Aunque contamos un sistema de votación en tiempo real, no tenemos una gran experiencia de usuario viendo la forma en la que los posts se mueven en la página principal. Usaremos animaciones para suavizar este problema.

_uihooks

Los `_uihooks` son una característica de Blaze relativamente nueva y poco documentada. Como su propio nombre indica, nos da acceso a acciones que podemos ejecutar cuando se insertan, eliminan o animan elementos.

La lista completa de acciones es esta:

- `insertElement` : se llama cuando se inserta un elemento.
- `moveElement` : se llama cuando un elemento cambia su posición.
- `removeElement` : se llama cuando se elimina un elemento.

Una vez definidas, estas acciones *reemplazarán* el comportamiento que Meteor tiene por defecto. En otras palabras, en vez de insertar, mover o eliminar elementos, Meteor usará el comportamiento que hayamos definido – ¡y será cosa nuestra que ese comportamiento sea correcto!

Antes de poder empezar con la parte divertida (hacer que se muevan las cosas), tenemos que entender cómo interactúa Meteor con el DOM (Document Object Model - la colección de elementos HTML que componen el contenido de una página).

Lo más importante que hay que tener en cuenta es que los elementos del DOM realmente no se pueden “mover”. Sólo se pueden añadir y eliminar (esto es una limitación del propio DOM, no de Meteor). Así que para crear la ilusión de que los elementos A y B se intercambian, Meteor tendrá que eliminar B e insertar una nueva copia (B') antes del elemento A.

Esto hace de la animación algo complicado ya que, no podemos simplemente mover B a una nueva posición, porque B habrá desaparecido tan pronto como Meteor redibuje de nuevo la página (que, como sabemos, sucede instantáneamente gracias a la reactividad). Pero no te preocupes, encontraremos la manera de hacerlo.

Pero, primero, una historia.

En 1980, en pleno apogeo de la guerra fría, los Juegos Olímpicos se celebraban en Moscú, y los soviéticos estaban decididos a ganar la carrera de 100 metros a cualquier precio. Así que un grupo de brillantes científicos soviéticos equiparon a uno de sus atletas con un teletransportador, y en cuanto sonó el disparo de salida, el corredor fue trasladado de inmediato a la línea de meta.

Afortunadamente, los jueces de la carrera se dieron cuenta de la infracción inmediatamente, y el atleta no tuvo más remedio que teletransportarse de nuevo a su casilla de salida, antes de permitirle participar de nuevo corriendo como los demás.

Mis fuentes históricas no son muy fiables, por lo que debes tomar esa historia como cogida con pinzas. Pero trataremos de mantener en mente la analogía del “corredor soviético con teletransportador” a medida que avancemos en este capítulo.

Cuando Meteor recibe una actualización y modifica reactivamente el DOM, nuestro post se teletransporta inmediatamente a su posición final, al igual que el corredor soviético. Pero como en los Juegos Olímpicos, en nuestra aplicación, podemos tener alrededor cosas que no se teletransportan. Así que tendremos que teletransportarlo a la “casilla de salida” y hacerlo “correr” (en otras palabras, animarlo) de nuevo hasta la línea de meta.

Para intercambiar los elementos A y B (situados en las posiciones p1 y p2, respectivamente), tendremos que seguir los siguientes pasos:

1. Borrar B
2. Crear B' antes de A en el DOM
3. Teletransportar B' a p2
4. Teletransportar A a p1
5. Animar A hasta p2
6. Animar B' hasta p1

El siguiente diagrama explica estos pasos con más detalle:

Step	User Interface	DOM
Step 0 Start		<div id="postA">...</div> <div id="postB">...</div>
Step 1 Delete Post B		<div id="postA">...</div>
Step 2 Create the new Post B'		<div id="postB">...</div> <div id="postA">...</div>
Step 3 Move B' to p2		<div id="postB">...</div> <div id="postA">...</div>
Step 4 Move A to p1		<div id="postB">...</div> <div id="postA">...</div>
Step 5 Animate A to p2		<div id="postB">...</div> <div id="postA">...</div>
Step 6 Animate B' to p1		<div id="postB">...</div> <div id="postA">...</div>

De nuevo, en los pasos 3 y 4 no estamos *animando* A y B' hasta sus posiciones sino que las “teletransportamos” allí al instante. Dado que el cambio es instantáneo, parecerá que B no se ha borrado, pero ya tenemos posicionados correctamente los elementos para que puedan ser animados hasta su nueva posición.

Afortunadamente, Meteor se ocupa de los pasos 1 y 2 y re-implementarlos será una tarea fácil. En los pasos 5 y 6, lo único que hacemos es mover los elementos al lugar adecuado. Así que, sólo tenemos que preocuparnos de los pasos 3 y 4, enviar los elementos al punto de arranque de la animación.

Para animar los posts que se están reordenando por la página, vamos a tener que meternos en territorio CSS. Sería recomendable una rápida revisión del posicionamiento con CSS.

Los elementos de una página utilizan posicionamiento **estático** por defecto. Los elementos posicionados de forma estática están fijos y sus coordenadas no se pueden cambiar o animar.

Por otra parte, el posicionamiento **relativo**, implica que el elemento está fijado a la página, pero se puede mover con relación a su posición original.

El posicionamiento **absoluto** va un paso más allá y permite dar coordenadas x/y a un elemento en relación al **documento** o al primer elemento “padre” **posicionado de forma absoluta o relativa**

Nosotros vamos a usar posicionamiento relativo en nuestras animaciones. Ya disponemos del CSS necesario en `client/stylesheets/style.css`, pero si necesitas añadirlo, este es el código para la hoja de estilo:

```
.post{
  position: relative;
}
.post.animate{
  transition: all 300ms 0ms ease-in;
}
```

`client/stylesheets/style.css`

Ten en cuenta que sólo animamos los posts con la clase CSS `.animate`. De esta forma, podemos añadir y quitar esa clase para controlar cuándo deben producirse o no las animaciones.

Esto facilita muchos los pasos 5 y 6: todo lo que necesitamos hacer es configurar la parte `top` a `0px` (su valor predeterminado) y nuestros posts se deslizarán de nuevo a su posición “normal”.

Esto significa que nuestro único problema es averiguar desde dónde animar los posts (pasos 3 y 4) con respecto a su nueva posición. En otras palabras, en qué posición hay que ponerlo. Pero, esto no es tan difícil: el desplazamiento correcto (offset) es la posición anterior restada a la nueva.

Ahora que entendemos los diferentes factores que entran en juego en la animación de una lista de elementos, estamos listos para empezar a aplicar la animación. Lo primero que necesitaremos para envolver nuestra lista de posts en un nuevo contenedor `.wrapper`:

```

<template name="postsList">
  <div class="posts page">
    <div class="wrapper">
      {{#each posts}}
        {{> postItem}}
      {{/each}}
    </div>

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}>Load more</a>
    {{else}}
      {{#unless ready}}
        {{> spinner}}
      {{/unless}}
    {{/if}}
  </div>
</template>

```

client/templates/posts/posts_list.html

Antes de continuar, vamos a revisar cuál es el comportamiento actual *sin animaciones*:

La lista de posts no-animada.

Vamos a por los `_uihooks`. Dentro del callback `onRendered` de la plantilla, seleccionamos el div `.wrapper`, y definimos la acción `moveElement`.

```

Template.postsList.onRendered(function () {
  this.find('.wrapper')._uihooks = {
    moveElement: function (node, next) {
      // do nothing for now
    }
  };
});

```

client/templates/posts/posts_list.js

Cada vez que cambie la posición de un elemento, *en vez de* obtener el comportamiento predeterminado de Blaze, Meteor llamará a la función `moveElement`. Y, dado que la función está vacía, *no va a pasar nada*.

Adelante, probemos: abre la vista de los “Mejores” posts y vota unos cuantos: el orden no cambiará hasta que no fuerces un re-render (ya sea volviendo a cargar la página o moviéndote entre distintas rutas).

Un callback `moveElement` vacío: no ocurre nada

Hemos comprobado que los `_uihooks` funcionan. ¡Ahora vamos a hacer que animen los posts!

La acción `moveElement` toma dos argumentos: `node` y `next`.

- `node` es el elemento que se está moviendo a una nueva posición en el DOM.
- `next` es el elemento que hay justo *después* de la nueva posición a la que estamos moviendo `node`.

Sabiendo esto, podemos definir el proceso de animación (si necesitas refrescar la memoria, no dudes en volver al ejemplo del “Corredor Ruso”). Cuando detectamos un nuevo cambio en la posición de un elemento, tendremos que hacer lo siguiente:

1. Insertar `node` antes de `next` (en otras palabras, establecer el comportamiento por defecto, como si no hubiéramos definido la acción `moveElement`).
2. Mover `node` a su posición original.
3. Moveremos todos los elementos que hay entre `node` y `next` para hacer sitio a `node`.
4. Animaremos todos los elementos hasta su posición original.

Para hacer todo esto usaremos la magia de **jQuery**, de lejos, la mejor librería de manipulación del DOM que existe. jQuery está fuera del alcance de este libro, pero vamos a ver rápidamente los métodos que vamos a usar:

- Con `$()` convertimos cualquier elemento del DOM en un objeto jQuery.
- `offset()` recupera la posición de un elemento en relación *al documento*, y devuelve un objeto que contiene las propiedades `top` y `left`.
- Con `outerHeight()` obtenemos la altura “exterior” (incluyendo el padding y, opcionalmente, el margin) de un elemento.
- Con `nextUntil(selector)` obtenemos todos los elementos que hay después del elemento seleccionado con el `selector`, excepto éste último.
- Con `insertBefore(selector)` insertamos un elemento antes del que seleccionamos con el `selector`.
- Con `removeClass(class)` eliminamos la clase CSS `class`, si está presente en el elemento.
- Con `css(propertyName, propertyValue)` establecemos el valor `propertyValue` para la propiedad `propertyName`.
- Con `height()` obtenemos la altura de un elemento.
- Con `addClass(class)` añadimos la clase `class` a un elemento.

```

Template.postsList.onRendered(function () {
  this.find('.wrapper')/_uihooks = {
    moveElement: function (node, next) {
      var $node = $(node), $next = $(next);
      var oldTop = $node.offset().top;
      var height = $node.outerHeight(true);

      // find all the elements between next and node
      var $inBetween = $next.nextUntil(node);
      if ($inBetween.length === 0)
        $inBetween = $node.nextUntil(next);

      // now put node in place
      $node.insertBefore(next);

      // measure new top
      var newTop = $node.offset().top;

      // move node *back* to where it was before
      $node
        .removeClass('animate')
        .css('top', oldTop - newTop);

      // push every other element down (or up) to put them back
      $inBetween
        .removeClass('animate')
        .css('top', oldTop < newTop ? height : -1 * height);

      // force a redraw
      $node.offset();

      // reset everything to 0, animated
      $node.addClass('animate').css('top', 0);
      $inBetween.addClass('animate').css('top', 0);
    }
  };
});

```

client/templates/posts/posts_list.js

Algunas notas:

- Calculamos la altura de `$node` para saber cuánto debemos mover los elementos `$inBetween`. Y usamos `outerHeight(true)` para incluir margen y padding en el cálculo.
- No sabemos si `next` va antes o después de `node` así que comprobamos las dos configuraciones cuando definimos `$inBetween`.
- Para cambiar los elementos de “teletransportados” a “animados”, simplemente añadimos o quitamos la clase `animate` (la animación definida en el código CSS de la aplicación).
- Dado que usamos posicionamiento relativo, siempre podemos poner a 0 la propiedad `top` del elemento para devolverlo a la posición donde se supone que tiene que ir.

Te estarás preguntando para qué es la línea `$node.offset()`. ¿Para qué obtenemos la posición de `$node` si no vamos a hacer nada con ella?

Míralo así: si le dices a un robot muy inteligente que se mueva al norte 5 kilómetros, y luego al sur otros 5, probablemente sabrá deducir que va a terminar en el mismo sitio, y que puede ahorrar energía y hacer bien el trabajo sin moverse.

Así que si quieras que el robot ande 10 kilómetros, le diremos que mida sus coordenadas a los 5 kilómetros, antes de que de la vuelta.

El navegador funciona de una manera similar: si le damos las instrucciones `css('top', oldTop - newTop)` y `css('top', 0)` a la vez, las nuevas coordenadas reemplazarán las viejas y no pasará nada. Si queremos ver la animación, debemos forzar al navegador a redibujar el elemento después de moverlo la primera vez.

Una forma sencilla de hacerlo es pedirle al navegador el `offset` del elemento.

Vamos a probar de nuevo. Volvamos a la vista “Best” y votemos unos posts: ¡Deberías verlos moviéndose suavemente hacia arriba y hacia abajo como en un ballet!

Animated reordering

Added post reordering animation.

[Ver en GitHub](#)

[Lanzar instancia](#)

Ahora que ya tenemos resuelta la reordenación más complicada, animar las inserciones y eliminaciones va ser muy sencillo.

Primero, haremos aparecer nuevos posts (esta vez, por simplicidad, usaremos animaciones JavaScript):

```
Template.postsList.onRendered(function () {
  this.find('.wrapper')._uihooks = {
    insertElement: function (node, next) {
      $(node)
        .hide()
        .insertBefore(next)
        .fadeIn();
    },
    moveElement: function (node, next) {
      //...
    }
  };
});
```

Para ver el resultado, podemos probar a insertar un post vía consola:

```
Meteor.call('postInsert', {url: 'http://apple.com', title: 'Testing Animations'})
```

Fading in new posts

Y ahora, haremos desaparecer los posts eliminados:

```
Template.postsList.onRendered(function () {
  this.find('.wrapper')._uihooks = {
    insertElement: function (node, next) {
      $(node)
        .hide()
        .insertBefore(next)
        .fadeIn();
    },
    moveElement: function (node, next) {
      //...
    },
    removeElement: function (node) {
      $(node).fadeOut(function () {
        $(this).remove();
      });
    }
  );
});
```

De nuevo, para ver el efecto, prueba a eliminar algún post desde la consola (`Posts.remove('algúnPostId')`).

Fading out deleted posts

Fade items in when they are drawn.

[Ver en GitHub](#)

[Lanzar instancia](#)

Hemos creado animaciones para elementos *dentro* de una página. Pero, ¿qué pasa si queremos animar las transiciones entre páginas?

Las transiciones entre páginas son trabajo del Iron Router. Haces click en un enlace y se reemplaza el contenido del

```
ayudante {{> yield}} en layout.html
```

Ocurre que, como cuando reemplazamos el comportamiento de Blaze para la lista de posts, ¡podemos hacer lo mismo para el elemento {{> yield}} y añadirle un efecto de transición entre rutas!

Si queremos animar la entrada y salida entre dos páginas, debemos asegurarnos de que se muestran una por encima de la otra. Lo hacemos usando la propiedad `position: absolute` en el contenedor `.page` que envuelve a todas las plantillas de páginas.

Piensa que no queremos que las páginas estén posicionadas de forma absoluta, porque de esta forma, se solaparían con la cabecera de la app. Así que establecemos la propiedad `position: relative` en el div `#main` que las contiene, de forma que el `position: absolute` de `.page` tome su origen desde `#main`.

Para ahorrar tiempo, hemos añadido el código necesario a `style.css`:

```
//...
#main{
  position: relative;
}
.page{
  position: absolute;
  top: 0px;
  width: 100%;
}
//...
```

client/stylesheets/style.css

Es el momento de añadir el código para las transiciones entre páginas. Nos debe resultar familiar, puesto que es exactamente el mismo que para las inserciones y eliminaciones de posts:

```
Template.layout.onRendered(function() {
  this.find('#main')._uihooks = {
    insertElement: function(node, next) {
      $(node)
        .hide()
        .insertBefore(next)
        .fadeIn();
    },
    removeElement: function(node) {
      $(node).fadeOut(function() {
        $(this).remove();
      });
    }
  );
});
```

client/templates/application/layout.js

Transitioning in-between pages with a fade

Transition between pages by
fading.

[Ver en GitHub](#)

[Lanzar instancia](#)

Hemos visto unos pocos patrones para animar elementos en nuestra aplicación Meteor. Aunque no es una lista exhaustiva, con suerte, nos aportará una base sobre la que construir transiciones más elaboradas.

Esperamos que con la lectura de los capítulos anteriores tengas una buena visión general de todo lo que involucra la construcción de una aplicación Meteor. Entonces, ¿dónde podemos ir ahora?.

En primer lugar, puedes comprar las ediciones **Full** o **Premium** para desbloquear el acceso a los capítulos adicionales. Estos capítulos te guiarán a través de escenarios del mundo real, tales como la construcción de una API para tu aplicación, la integración con servicios de terceros o la migración de datos.

Además de contar con la [documentación](#) oficial, el [Manual de Meteor](#) profundiza en temas específicos como Tracker o Blaze.

Si quieres sumergirte en los entresijos de Meteor, te recomendamos echarle un vistazo a [Evented Mind](#) de Chris Mather, una plataforma de aprendizaje con más de 50 vídeos sobre Meteor (y nuevos vídeos que se agregan cada semana).

Una de las mejores formas de mantenerse al día con Meteor es suscribirse al boletín semanal de Arunoda Susiripala [MeteorHacks](#). En el blog también puedes encontrar un montón de consejos avanzados sobre Meteor.

[Atmosphere](#) es el repositorio de paquetes no oficiales de Meteor, es otro gran lugar para aprender más: puedes descubrir nuevos paquetes y echar un vistazo a su código para ver qué patrones utiliza la gente.

(Aviso legal: Atmosphere es mantenida, en parte por Tom Coleman, uno de los autores de este libro).

[Meteorpedia](#) es un wiki sobre Meteor. Y, por supuesto, ¡está hecho con Meteor!

Otra iniciativa de Arunoda de MeteorHacks, [BulletProof Meteor](#) te guiará a través de lecciones con preguntas tipo test, y enfocadas al rendimiento de Meteor.

Josh y Ry de la empresa [differential](#) graban el [Podcast Meteor](#) todas las semanas, otra forma de mantenerse al día con lo que pasa en la comunidad Meteor.

Stephan Hochhaus ha compilado una lista bastante exhaustiva de [recursos Meteor](#).

El blog de [Manuel Schoebel](#) y el de [Gentlenode](#) son una buena fuente de información sobre Meteor.

Si encuentras algún obstáculo, el mejor lugar para preguntar es [Stack Overflow](#). Asegúrate de etiquetar la pregunta con la etiqueta `meteor`.

Por último, la mejor forma de estar al día con Meteor es mantenerse activo en la comunidad. Nosotros recomendamos inscribirse en la [lista de correo](#) de Meteor, seguir los grupos de Google [Meteor Core](#) y [Meteor Talk](#) y crear una cuenta en el foro de Meteor [Crater.io](#).

Cuando hablamos del Cliente, nos referimos al código que se ejecuta en el navegador de los usuarios, ya sea uno tradicional, como Firefox o Safari, o algo tan complejo como un UIWebView en una aplicación nativa para el iPhone.

Una colección es el almacén de datos que se sincroniza automáticamente entre el cliente y el servidor. Las colecciones tienen un nombre (como `posts`), y por lo general existen tanto en el cliente como en el servidor. Si bien se comportan de forma distinta, tienen una API común basada en la API de Mongo.

Una computación es un bloque de código que se ejecuta cada vez que cambia una de las fuentes de datos reactivos de las que depende. Si tienes una fuente reactiva (por ejemplo, una variable de sesión) y quieres responder reactivamente a ella, tendrás que crear una computación.

Un cursor es el resultado de ejecutar una consulta en una colección Mongo. En el lado del cliente, un cursor no es tan sólo un conjunto de resultados, sino que es un objeto *reactivo* desde el que se puede observar (con `observe()`) los cambios (añadir, eliminar o actualizar) en la colección correspondiente.

El DDP es el Protocolo de Datos Distribuidos que utiliza Meteor para sincronizar colecciones y efectuar llamadas a métodos. DDP pretende ser un protocolo genérico, que toma el relevo a HTTP para aplicaciones en tiempo real con gran carga de datos.

Deps es el sistema reactivo de Meteor. Deps se utiliza entre bastidores para sincronizar automáticamente el HTML con el modelo de datos subyacente.

Mongo es un almacén de datos basado en documentos y a los objetos que salen de las colecciones se les llama “documentos”. Son objetos JavaScript sin formato (aunque no pueden contener funciones) con una única propiedad especial, el ‘_id’, que Meteor utiliza para realizar un seguimiento de sus propiedades en el DDP.

Cuando una plantilla necesita mostrar cosas más complejas que una simple propiedad de un documento, ésta puede hacer uso de su ayudante, una función que se utiliza para procesar los datos que se muestran en ella.

Es una técnica que permite simular llamadas a métodos en el cliente para evitar retrasos mientras se espera la respuesta del servidor.

La empresa que desarrolla Meteor.

Un método en Meteor es una llamada desde el cliente, a un procedimiento remoto en el servidor, con un poco de lógica añadida que permite realizar un seguimiento de los cambios en los datos además de compensar la latencia de la llamada.

La colección del lado del cliente es un almacén de datos en memoria que ofrece una API tipo Mongo. La librería que se utiliza se llama “**MiniMongo**”, para indicar que es una versión más pequeña de Mongo que se ejecuta por completo en la memoria del navegador.

Un paquete Meteor puede ser: código JavaScript que se ejecuta en el servidor, código JavaScript que se ejecuta en el cliente, instrucciones para procesar recursos (como SASS a CSS), o recursos que deben ser procesados.

Un paquete es como una librería con superpoderes. Meteor incluye una gran cantidad de paquetes (`meteor list`).

También existe **Atmosphere**, que es una colección de paquetes de terceros mantenida por la comunidad (`mrt add ...`).

Una publicación es un conjunto de datos con nombre que se personaliza para cada usuario que se suscribe a ella. Se configuran en el servidor.

El servidor Meteor es un servidor HTTP y DDP ejecutados vía Node.js. Se compone de todas las librerías y del código JavaScript del lado del servidor. Cuando se inicia el servidor, se conecta a una base de datos Mongo (que configura por sí mismo en el primer arranque).

La sesión en Meteor es una fuente de datos reactiva que usa tu aplicación para hacer un seguimiento del estado del usuario.

Una suscripción es una conexión a una publicación desde un cliente específico. La suscripción es el código que ejecuta el navegador y que utiliza para comunicarse con una publicación del servidor y que, además, mantiene los datos sincronizados.

Una plantilla es una forma de generar código HTML desde JavaScript. Por defecto, Meteor sólo soporta el sistema Spacebars, pero hay planes para incluir más.

Cuando se muestra un plantilla, lo que se representa es un objeto JavaScript que proporciona datos específicos para esta representación en particular. Por lo general, este tipo de objetos son, de tipo POJO (plain-old-JavaScript-objects), a menudo son documentos de una colección, aunque pueden ser más complejos e incluir funciones.

1.9

- Updated Introduction & Getting Started sections for Windows.
- Changed `rendered` to `onRendered` and `created` to `onCreated`.
- Explained package names in Getting Started chapter.
- Added note about default Iron Router help page in Routing chapter.
- Fixed `audit-argument-checks` link in Creating Posts chapter.
- Fixed file paths in Comments chapter.
- Wrong `limit()` changed to `postsLimit()` in Pagination chapter.
- Changed `UI.registerHelper` to `Template.registerHelper` in Voting chapter.
- Added a note about `.animate` CSS class in Animations chapter.
- Fixed file paths in Animations chapter.

1.8

- Rewrote animation chapter to use `_uihooks`.
- Wrapped every page in a `.page` div.
- Used the official `twbs:bootstrap` Bootstrap package.
- Added `.page` CSS to `style.css`.
- Used `Template.registerHelper` instead of `UI.registerHelper`.
- Removed Deploying On Modulus section (now referencing their docs instead).
- Updated `db.users.find()` result in “Adding Users” chapter.
- Added a note about the Meteor shell in the “Collections” chapter.

1.7.2

- Adding paragraph about `subscriptions` in Pagination chapter.
- Various typo fixes.
- Various code fixes.

1.7.1

Various fixes.

- Fix code highlighting in Voting chapter.
- Change “router” to “route” in Pagination chapter.
- Removed mentions of `Router.map()` in Comments and Pagination chapters.
- Linking to Boostrap site in Adding Users chapter.

- Added BulletProof Meteor to Going Further chapter.

1.7

Updating the book for Iron Router 1.0.

- Defining routes with new path-first syntax.
- Use `subscriptions` instead of `onBeforeAction` in posts controller.
- Use `this.next()` in `onBeforeAction` hook.
- Rename `XyzPostsListController` to `XyzPostsController`.

1.6.1

- Fixing a few typos.
- Finishing switching `Meteor.Collection` to `Mongo.Collection`.
- Updated introduction.
- Added “Get A Load Of This” section in Routing chapter.

1.6

Updating the book for Meteor 1.0.

- `collections` directory is now in `lib`.
- `views` directory is now named `templates`.
- Removed `$` from bash prompts.
- Now using Bootstrap 3.
- Being more consistent about using `//...` to denote skipped lines in code.

- Explained the advantages of Meteor packages over manually adding files.

- Explicitly adding `underscore` package.
- Updated “5 Types of Packages” section.
- Not creating `collections` directory anymore.
- Updated CSS code.

- Changed “partials” to “inclusions”.

- Not talking about “managers” anymore.

- Cut down Collections chapter intro.
 - Changed `Meteor.Collection()` to `Mongo.Collection()`.
 - Added “Storing Data” section.
 - General edits and tweaks.
-
- Added “Post Not Found” section.
 - General edits and improvements.
-
- Added reminder to revert code changes at the end of the chapter.
-
- Now using `ian:accounts-ui-bootstrap-3` package.
-
- Using `Trackers` instead of `Deps`.
-
- Removed `message` field from posts.
 - Added “Security Check” section.
 - Added “Preventing Duplicates” section.
 - Changed `post` to `postInsert`, updated `postInsert` method description.
-
- Updated code examples.
 - Added more explanations.
-
- Remove “Using Deny as a Callback” section.
-
- Completely rewrote error handling code.
 - Added “Seeking Validation” section.
-
- Various edits and updates.
-
- Rename `comment` template to `commentItem`.

- Various edits.
- Added “No Trespassers Allowed” note.
- Added section about `reactive-var` package.
- Got rid of `iron-router-progress`.
- Various edits.
- Various edits.

This chapter is out of date. Update coming sometimes after 1.0.

Note: the following extra chapters are only included in the Full and Premium editions:

- Updated package syntax.
- Minor tweaks.
- Minor edits.
- Added `favorite_color` custom attribute.
- Various minor edits.
- Minor edits.

1.5.1

- Fix quotes in comments chapter.
- Clarified Session chapter.
- Added link to the blog in chapter 8.

- Adding a note about reversing changes at the end of Session sidebar.
- Reworking section about the five types of packages.
- Changing “partial” to “inclusion” in Templates chapter.
- Added note about Animations chapter being out of date.

1.5

- Updated Pagination chapter.
- Fixed typos.
- Removed mentions of Meteorite throughout the book.
- Updated Creating A Package sidebar for Meteor 0.9.0.
- Now including changelog in book repo.
- Book version is now tracked in changelog, not in intro chapter.
- Added section about manual.meteor.com in Going Further chapter.

1.3.4

- Replaced Vocabulary chapter with **Going Further** chapter.
- Added new Vocabulary sidebar.

1.3.3

- Various typos and highlighting fixes.
- Small correction in **Errors** chapter.

1.3.2

Various typos fixes.

1.3.1

Finished 0.8.0 Update.

- **12 – Pagination:** Use `count()` instead of `fetch().length()`.
- **14 – Animations:** Rewrote the chapter to use a helper instead of the `rendered` callback.

1.3

Updated to support Meteor 0.8.0 and Blaze.

- **5 – Routing:** Routing changes to support IR 0.7.0:
 - `{{yield}}` becomes `{{> yield}}`
 - Explicitly add `loading` hook.
 - Use `onBeforeAction` rather than `before`.

- **6 – Adding Users:** Minor change for Blaze:
 - `{{{loginButtons}}}` becomes `{}{> loginButtons}}`
- **7 – Creating Posts:**
 - HTML changes for stricter parser.
 - Update our `onBeforeAction` hook to use `pause()` rather than `this.stop()`
- **13 – Voting:** Small change to the `activeRouteClass` helper.

1.2

The first update of 2014 is a big one! First of all, you'll notice a beautiful, photo-based layout that makes each chapter stand out more and introduces a bit of variety in the book.

And on the content side, we've updated parts of the book and even written two whole new chapters:

- **[NEW!] 3.5 – Using GitHub:** New sidebar on how to use GitHub.
- **[NEW!] 17.5 – Migrations:** New sidebar about database migrations.
- **2.5 – Deploying:** Rewrote chapter from scratch to be more up to date.
- **4.5 – Publications and Subscriptions:** Merged in content from [Understanding Meteor Publications & Subscriptions](#)
- **15 – RSS Feeds & APIs:** Updated chapter to use Iron Router.
- **16 – Using External APIs:** Updated chapter to use Iron Router.
- **17 – Implementing Intercom:** Rewrote chapter to match the [Intercom package](#).

1.1

- **5 – Routing:** Rewrote chapter from scratch to use [Iron Router](#).
- **5.5 – The Session:** Added a section about Autorun.
- **10 – Comments:** Updated chapter to use IR.
- **12 – Pagination:** Rewrote chapter from scratch, now managing pagination with IR.
- **13 – Voting:** Updated the chapter to use IR, simplified the template structure.

Minor updates include API changes between the old Router and Iron Router, file paths updates, and small rewordings.

- **6 – Adding Users**
- **7 – Creating Posts**

- **7.5 – Latency Compensation**

- **8 – Editing Posts**

- **9 – Errors**

- **11 – Notifications**

- **12 – Animations**

If you'd like to confirm what exactly has changed, we've created a [full diff of our Markdown source files \[PDF\]](#)

1.02

- Various typo fixes

1.01

- Updated “Creating a Meteorite Package” chapter to Meteor 0.6.5
- Updated package syntax in Intercom and API extra chapters.

1.0

First version.