BORJA BARRERA VILLAGRASA

Trabajo de Teoría en colaboración con José Alberto Mena García para la asignatura de Sistemas y Tecnologías Web de cuarto de carrera del Grado de Ingeniería Informática de la Universidad de la Laguna

INDICE

1. Introducción

1.1 ¿Qué es METEOR?

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 abstrae muchas de las molestias y dificultades que nos encontramos habitualmente en el desarrollo de aplicaciones web.

1.2 ¿Por qué METEOR?

Meteor es “full stack” y es fácil de aprender.

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 nos permite escribir aplicaciones web más eficientes, versátiles y modernas. La comunidad, sin ser todavía amplia si que es lo suficientemente grande como para que existan bastantes librerías y utilidades, además de recursos y documentación.

Para lograr esto METEOR se basa en siete principios fundamentales:

1. Data on the wire: No se manda porciones de Html sobre la red, se mandan datos al cliente (plantilla) y es él el que decide como los visualiza.

2. One language: Todo el código tanto en el lado del cliente como en el servidor se escribe en JavaScript (Aunque se soporta CoffeeScript por medio de un paquete).

3. Database everywhere: Utiliza de igual forma el API de acceso a la base de datos tanto en el cliente como en el servidor.

4. Latency compensation: En el cliente se simula la interacción para que se vea como si no hubiera tiempo de latencia en el acceso a la base de datos.

5. FULL STACK reactivity: Actualiza en tiempo real automaticamente, toda la información desde la base de datos y la sesión del usuario al sistema de plantillas esto es realizado por su sistema de orientación a eventos que escucha y replica cambios en la información.

6. Embrace the ecosystem: Meteor no pretende reinventar la rueda, por lo tanto animan al usuario a que integre otras herramientas existentes, por lo tanto es muy común ver un uso de meteor complementado por otros frameworks existentes que facilitan el desarrollo.

7. Simplicity equals productivity: La mejor manera de que algo parezca simple es que en realidad sea simple, esto se logra con una API limpia que simplifica el código y por lo tanto aumenta la productividad.

2. Empezando.

2.1 Instalación.

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

Por el contrario, si estás usando Windows, echa un vistazo a la guía oficial de instalación: **install instructions** (https://www.meteor.com/install) en la web de Meteor.

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

> Sin instalar Meteor :

Si no podemos (o no queremos) instalar Meteor de forma local, recomendamos usar **Nitrous.io**. (https://www.nitrous.io/)

2.2 Creando una simple aplicación.

Cuando tengamos instalado METEOR, vamos a crear nuestra aplicación. Para ello, utilizaremos la herramienta de línea de comandos meteor:

meteor create <Nombre>

Con ello crearemos un proyecto básico listo para usar. Cuando termina deberíamos ver un directorio llamado con el nombre que le dimos cuando ejecutamos el comando, que contiene una serie de ficheros, como por ejemplo:

.meteor

microscope.css

microscope.html

microscope.js

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

Si queremos ejecutar la aplicación vamos a una terminal y escribimos:

cd microscope *--> directorio donde se encuentran los fichero mencionados anteriormente*

meteor *--> comando para ejecutar la aplicación.*

Ahora abrimos **localhost:3000** en el navegador y deberíamos ver algo como esto:

2.3 Añadir un paquete

A modo de ilustrar de como se añaden paquetes a METEOR, añadiremos el paquete del framework **bootstrap** y el paquete **Underscore** (*una librería de utilidades JavaScript, que 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

2.4 Estructura de una aplicación Meteor.

Antes de empezar a escribir código debemos estructurar de forma adecuada nuestro proyecto. Para asegurarnos de que disponemos de un entorno limpio y claro, abrimos el directorio donde tenemos los ficheros (*en nuestro caso "**microscope**"*) y borra los archivos:

microscope.html,

microscope.js,

microscope.css.

A continuación, crea cuatro directorios dentro de **/microscope**:

/client

/server

/public

/lib.

Ahora, creamos los siguientes directorios vacíos.

/client

main.html

main.js

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.

2.5 CSS de nuestra aplicación

En este tutorial no vamos a tratar de 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; }
```

3. Despliegue.

*3.1 Despliegue de la aplicación en **Meteor.com**.*

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

3.1 Despliegue de aplicaciones en Meteor.com

Desplegar en un subdominio de Meteor es la opción más sencilla. 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.

4. Plantillas.

4.1 Las plantillas en Meteor

4.2 Ayudantes de plantillas

*4.3 El ayudante **domain***

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

Esto implica que, solo utilizaremos el directorio **`/client`.**

Primero, si todavia no lo tienes creado, creamos un nuevo archivo **`main.html`** dentro del directorio client, rellenándolo con el siguiente código:

```
<head>
  <title>Micro</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.

4.1 Las plantillas en Meteor

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.

```
/client
 /templates
  /posts
...
```

Ahora, 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.

> NOTA:

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}}.**

Con los conocimientos sobre los Spacebars, 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**.

El objeto post 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.

4.2 Ayudantes de plantillas

Hasta ahora hemos estado tratando con Spacebars, que es poco más que HTML con algunas etiquetas extra. Pero para que una plantilla tenga vida, necesita ayudantes.

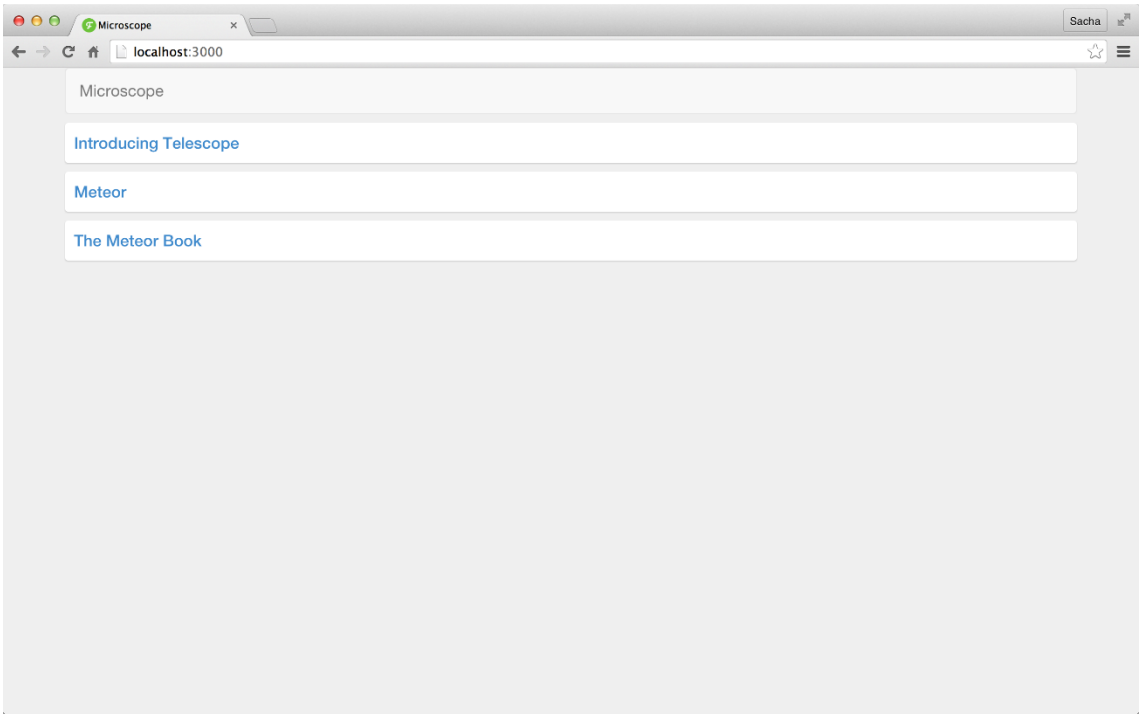
Es decir, mientras la función de las plantillas es mostrar o iterar sobre variables, los ayudantes son los que hacen el trabajo pesado asignando un valor a cada variable.

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
 });
```

>client/templates/posts/posts_list.js

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



Lo que estamos haciendo es 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, 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>
```

>client/templates/posts/posts_list.html

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.

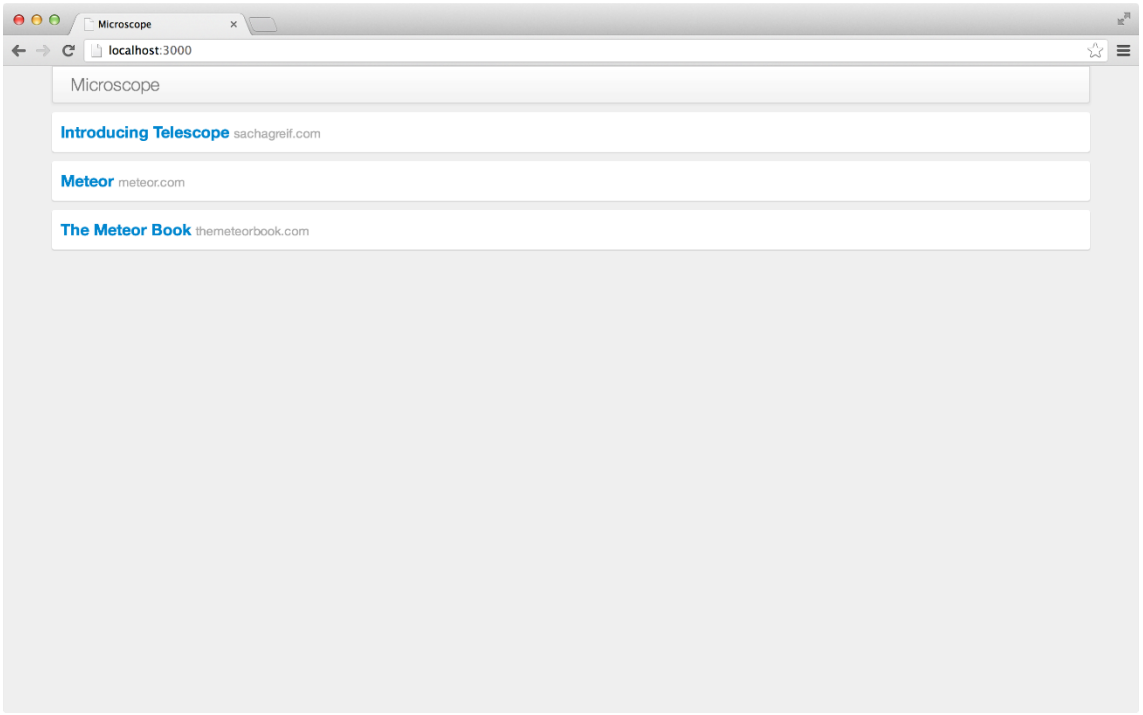
4.3 El ayudante domain

Ahora, 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.



>NOTA:

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?*.

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.

5. Colecciones.

En el apartado **(1) "Introducción"** hablamos sobre la sincronización automática de datos entre cliente y servidor.

En este apartado miraremos más de cerca 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

5.1 Colecciones en el lado del 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

5.2 Colecciones 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

En segundo lugar, los documentos se almacenan en la memoria del navegador, lo que significa que el acceso a ellos es prácticamente instantáneo.

5.3 Comunicación cliente-servidor

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.

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"))**.

Escribamos esto en la consola del segundo navegador:

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

>Consola del segundo navegador

A pesar de que no hemos refrescado ni interactuado con el segundo navegador, y desde luego no hemos escrito código para insertar actualizaciones.

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`**.

5.4 Rellenando la base de datos

En este apartado, 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.

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

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.

5.5 Datos dinámicos

Si abrimos una consola de navegador, veremos los tres mensajes cargados desde ***MiniMongo*** (*Es la implementación de Mongo en el lado del cliente de Meteor*):

```
> Posts.find().fetch();
```

>Consola del navegador

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

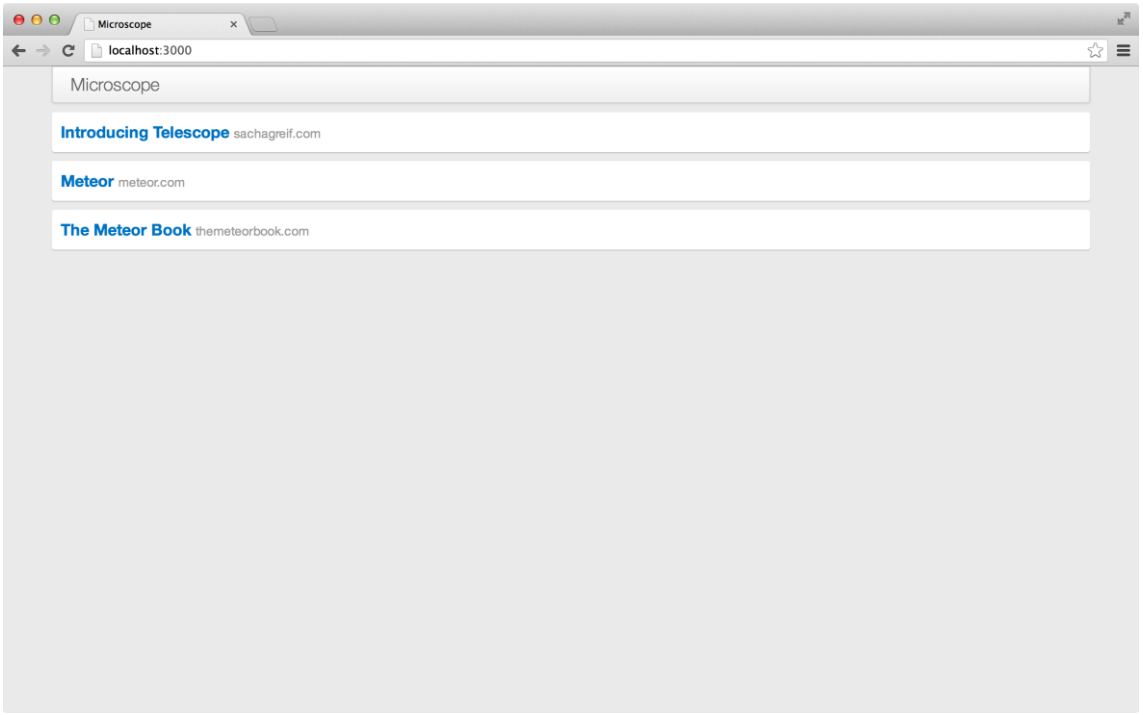
Simplemente reemplazamos el objeto JavaScript estático **`postsData`** por una colección dinámica.

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

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



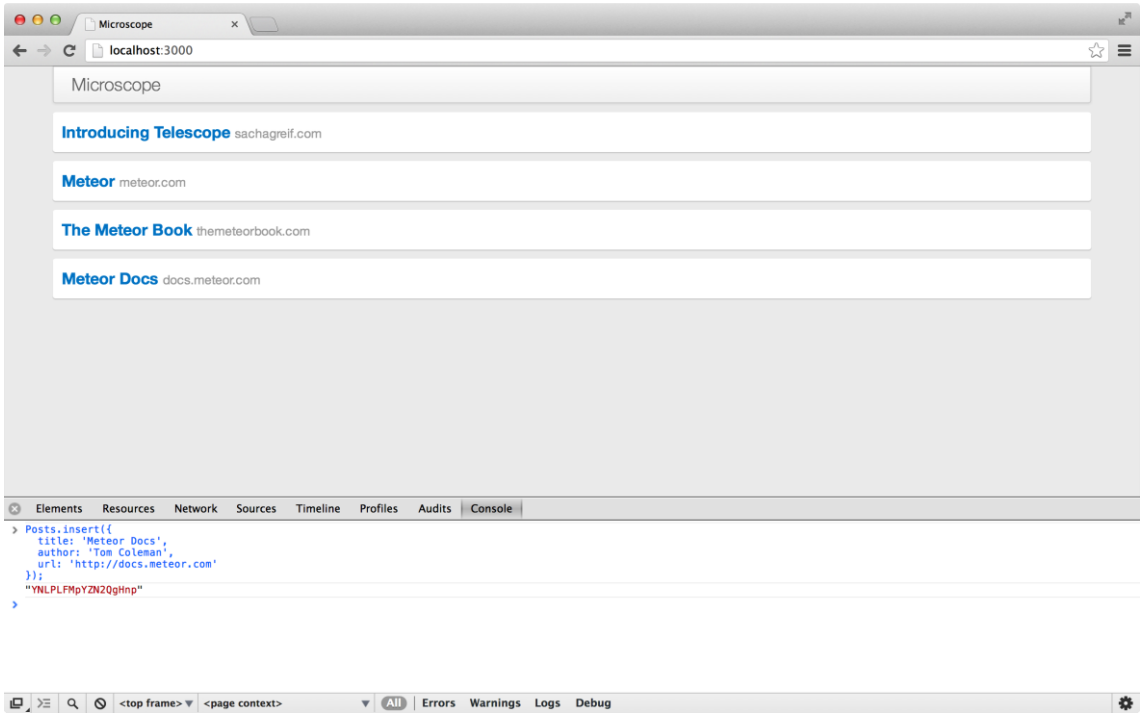
Nuestro ayudante **`{{#each}}`** ha recorrido todos nuestros **`Posts`**, y los ha mostrado en la pantalla. La colección del lado del servidor ha tomado los posts de *Mongo*, los ha pasado a nuestra colección del lado del cliente, y nuestro ayudante *Spacebars* los ha pasado a la plantilla.

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:



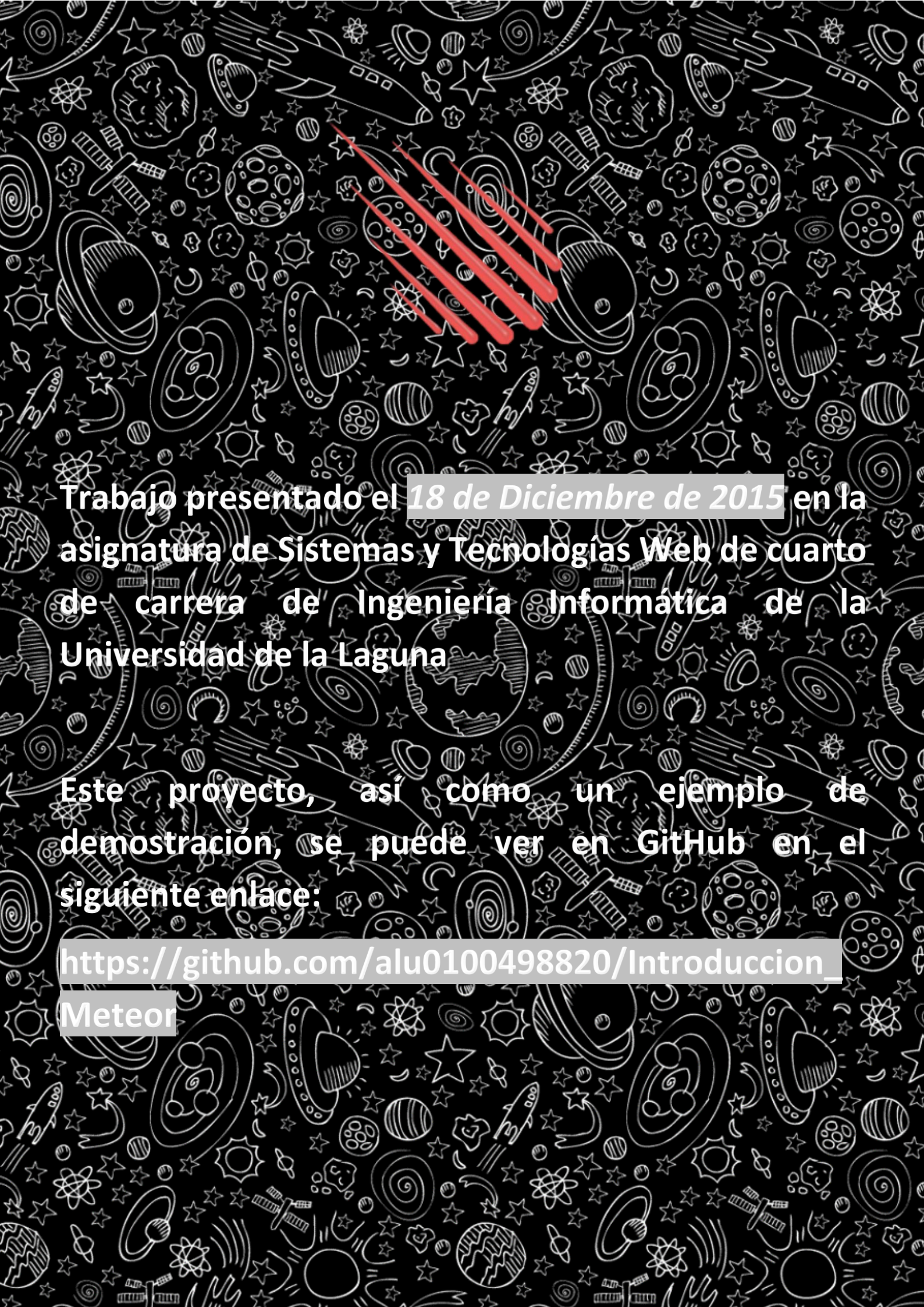
Acabas de ver la reactividad en acción. Cuando le pedimos a *Spacebars* que recorra el cursor **`Posts.find()`**, él ya sabe cómo monitorizar este cursor en busca de cambios, y de esa forma, alterar el código HTML para mostrar los datos correctos en la pantalla.

Y con esto terminamos nuestra tutorial de introducción al framework METEOR.

Si desean profundizar más sobre este framework les animamos a que consulten la bibliografía

6. Bibliografía

- https://www.meteor.com/
- http://docs.meteor.com/#/basic/
- https://github.com/meteor
- http://es.discovermeteor.com/
- https://en.wikipedia.org/wiki/Meteor_(web_framework)
- http://slides.com/edsadr/aplicaciones-en-tiempo-real-con-meteor#/



Trabajo presentado el *18 de Diciembre de 2015* en la asignatura de Sistemas y Tecnologías Web de cuarto de carrera de Ingeniería Informática de la Universidad de la Laguna

Este proyecto, así como un ejemplo de demostración, se puede ver en GitHub en el siguiente enlace:

https://github.com/alu0100498820/Introduccion_Meteor