ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación :

INGENIERO TÉCNICO DE TELECOMUNICACIÓN, ESPECIALIDAD EN SONIDO E IMAGEN

Título del proyecto:

DESARROLLO DE UN FRAMEWORK PARA LA CREACIÓN DE APLICACIONES MÓVILES CON TECNOLOGÍAS WEB

MEMORIA

Alumno: Aitor Osés Martín

Tutor: Marko Galarza

Fecha: Pamplona, 20 de Abril de 2014

INDICE

Tabla de contenido

1. OBJETIVO DEL PFC

El fin de este proyecto final de carrera es crear un marco de trabajo dentro del desarrollo de aplicaciones de internet y aplicaciones móviles con tecnologías web. Actualmente, en el mundo de HTML5 las prácticas para poder crear una aplicación exitosa, y a su vez escalable, están lejos de ser un estándar por lo que el trabajo desarrollado en el presente proyecto busca crear un estándar o convenio para el desarrollo de aplicaciones web.

La principal característica técnica que cumplirán las aplicaciones creadas con este *framework* (marco de trabajo) es ofrecer un patrón de programación reactivo, es decir, que las aplicaciones responderán en tiempo real a todos los cambios internos. Por ejemplo, cuando las fuentes de datos sufran un cambio en un instante, automáticamente se actualizará la interfaz de otros usuarios para reflejar estos cambios.

Los puntos fuertes que se estudiarán para el desarrollo de este framework son:

- El intercambio de información entre usuarios y servidores.
- Facilitar una experiencia nativa móvil.
- Tener una arquitectura escalable.

A su vez puntos se estudiarán desde 3 perspectivas

- La parte **Frontend** (cliente).
- La parte **Backend** (Servidor).
- La **comunicación** entre ambos.

El producto de este PFC, será el conjunto de estas librerías unificadas listo para utilizarse en producción.

Este proyecto de software será expuesto con una licencia MIT[1] de código abierto en GitHub[2] para poder ser compartido y utilizado por cualquier persona, para que otros puedan contribuir y mejorar el proyecto. También será instalable mediante el gestor de paquetes NPM[3] de *node*.

El nombre final del producto será **Evented.io** y será instalable a través de un gestor de paquetes.

[1] MIT License es un tipo de licencia OpenSource sin CopyRight.
[2] GitHub es una comunidad social de software OpenSource.
[3] **NPM**, Node Package Manager, *véase* http://www.npmjs.org/~grayfox

2. Estado del arte

2.1 Aplicaciones móviles

Cuando nos referimos a **web móvil** hablamos de un concepto de web en el que el usuario accede a la información desde cualquier punto y con cualquier tipo de dispositivo.

El crecimiento del sector móvil (smartphone, tabletas, dispositivos de diferentes resoluciones y tamaños) en los últimos años, esta forzando una demanda por el usuario final de la web que exige que las webs se muestren de forma incondicional a su formato.

Las **aplicaciones móviles** generalmente suelen estar codificadas en un lenguaje de programación y librerías ofrecidas por el proveedor del sistema operativo (Android, iOS, etc..).

Estas aplicaciones suelen tener un comportamiento muy responsivo y tienden a tener unas interfaces con un **estilo visual** y **animaciones** muy atractivo que simulan la física del movimiento en los distintos elementos. Una **aplicación web** en cambio, puede ser desarrollada con estos objetivos en mente, o se puede diseñar a base de componentes de la misma manera que está ideada una aplicación nativa, pero esto requiere que sea el navegador quien ejecute la aplicación y no el servidor quien procese una petición para servir un contenido estático y poco **interactivo.**

Las páginas web no muestran este comportamiento por norma general debido a limitaciones impuestas por el software. Además las aplicaciones nativas están aceleradas por Hardware, por la **GPU**[4], y los navegadores web hasta ahora solo han utilizado la CPU por lo que son mas lentos a la hora de realizar cálculos complejos como lo son el renderizado y animación de una página web. Aun así, ya han aparecido técnicas que fuerzan el uso de la GPU pese a no estar estandarizadas.

El problema en el mundo de la Web, es que las prácticas están poco estandarizadas y eso es lo que este PFC pretende hacer, crear un **convenio** para el desarrollo de aplicaciones, pero con tecnologías web.

[4] GPU, unidad de procesado gráfico, comúnmente conocido como el procesador de la tarjeta gráfica. Al contrario que la CPU, que es de propósito general, está optimizado para el cálculo de coma flotante, por lo que es apropiado para el calculo de “renders” y operaciones de programación gráfica, donde se trabaja mucho con matrices.

2.2 Tecnología

2.2.1 Servidor

La arquitectura en el lado del servidor que vamos a tomar se diferencia un poco de lo habitual. Normalmente hablaríamos de una base de datos y un servidor de páginas web. En este caso hablamos de un servidor de **API**'s y **Websockets**[5] **(sockets web)**, en general conocidos como Servicios Web, que nuestra aplicación se encargará de consumir.

La arquitectura común (antigua) desde que empezó internet está orientada a páginas web, donde el navegador mediante el protocolo http, ejerce de consumidor de unos recursos, conocidos como páginas web o páginas HTML.

Ilustración 1. Arquitectura común o básica

[5] **WebSocket** es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP, diseñado para ser implementado en navegadores y servidores web.

La cuestión es que este esquema, no se adapta correctamente al concepto de uso que tenemos de un teléfono móvil. Con el **boom de los smartphone** y el uso de las aplicaciones móviles, el usuario final está acostumbrado a tener una experiencia de usuario como las que ofrecen las aplicaciones mas utilizadas (Facebook, Google Apps, Youtube…).

La arquitectura expuesta anteriormente tiene dos problemas principalmente:

1. **El servidor procesa el contenido que se sirve a todas las peticiones**. Teniendo en cuenta la creciente cantidad de usuarios de las aplicaciones web, esto supone una carga considerable para el servidor.

   En las aplicaciones nativas, esto no pasa ya que la aplicación se encarga del contenido y el servidor solo procesa los datos de entrada y salida.

   Por tanto, necesitamos una nueva arquitectura.

2. **El contenido es estático.** El contenido de la página llega ***prerenderizado***[6] y el cliente suele tener poca *interactividad*.

   Además, el tiempo de respuesta depende del ping y del tiempo de procesado del servidor. Por lo tanto, si hay congestión de red, el usuario será incapaz de ver nuevo contenido de una forma rápida.

   Habrá que tomar la misma solución que toma una aplicación nativa, es decir, tomar datos y renderizarlos activamente, generando el contenido en el lado del cliente en base a datos.

[6] **Renderizar**, es la acción de generar el contenido gráfico de una aplicación. Renderizar datos es poblar una plantilla con datos existentes y generar el contenido exacto que se va a visualizar.

2.2.2 Cliente

Hemos considerado las limitaciones que tiene el uso de la tecnología que se ha utilizado hasta ahora en la parte del servidor. Ahora veremos alguno de los problemas que tiene la parte de cliente frente a una aplicación nativa.

El principal problema de construir una aplicación web es el hecho de que parece que siempre tenemos que partir de cero.

Siempre comenzamos con una **pagina en blanco** (index.html), le añadimos unos **scripts JavaScript** (scripts.js), y una **hoja de estilos CSS**, comenzamos a crear una nueva aplicación.

Siempre nos encontramos con que tenemos que resolver la misma clase de problemas:

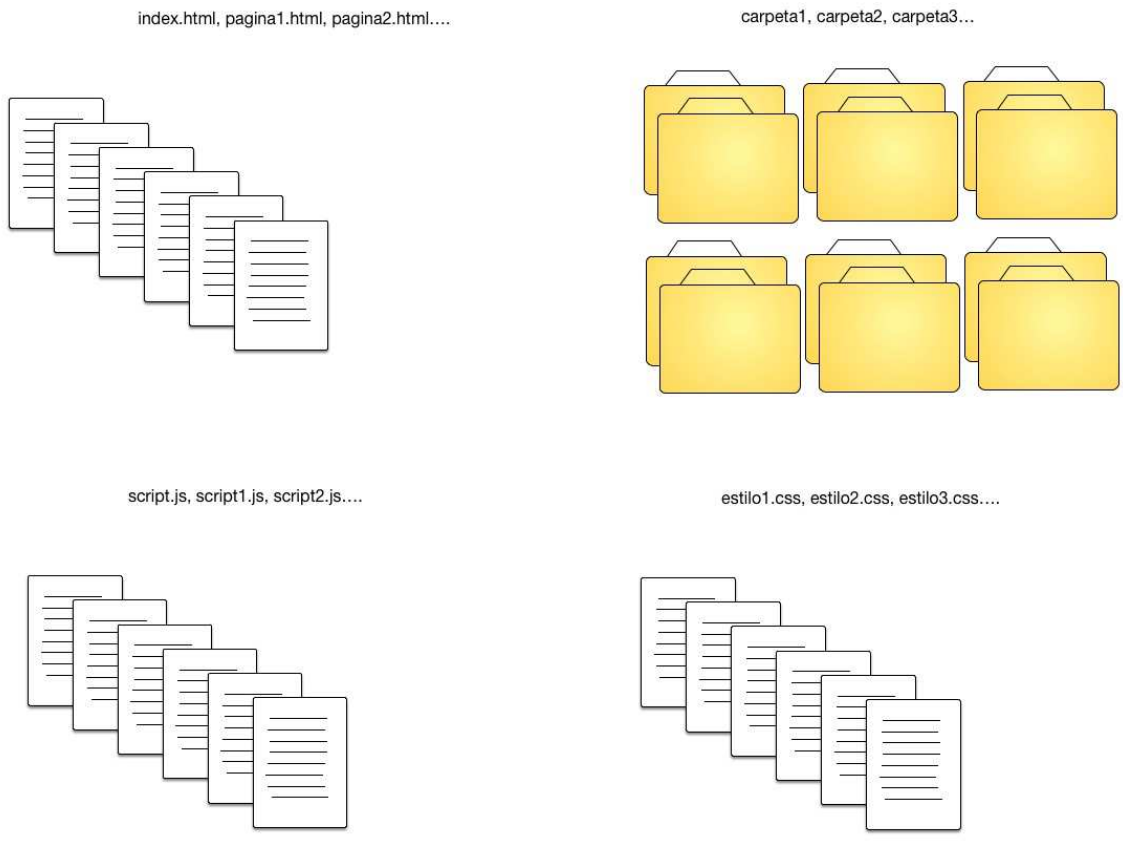


Ilustración 2. ¿Cómo ordenar los archivos y directorios de un proyecto?

1) Cómo organizar la estructura de la aplicación
2) Cómo modularizar el código: distintos scripts, un solo y largo script…
3) Cómo mantener limpio el proyecto…
4) Cómo conseguir que si cometemos un error en los cimientos de la aplicación, al realizar un cambio, no estropeemos el funcionamiento de la aplicación entera…
5) Cómo conseguir que la aplicación tenga un buen rendimiento. Recordemos que es un navegador web…

Es decir, son todos problemas básicos de arquitectura y de funcionalidades que

vamos a necesitar en prácticamente todos los proyectos en los que trabajemos.

Al final, si intentamos dar una solución partiendo de cero, el resultado suele ser:

- Una organización ininteligible
- Un código incomprensible
- Y una escalabilidad muy reducida
- Sobretodo, a la hora de trabajar en equipo, estos problemas conducirán a un proyecto poco exitoso.

Es decir, **necesitamos** una plantilla de proyecto de base, que tenga estos problemas comunes a todos los proyectos resueltos de una manera elegante.

2.2.3 Comunicación

Por otro lado, tenemos la comunicación entre los dispositivos de toman parte en este proyecto.

Utilizan el protocolo HTTP, que a su vez es un protocolo de red basado en el paradigma **"cliente-servidor".**

El protocolo HTTP consiste básicamente en que un cliente inicie una comunicación con un servidor mediante TCP (un socket), haga una petición y el servidor le conteste y después cierre la conexión. Es un protocolo sencillo y se adaptó fabulosamente a lo que internet era en sus comienzos, contenido estático.

Una aplicación web, al utilizar un navegador, también se ve obligada a utilizar este protocolo, no siendo posible la comunicación mediante TCP. Esto supone una limitación porque cuando el cliente necesita interactividad, tiene que iniciar conexiones de este tipo cada vez que necesita algo del servidor. Además el **handshake**[7] de HTTP genera un tiempo de respuesta bastante notable para el usuario final.

Si comparamos la comunicación directa de un socket TCP, que solo necesita establecer la conexión una vez, el claro vencedor son los sockets TCP.

El otro problema que tiene HTTP, es que la comunicación siempre la comienza el cliente. Si el cliente tiene que estar constantemente preguntándole al servidor si existen novedades, una cantidad excesiva de usuarios, perjudicarán enormemente a la congestión del servidor y esto no es para nada deseable. Es necesario hallar alguna técnica para que el servidor también sea capaz de iniciar conversaciones con sus clientes.

Por todo esto, debemos encontrar otra forma de comunicación distinta de HTTP, el cual está bien para ciertas tareas, pero claramente no termina de adaptarse a la funcionalidad que exige una App móvil.

[7] **Handshake** (apretón de manos), es el proceso de iniciar la comunicación en un protocolo. "¿Estas preparado? – Si, lo estoy – Necesito esto", es decir, para hacer una petición se necesitan al menos 3 RTT, y 1 RTT ~= 200 ms para una conexión normal. Es decir, mas de medio segundo de respuesta para que el servidor empiece a procesar la petición

3 Framework Evented.io

Evented.io es el nombre del marco de trabajo que he construido para dar solución a los problemas expuestos anteriormente.

La **arquitectura** que expone este framework es la siguiente:

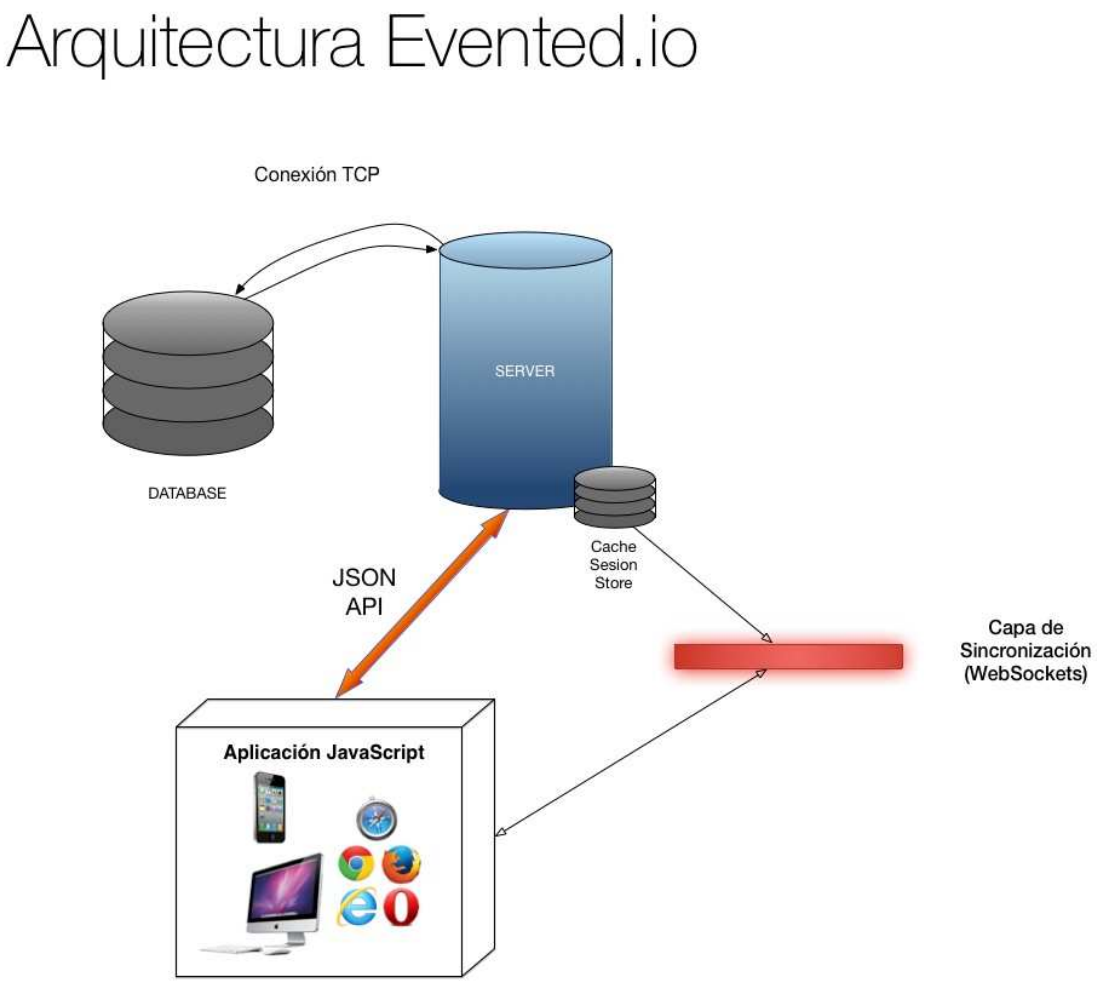


Ilustración 3. Arquitectura de Evented.io

I. Una base de datos NoSQL
II. Un servidor de API's
III. Un almacén de cache para mejorar el rendimiento y recolectar estadísticas
IV. Una capa de sincronización basada en la tecnología de websockets
V. Una aplicación Web, escrita puramente en JavaScript que se ejecuta en el navegador

La idea general es tener una aplicación JavaScript que se ejecute en el navegador y que sea capaz de realizar llamadas asíncronas a una **API**[8] expuesta por el servidor.

Esa aplicación se conectará a puntos de terminación expuestos por el servidor conocidos como "**routes**", que serán meras URL's a las que llamará el protocolo HTTP para generar acciones sobre el servidor o para obtener datos.

Además, esta aplicación utilizará un modelo de programación que ha sido muy utilizado en el desarrollo de aplicaciones de escritorio por mucho tiempo, el modelo **MVC**[9], o Modelo Vista Controlador.

En el servidor existirá una base de datos (cuyas especificaciones técnicas se detallan más adelante) que guardará las colecciones de datos necesarias para la aplicación, pero además tendrá un almacén de caché para guardar ciertos objetos muy comunes para reducir las peticiones a la base de datos principal.

La aplicación, al iniciarse, iniciará una comunicación con el servidor mediante sockets para la exposición de eventos. Así el cliente no necesitará iniciar la comunicación con el servidor y se elimina el problema de tener que solucionarlo haciendo **polling**[10] a la base de datos. El diseño de esta comunicación ha sido el trabajo más exigente del proyecto.

Esta es la idea general de la solución que da evented.io a esta clase de problemas. Ahora veremos con mas detalle, qué se utiliza para construir la implementación de esta solución.

[8] **API** significa Interfaz de Programación de Aplicaciones. Es un conjunto de "funciones" que proveen el acceso a características o a los contenidos que proporciona un servidor. Una de las API's mas conocidas es "Google MAPS", que ofrece una API pública para interaccionar con ellos de forma externa.

[9] **MVC**, Modelo-Vista-Controlador, es un patrón de programación en el que se separan 3 componentes que son los responsables del comportamiento de toda la aplicación. El Modelo se ocupa de los datos, de su persistencia, de los cambios en ellos y de generar eventos. A los que responde el Controlador, que procesará estos eventos con fin de modificar la parte de UI o lo que está viendo el usuario, la Vista. Finalmente, la Vista también será capaz de generar eventos, como el "click" del ratón, que el Controlador procesara, para cambiar el Modelo.

[10] **Polling** se le llama a estar continuamente solicitando un recurso para comprobar si su disponibilidad o contenido han cambiado. Es una técnica que ha sido muy utilizada, pero tiene unos problemas de sobreuso de la red bastante importantes cuando el volumen de usuarios comienza a ser elevado.

3.1 Solución en el Frontend (cliente)

Como ya se ha comentado anteriormente, las aplicaciones van a utilizar un patrón MVC.

Para ello, se va a utilizar una librería del tipo meta-framework JavaScript conocido como AngularJS, y desarrollado por Google.

3.1.1 Módulos

En primer lugar mencionar que Angular provee de la noción de **módulos**, de manera que los scripts se organizan en base a dicho concepto.

Angular provee por defecto una amplia librería de módulos básicos que proporcionan muchísima funcionalidad sin la necesidad de implementarla nosotros mismos.

Para hacer uso de estos módulos, Angular utiliza un concepto denominado **Inyección de Dependencias**[11]. Por ejemplo, Angular provee un modulo llamado **"$http"** que implementa la funcionalidad para trabajar con peticiones de red asíncronas de una forma sencilla. Así, llamando simplemente a **$http.get(url),** estaríamos haciendo una llamada GET a la URL indicada**.**

Dentro de uno de nuestros módulos, podemos inyectar el modulo $http y utilizarlo libremente.

Así es como eliminamos el problema de terminar con una base de código de baja calidad, coloquialmente denominada "espagueti". La funcionalidad está modularizada y es reusable porque los módulos son totalmente "inyectables".

3.1.2 Bindings

Los bindings son un concepto inicialmente un poco extraño pero fáciles de entender. Un binding realmente es un enlace entre la plantilla HTML y un objeto JavaScript. El binding realmente lo que hace es mantener la sincronía entre ambos.

Es decir, si se detecta un cambio en los datos (el modelo), el binding se encarga de actualizar la vista, y viceversa. **Es la manera de conectar los datos al contenido que se está mostrando.**

De esta forma, conseguimos tener una representación actualizada de los datos que tenemos en nuestra aplicación.

Es interesante porque de esta forma, nosotros podemos crear un binding sin siquiera tener aún los datos (provenientes de una petición de red, por ejemplo), para que en el

[11] La inyección de dependencias consiste en definir ciertos módulos y las dependencias entre ellas. De manera que los módulos se diseñan de manera desacoplada, pero se pueden introducir módulos ajenos de una manera sencilla definiéndolos como una dependencia

momento en que lleguen, aparecer reflejados en nuestra plantilla HTML.

3.1.3 Evented.io

Para evented.io, se ha creado una librería de implementación parcial en JavaScript puro y otra parte diseñada específicamente para AngularJS.

Esta librería, que realmente es un **módulo** de Angular, sirve para implementar un protocolo de comunicación por websockets que se describirá en detalle en la sección 3.2.

La idea básica es la utilización de la nueva tecnología de "Websockets" para establecer una conexión TCP directa con el servidor. Dichos sockets, al estar conectados directamente al modelo de AngularJS, reflejarán en cada cliente los cambios producidos en el servidor en tiempo real.

A continuación se expone la implementación del servidor de API's y Websockets de **"Evented.io".**

3.2 Solución en el Backend

En la parte del Backend, se ha creado un paquete que con una configuración sencilla facilita la creación de un servidor de API's y Websockets:

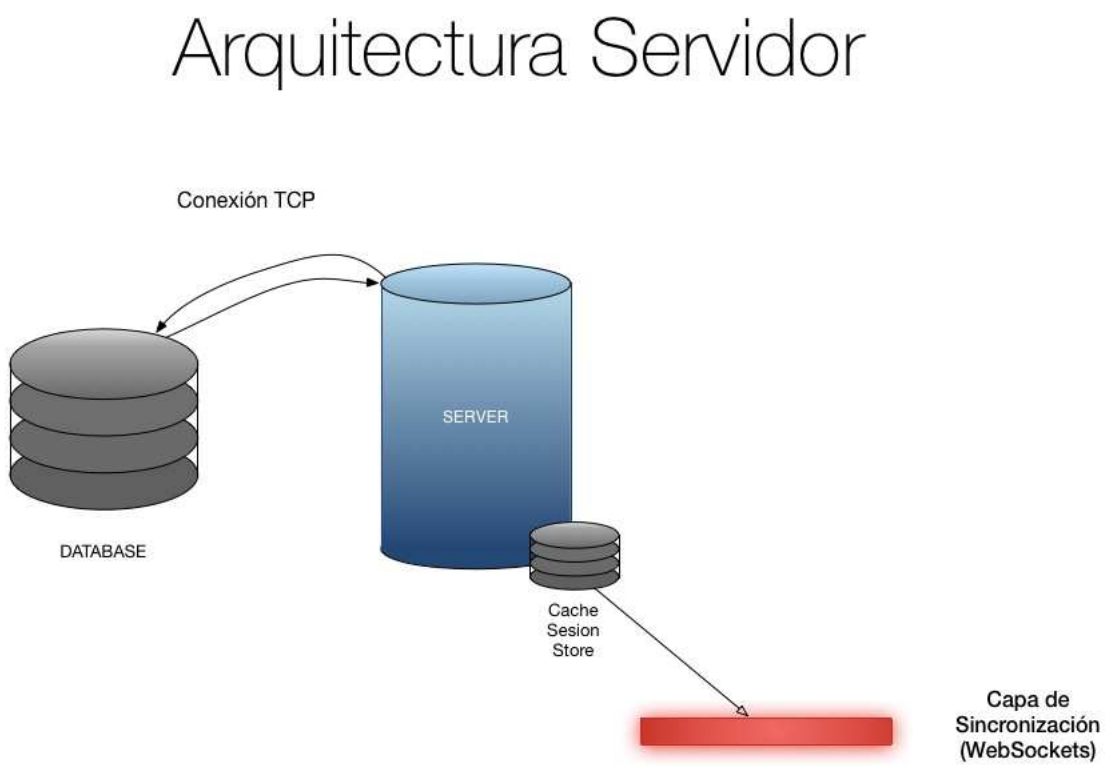


Ilustración 4. Arquitectura base del servidor.

El servidor está escrito utilizando el entorno de ejecución **Node.js**[12]**,** lo cual nos permite desarrollar nuestra aplicación enteramente en JavaScript, tanto en el cliente como el servidor, e incluso compartir código entre ambos. Como *node* esta basado en un ciclo de eventos, no realiza esperas, sino que simplemente cada vez que tiene que esperar vuelve al ciclo de eventos hasta que la tarea ha sido terminada pudiendo así atender otras peticiones en paralelo.

Un ejemplo sería una API que obtiene o escribe un fichero en disco duro. *Node*, delegaría esta tarea en el sistema operativo retirándose a su ciclo de eventos para atender nuevas peticiones. Cuando el sistema operativo ha terminado, llamará la atención de *Node* pasándole los resultados y este por último responderá al solicitante con los mismos.

[12] **Node.js**, es un Runtime JavaScript que nos permite, entre otras cosas, crear servidores enteramente utilizando JavaScript. JavaScript fue ideado para funcionar con un ciclo de eventos, por lo que su I/O (operaciones de entrada y salida), son asíncronas. Y como JavaScript es un lenguaje con características de los lenguajes funcionales, permite implementar características, que otros lenguajes como PHP no permiten, con un rendimiento muy elevado.

Otra característica importante es que está puramente ideado para servir y aceptar datos JSON[13]. Ello supone basar la comunicación directamente en objetos JavaScript, sin preocuparse por el formato de las transacciones de datos ni crear un analizador (*parser*), ya que se utiliza el nativo de JavaScript.

De la misma forma, la base de datos escogida para el framework es **MongoDB[14],** una base de datos cuyo lenguaje de query no es **SQL,** sino una API JavaScript.

El objetivo de unificar los lenguajes utilizados dentro del framework es facilitar la curva de aprendizaje y la reusabilidad.

El servidor provee una serie de API's de autenticación por defecto y la posibilidad de extenderse con nuevas API's que operen sobre la base de datos con finalidad que se desee.

Estas API's se extienden mediante la creación de modelos y acciones que operan sobre esos modelos. Por ejemplo, podemos crear el modelo (colección):

Usuario:
Atributos:
- *Nombre*
- *Email*
- *Teléfono*

Cada modelo coche puede ser almacenado en la base de datos, y podemos crear con ello una API que nos devuelva, modifique o borre usuarios de la colección en base de datos.

La aplicación cliente utilizará estas API's para obtener usuarios de la colección usuarios o para realizar modificaciones sobre los mismos.

Además de esta funcionalidad, se ha integrado también la funcionalidad de exposición de las acciones sobre las colecciones en eventos de socket. Esto supone, que cualquier modificación que se realice sobre la colección en base de datos, será instantáneamente enviada a través de los sockets a todos los clientes conectados. Lo veremos con un poco mas de detalle en la próxima sección.

[13] **JSON**, es un lenguaje de serialización para objetos JavaScript
[14] **MongoDB** es una base de datos NoSQL, lo cual implica que no utiliza el modelo relacional de Bases de Datos, sino que en vez de guardar los datos en tablas, se guardan estructuras de datos directamente en documentos JSON con un esquema dinámico en lugar de rígido (impuesto por las tablas). Los documentos se organizan en colecciones y en lenguaje de query es tan sencillo como "NombreColleción.find()", todos los documentos de la colección.

3.3 Solución a la comunicación

En esta sección se profundiza en la forma en la que se lleva a cabo la comunicación en **evented.io.**

En primer lugar se utilizan los puntos de terminación del servidor de los que hemos hablado anteriormente. Son rutas a las que se hacen peticiones HTTP y estas devuelven datos en formato JSON.

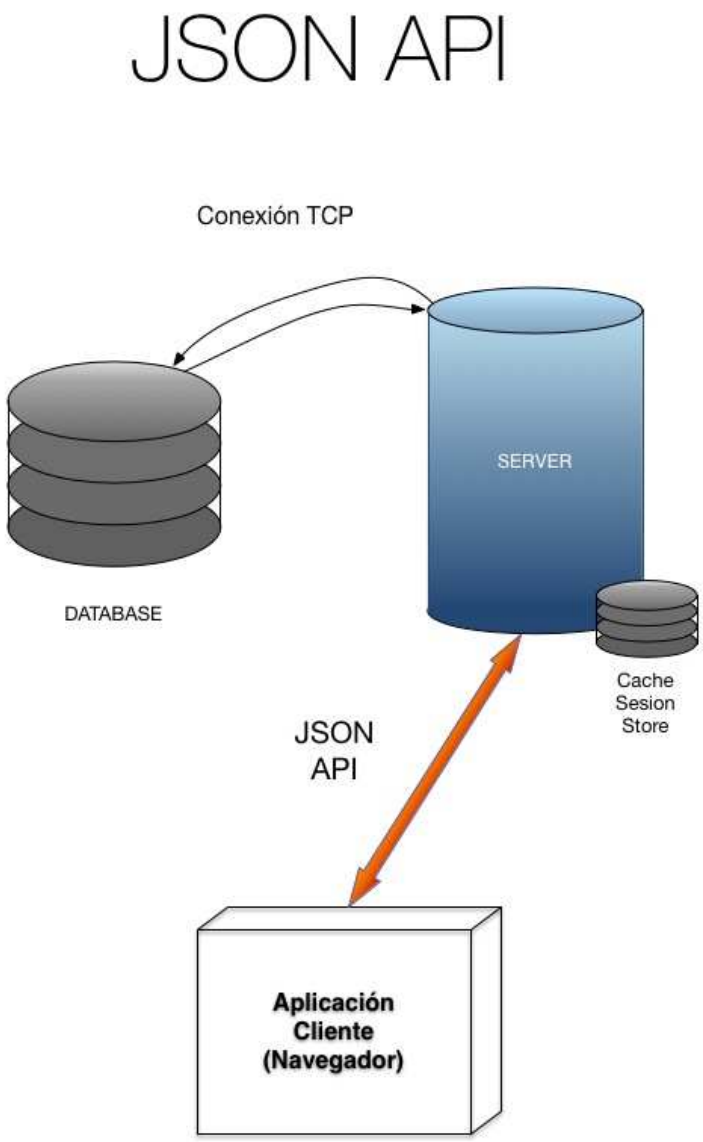


Ilustración 5. Comunicación mediante una API JSON

Por ejemplo, cuando llamamos a la API de autentificación, esta es la secuencia que ocurrirá:

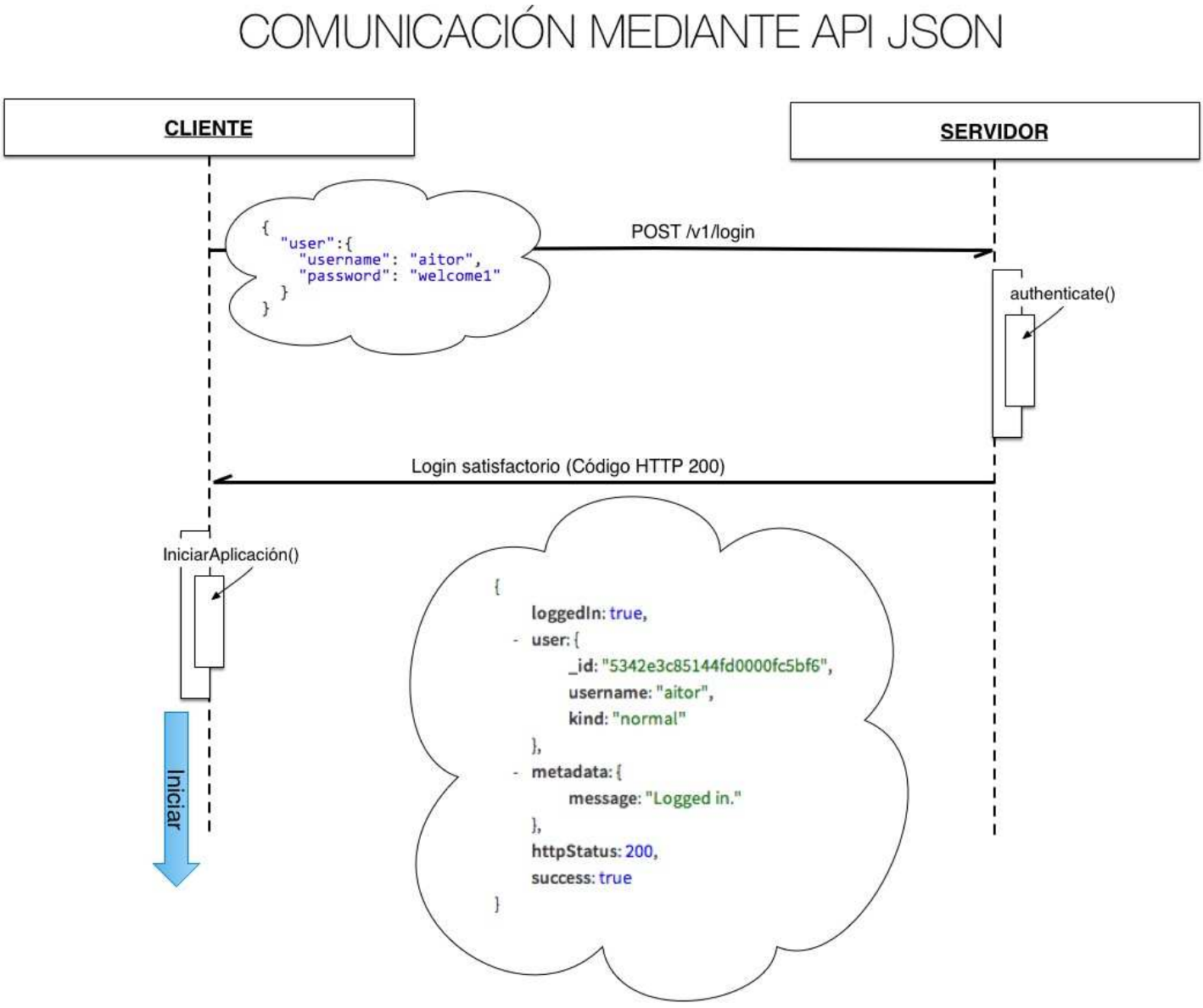


Ilustración 6. Intercambio de mensajes JSON

Un POST a la ruta www.dominio.com/v1/login, enviándole los datos del usuario que queremos autenticar generará una nueva sesión de usuario y devolverá el usuario almacenado en base de datos indicando que el *login* ha sido satisfactorio.

En el caso de los *WebSockets*:

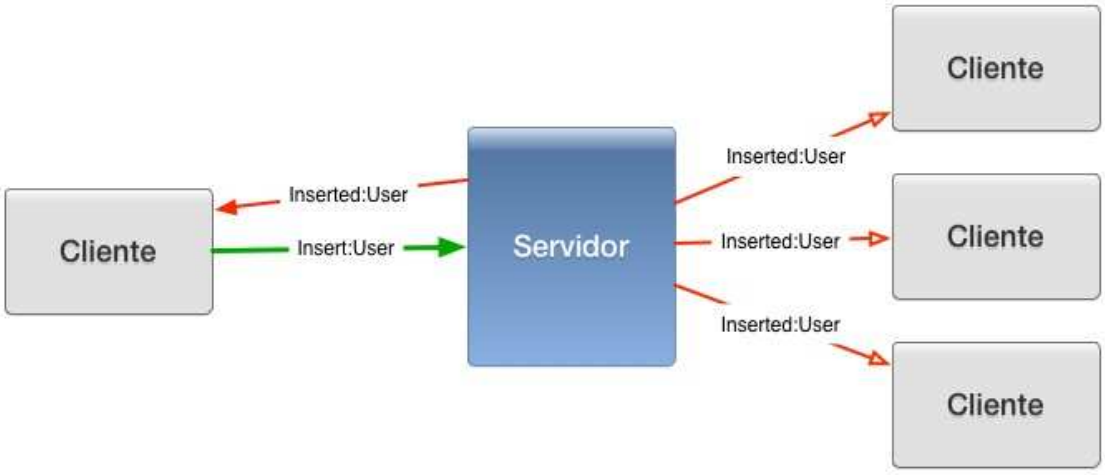


Ilustración 7. Secuencia de eventos generados en una inserción dentro de una colección.

Cuando un cliente introduce un documento dentro de una colección, o ya sea el propio servidor (en el caso de la Ilustración 7, se muestra la colección "*User*"), automáticamente se generará un evento en los sockets del resto de clientes indicándoles que ha aparecido un nuevo cliente en la colección.

De este modo, se mantiene una sincronización entre las colecciones y los host conectados al servidor.

Evented.io, proporciona una librería para crear estas colecciones tanto en el cliente como en el servidor para hacer uso de este protocolo de eventos.

4 Utilizando Evented.io

Para utilizar Evented.io en nuestros proyectos se debe instalar en primer lugar la herramienta de línea de comandos de Evented.io, **evented-cli.**

Esta herramienta está escrita también utilizando el *Runtime* de Node.js y nos ofrece la posibilidad de hacer un *scaffolding* completo de una aplicación, es decir, crear la estructura de directorios y archivos base que se van a necesitar con algunos ejemplos de utilización ya comentados.

Para poder utilizar esta línea de comandos, es necesario tener instalado:

- **Node.js.**
- su gestor de paquetes **NPM.**
- y **GIT,** un sistema de control de versiones que utilizará la línea de comandos para acceder al repositorio semilla de la aplicación.

Los pasos para instalar estas dependencias se detallan en sus respectivas páginas web. Una vez instaladas basta con hacer:

npm install –g evented-cli

desde una consola de Windows o Unix para iniciar la descarga de **evented-cli.**

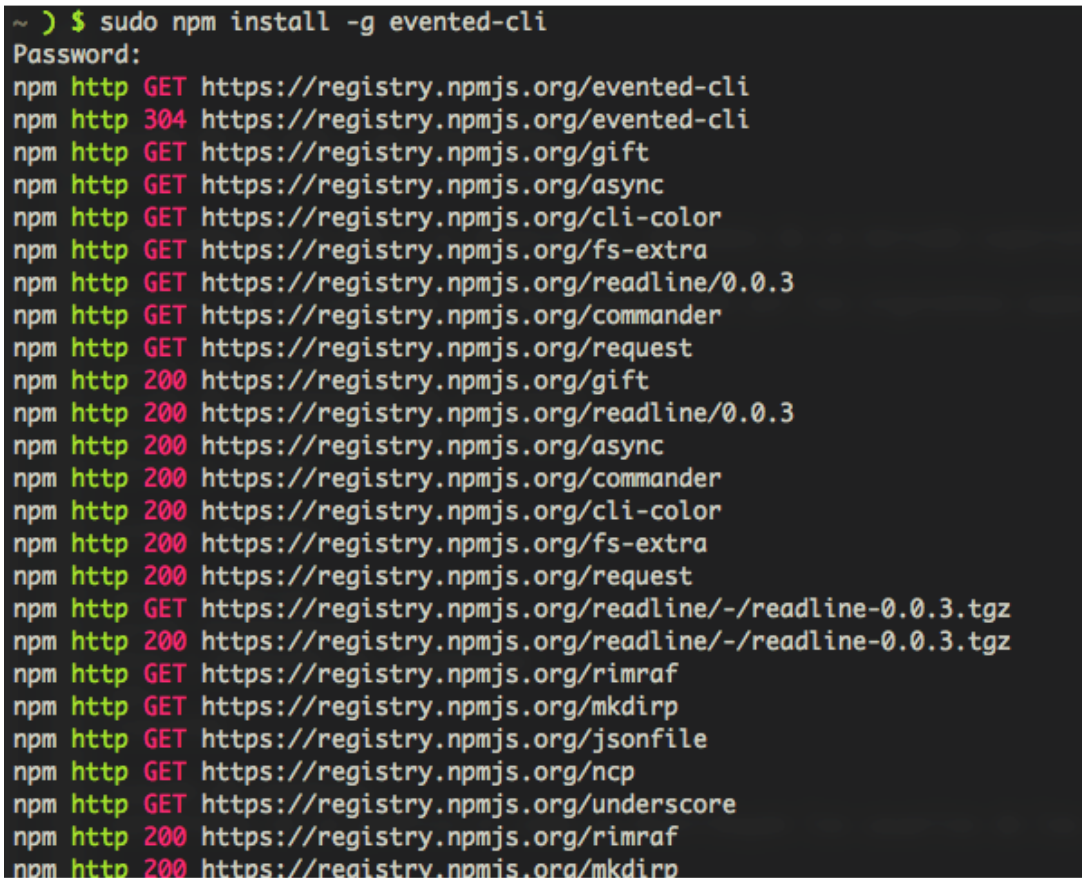


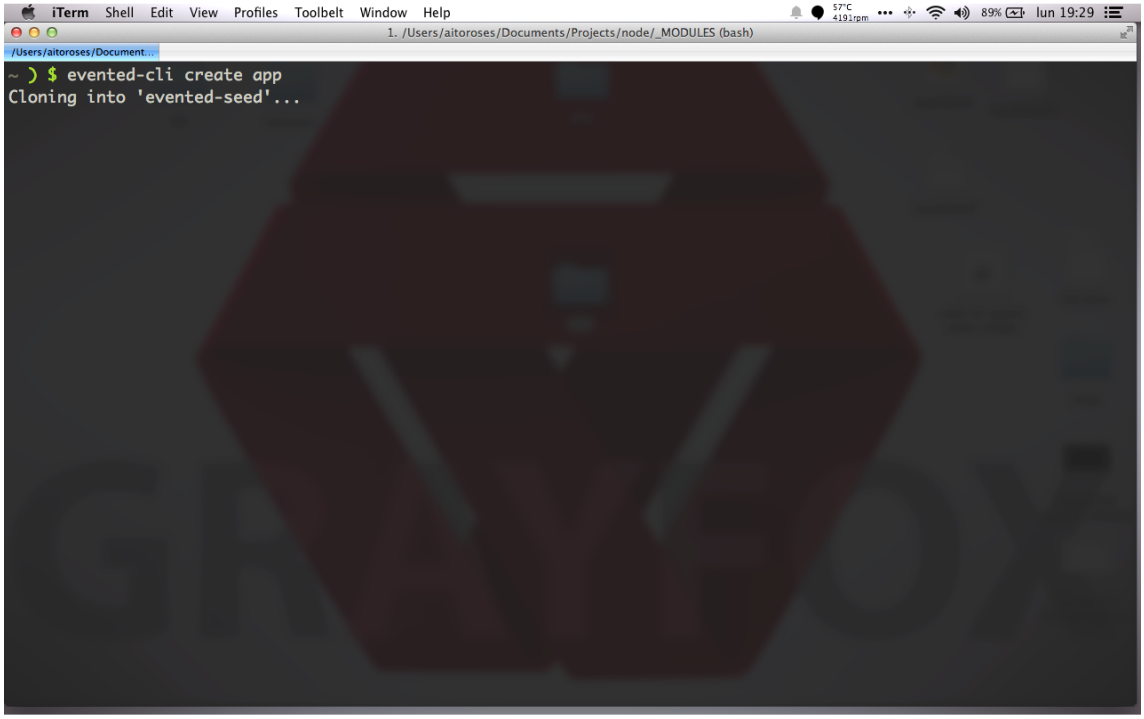
Ilustración 8. Instalación de evented-cli mediante una línea de comandos

El script de instalación instala todas las dependencias del proyecto.
Una vez terminado este paso, pasamos crear nuestro primer **proyecto evented.io**.

Para ello, utilizaremos el comando *create* de la línea de comandos.

evented-cli create NombreDelProyecto

Este comando accede al repositorio de Evented.io y crea un directorio con el nombre del proyecto. La Ilustración 9 muestra un ejemplo con la jerarquía de directorios.



(a)

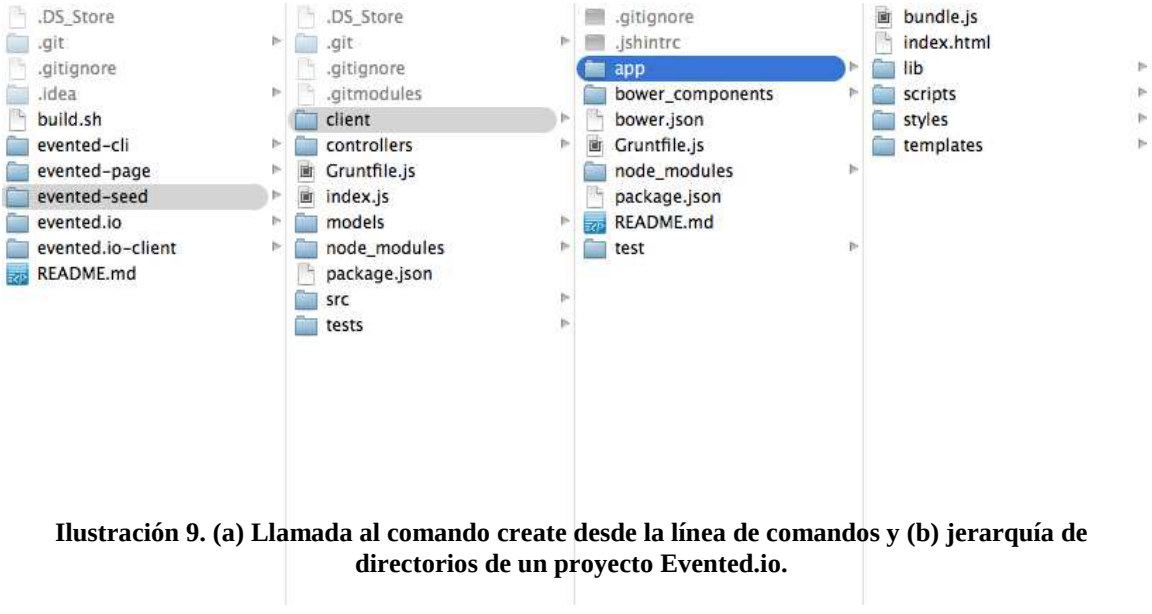


Ilustración 9. (a) Llamada al comando create desde la línea de comandos y (b) jerarquía de directorios de un proyecto Evented.io.

En este paso se puede considerar que se ha creado un proyecto básico **Evented.io.**

4.1 Cliente

Para lanzar el cliente ese debe avanzar hasta la carpeta **client** con la línea de comandos y ejecutar:

grunt dev

Este comando lanza el servidor de desarrollo de la aplicación cliente, el cual se encargará de compilar los archivos, comprimir los scripts JavaScript y buscar errores en el código. Además observará cambios en los archivos del directorio y subdirectorios para volver a lanzar las tareas de construcción.

Accediendo a través del navegador a http://localhost:8888, accedemos al servidor de desarrollo y observamos la aplicación semilla descargada con el repositorio.

En este punto podemos ya dedicarnos al desarrollo específico de nuestra nueva aplicación. Este desarrollo seguirá las líneas de *Angular*, por lo que será mejor referirnos a la documentación oficial del proyecto http://docs.angularjs.org , donde encontraremos toda la información referente a la utilización de sus módulos.

El desarrollo de una aplicación con Angular se puede sintetizar en el siguiente diagrama:

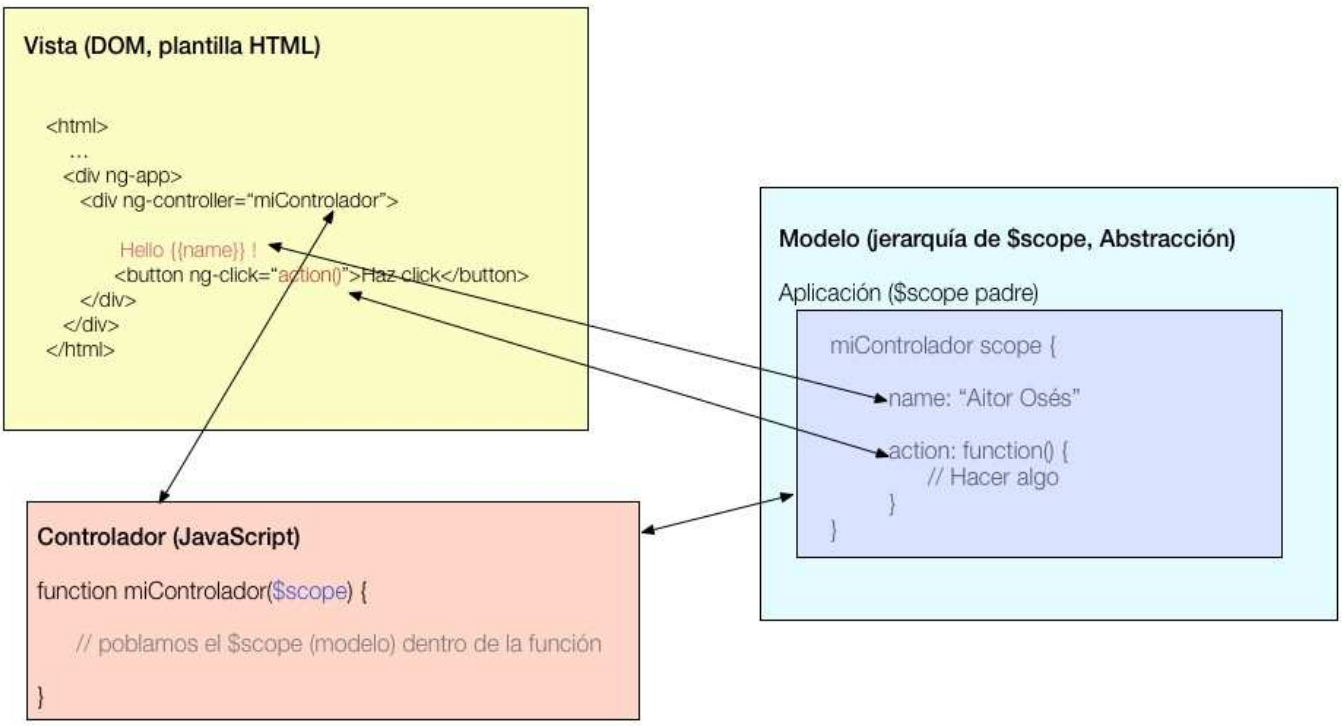


Ilustración 10. MVC en Angular.js

El concepto de la *Ilustración. 10* es la base de AngularJS y muestra cómo se entrelaza el código **JavaScript** con el **HTML** generando de forma abstracta el "modelo".

Lo que sí forma parte de este PFC es el proceso de utilización de la librería de *websockets* de Evented.io para crear colecciones de datos en el cliente.

Librería de Evented.io para Angular.js

En esencia los *websockets* actúan de forma transparente y de lo único de lo que el desarrollador debe preocuparse es de crear las colecciones que se manejarán desde el cliente. Para ello se crea un controlador que controlará una de nuestras vistas:

```
function miControlador($scope, $evented) {
  var Usuarios = $evented.Collection("Users");
  Usuarios.attach("usuarios", $scope);
}
```

Ilustración 11. Controlador en el que creamos la colección Usuarios

Llamando a ***"$evented.Collection"*** creamos la colección **Usuarios** en una variable JavaScript, y automáticamente se inicia el proceso de suscripción a una colección publicada por el servidor llamada **"Users"** a través de los *websockets* HTML5.

Esta variable puede ser utilizada en nuestras plantillas mediante el controlador de la siguiente manera.

```
<div ng-controller="miControlador">
  <!-- Lista -->
  <ul>
    <li ng-repeat="for user in usuarios">
      {{user.name}} - {user.email}
    </li>
  </ul>
  <!-- fin de lista -->
</div>
```

Ilustración 12. Plantilla donde se hace uso del controlador de la ilustración 11

De esta forma mostramos una lista en la que nos aparecen los nombres e emails de los usuarios de la colección "Users" publicada por el servidor.

Para más información acerca del uso de esta libraría y de Evented.io en el cliente , referirnos a la página y documentación del proyecto en GitHub (http://www.github.com/aitoroses/evented.io).

4.2 Servidor

La creación de un servidor con Evented.io, es decir, una nueva aplicación de servidor, se debe proceder de la siguiente manera.

- Habilitar una instancia de base de datos de MongoDB[15] en el puerto por defecto.
- Habilitar una instancia de servidor Redis[16] en el puerto por defecto.

El proceso de instalación de estos servidores se detalla en las correspondientes páginas web especificadas en las notas al pié.

Dentro de la carpeta "**NombreDeProyecto**" existe un archivo llamado **index.js. Se puede considerar** este archivo como el motor de nuestro servidor, ya que es respaldado por todas las ***librerías de Evented.io***.

Index.js

```
// SCRIPT DEL SERVIDOR EVENTED.IO

// Creamos el servidor
var evented = require('evented.io');

var Evented = evented({port: 5000});

// Definimos las colecciones
var User = new Evented.Collection('User');
var Tarea = new Evented.Collection('Tarea');

// Publicamos parte de las colecciones
Evented.publish(User, function() {
  return User.find({_id: this.userId()});
});
Evented.publish(Tarea, function() {
  return Tarea.find({user_id: this.userId()});
});

// FIN DEL SCRIPT
```

Ilustración 13. Script de ejecución del Servidor Evented.io

[15] Instalación de MongoDB: http://docs.mongodb.org/manual/installation/
[16] Instalación de Redis: http://redis.io/download

Con este sencillo script se obtiene lo siguiente:

1. Crear un servidor de Node Evented.io en el puerto TCP 5000
2. Definir dos colecciones de datos en memoria que extraen su contenido de las colecciones de base de datos **User** y **Tarea.**
3. Crear por debajo del código ciertas API's básicas de autenticación.
4. Crear publicaciones cuyas subscripciones se resuelven en el momento de conexión de nuevos clientes. En concreto la colección **User** publica solamente aquel usuario que tiene el mismo identificador que el usuario que se está logueando en la aplicación.
   Y en el caso de **Tarea** está publicando aquellas tareas que pertenecen al usuario que se está logueando.
5. La posibilidad de crear colecciones en el cliente que se comuniquen con las colecciones del servidor.

Al ejecutar la operación ***start*** de la herramienta de línea de comandos de evented-cli:

evented-cli start

Arranca el servidor Evented.io:

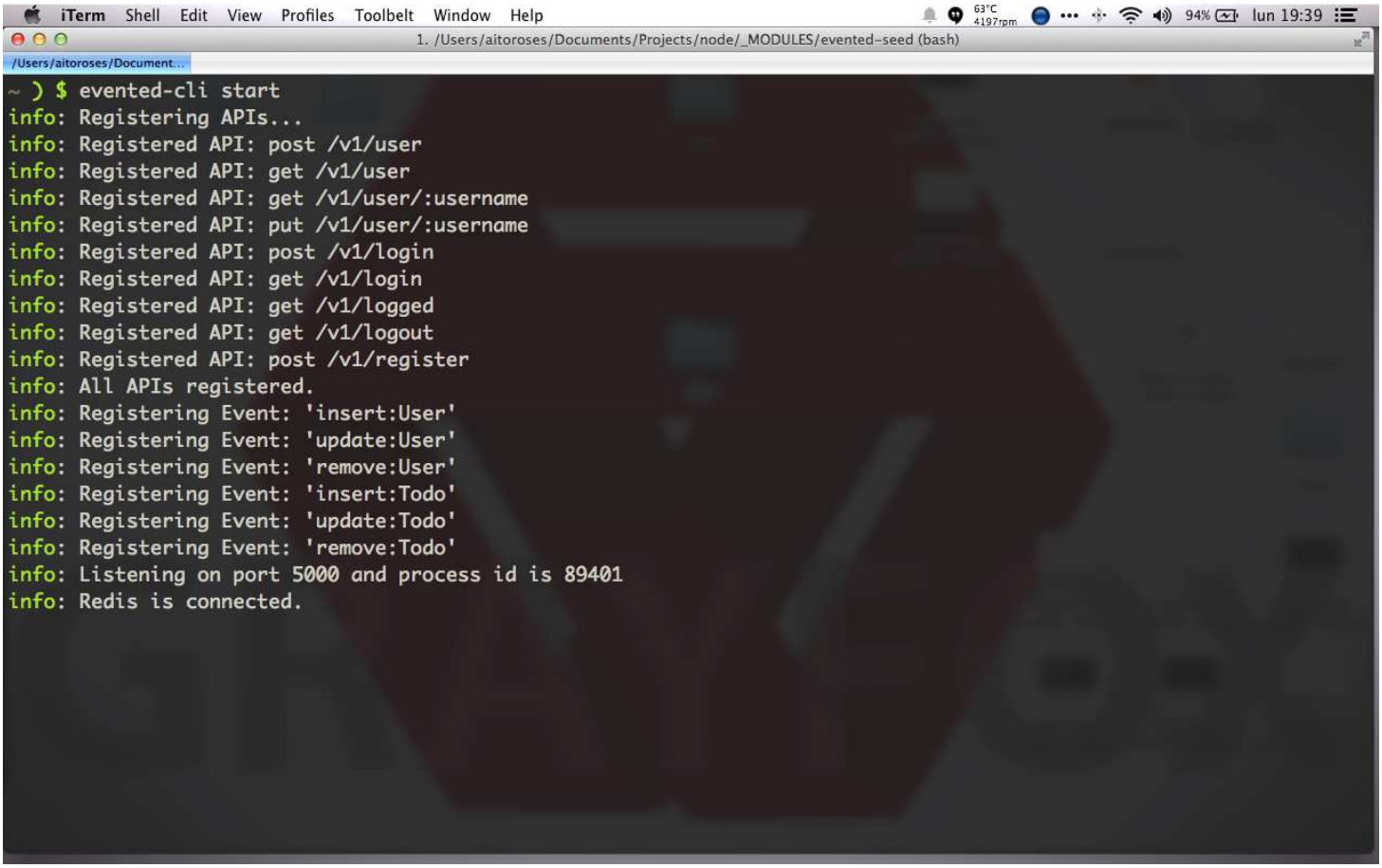


Ilustración 14. Arranque del Servidor

El servidor ha definido una serie de API's HTTP básicas cuya extensión con nuevos modelos y controladores se pasan a explicar a dentro de esta sección.

Los nuevos eventos que se definen son:

- insert:User
- update:User
- remove:User
- insert:Tarea
- update:Tarea
- remove:Tarea

Estos eventos son los eventos de ***WebSockets*** del servidor.

- Un evento *insert* se genera por el cliente cuando hay un nuevo documento dentro del a colección.
- Un evento *update* se genera por el cliente cuando hay una modificación de un documento dentro de la colección.
- Un evento *remove* se genera por el cliente cuando se ha eliminado un documento de la colección.

Cuando **Evented.io** recibe uno de estos eventos procesa el evento y genera otro distinto que retransmite a todos los usuarios conectados a través de los ***websockets***.

- Inserted:Colección, cuando se emite un nuevo documento en la colección.
- Updated:Colección, cuando se emite la modificación de un documento.
- Removed:Colección, cuando se emite la eliminación de cierto documento

Este protocolo de eventos funciona de forma transparente al usuario y da la **impresión de manejar colecciones de base de datos directamente desde las colecciones en el cliente**, mientras que lo que en realidad está sucediendo es esta secuencia de eventos.

Además, se logra una sincronización entre la base de datos servidor Evented.io y la aplicación cliente de Evented.io.

El servidor de API's

El inicio del servidor registra las API's propias del framework referentes a la **gestión de usuarios y sesiones**.

Para mostrar el funcionamiento del servidor detallamos como ejemplo las API's de usuario que existen por defecto en el framework.

Estas llevan el prefijo **v1,** porque sus controladores son versionables (**v1, v2**...).

Login

Desde **`localhost:5000`** (el puerto donde está funcionando) nos dirigimos desde el navegador a la ruta:

`http://localhost:5000/v1/login`

Se accede directamente a la **API `GET /v1/login`** que se muestra en las API's registradas de la ***ilustración 12***.

El servidor, permite obtener descripciones de sus API's de una forma similar, pasando como parámetro la variable ***`describe_api=true`*** .

De este modo, si queremos obtener una descripción de la **API `GET /v1/login`** tendríamos que acceder a la ruta:

`http://localhost:5000/v1/login?`**`describe_api=true`**

Ilustración 15. Descripción de API's

Podemos observar la descripción de la API, los parámetros requeridos y los opcionales, los roles y los tipos de usuario que pueden acceder a él, etc..

En este caso, los parámetros requeridos son **`username`** y **`password`**. Así pues si se accede a la ruta pasándole sus parámetros requeridos:

Ilustración 16. Login de usuario

La respuesta del servidor es la de login satisfactorio. También devuelve el tipo de usuario que somos, su *id* en la base de datos etc..

La nueva variable **`loggedIn`** indica que el usuario está **logueado con éxito**. Si se observan las respuestas veremos el parámetro **httpStatus**, que es el código http de la respuesta.

Caso de error

Ilustración 17. Login erróneo

Siempre existe un código de respuesta por lo que en caso de error obtendremos el de autorización no permitida.

Extender el framework en futuras evoluciones y versiones con nuevos códigos de error es posible. Para ello referirse a la **documentación de Evented.io (http://evented.meteor.com, página web de Evented.io).**

Comprobación del login

Una vez el usuario se ha logueado, puede acceder a su información de usuario y probar si realmente se ha creado una sesión en el servidor accediendo a **GET /v1/logged**

Ilustración 18. API para obtener el usuario logueado

El servidor responde con un mensaje JSON indicando que el usuario está logueado, además de un mensaje de texto señalando "**Estás logueado**.". Resaltar que los mensajes son internacionalizables, es decir, se pueden traducir.

También envía un campo para metadatos y el objeto del usuario obtenido de base de datos (*obsérvese* la contraseña *hasheada*[17].)

[17] Hashing es la técnica de digestión de mensajes mediante un algoritmo en concreto. . Mediante hashing, se obtiene el resumen de una contraseña. Se utiliza por seguridad. En base de datos se almacena el hash (resumen) en vez de la propia contraseña. A la hora de realizar un login se hashea la contraseña entrante y se comparan ambos hashes, si coinciden se autoriza el login. El hash por si solo no sirve para nada.

Papel de Redis[18] en Evented.io

Redis es una Base de datos en memoria. Su papel en este servidor es la de almacenar datos importantes como las sesiones y las publicaciones en un lugar con persistencia pero ligero y rápido como la memoria RAM.

La sesión del usuario queda registrada en Redis con el *middleware*[19] conocido como *Passport* [20], que permite también configurar OAUTH para poder loguearnos con el usuario de cualquier *oauth provider*[21] (Facebook, Google, Twitter, Github...) aunque de momento no está implementado en esta versión del framework.

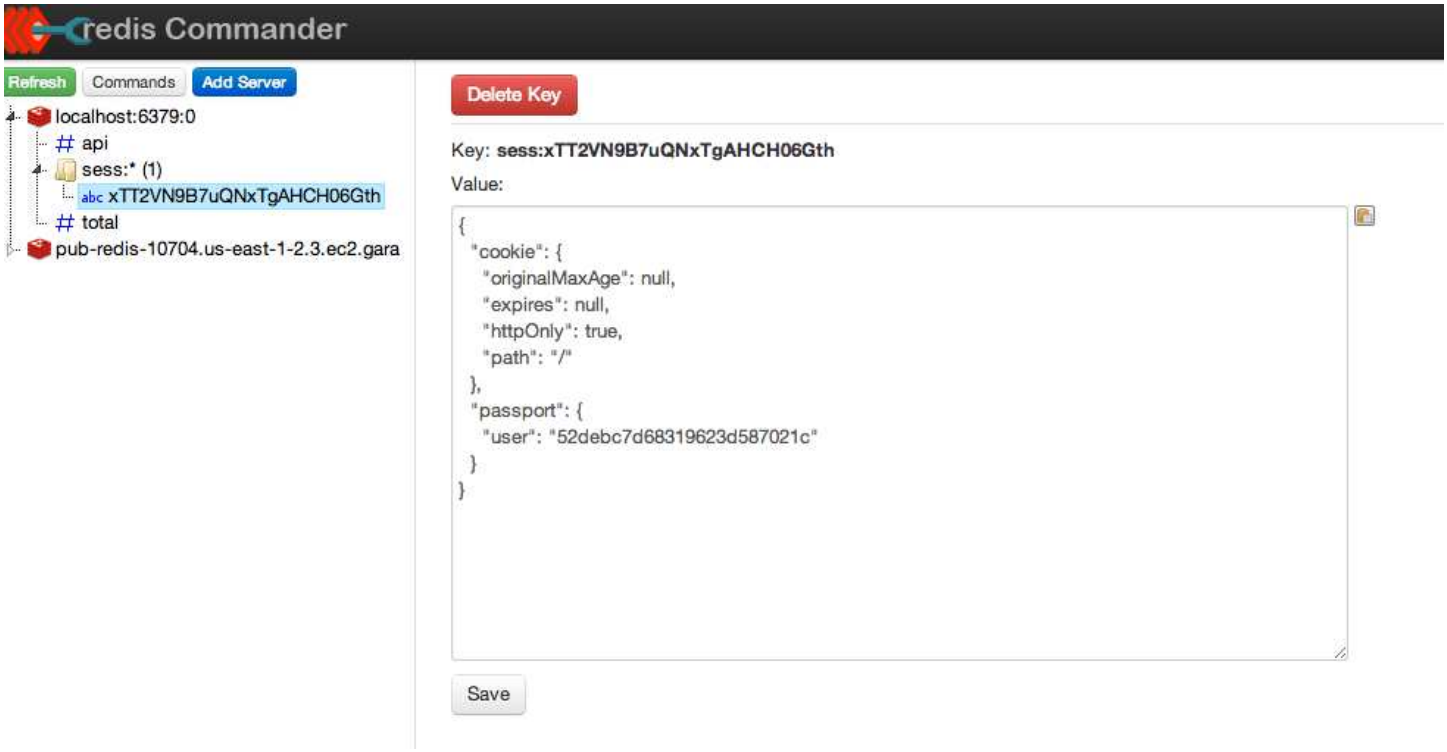


Ilustración 19. Almacenamiento de sesión en Redis

[18] **Redis**, se conoce a un tipo de base de datos NoSQL en memoria. Es decir, los datos permanecen en memoria. No obstante, estos datos también quedan respaldados por un sistema de persistencia. Redis actual como un almacén clave-valor y se utiliza para cachear ciertos objetos que deben perdurar a pesar de que por ejemplo, un servidor se reinicie.

[19] **Middleware,** es un concepto bastante utilizando en los frameworks del lado del servidor. Básicamente son pasos de preprocesado sobre una petición como podría ser el añadir cierto tipo de cabecera al mensaje.

[20] **Passport** es un módulo de Node.js que proporciona funcionalidad para crear sesiones de usuario de una forma sencilla basándose en distintas técnicas (OAUTH, HTTP AUTH…).

[21] **OAUTH Provider** se le conoce al servicio de autenticación que ofrecen sistemas tales como Facebook, Google, Twitter… Que permite ser utilizado para loguearse en otros servicios o páginas web con los mismos credenciales.

Se observa cómo en la base de datos Redis se está almacenando la sesión del usuario que se está logueando. Concretamente se guarda un registro que guarda el SESSION_ID o identificador de sesión, junto con el identificador de Usuario de base de datos.

API's de acceso a colecciones

Para cada colección en base de datos podemos crear un conjunto de API's http que nos permiten interaccionar con ellas utilizando operaciones CRUD[22] expuestas mediante una API REST[23] JSON.

Evented.io, trae configurado un controlador de APIs para la colección **User** por defecto. Por ejemplo, accediendo a **GET /v1/user**

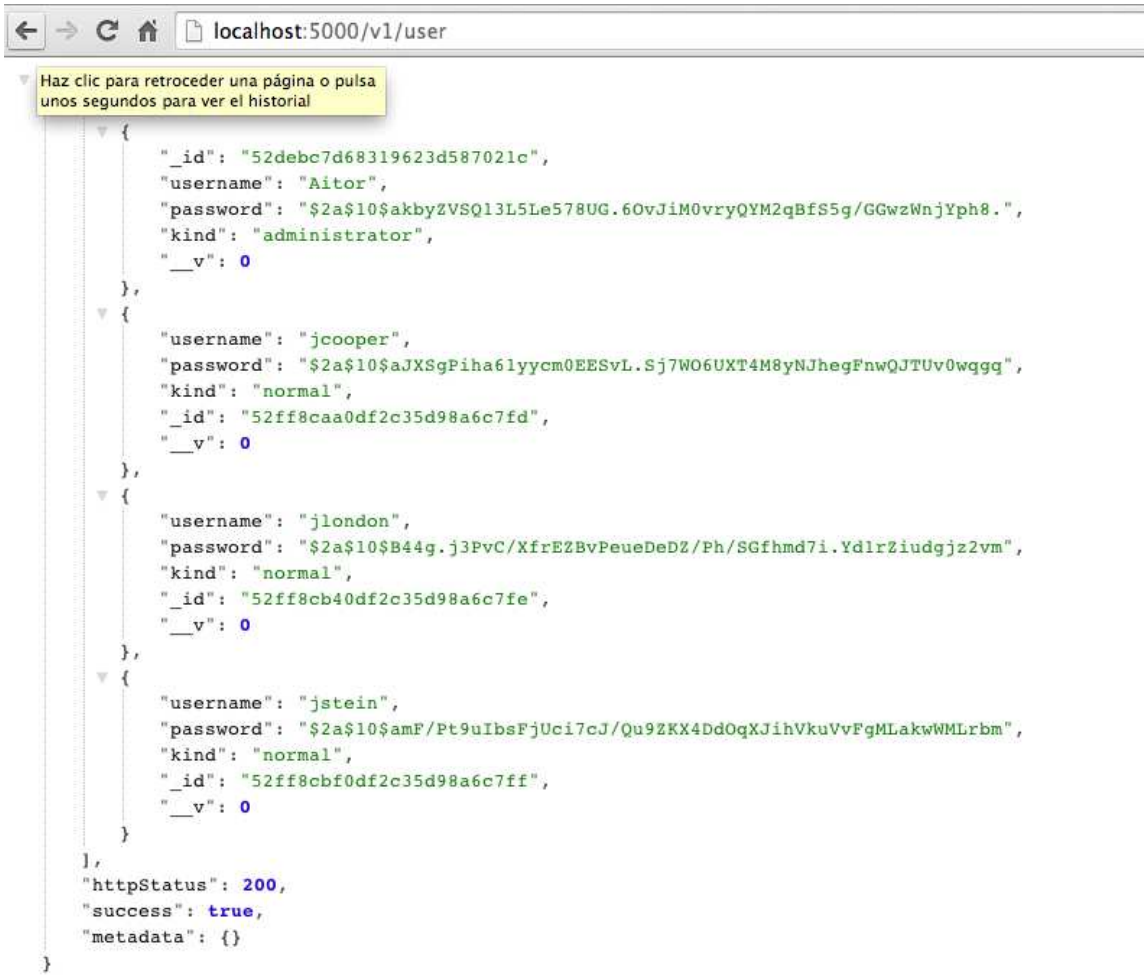


Ilustración 20. API GET /v1/user

En cambio utilizando otro verbo HTTP se puede hacer otro tipo de operaciones CRUD:

- POST /v1/user (Añadir)
- PUT /v1/user (Modificar)
- DELETE /v1/user (Eliminar)

Con la llamada a **GET /v1/user**, se nos devuelve un array JSON con los usuarios que existen dentro de la colección **User** que podremos manipular desde JavaScript**.**

[22] **CRUD**, viene de "Create, Read, Update, Delete". Son operaciones para obtener registros de una colección, crearlos, modificarlos o eliminarlos.

[23] **REST**, es en síntesis la exposición de unas URI JSON que permiten realizar operaciones CRUD, mapeándolas a los verbos HTTP: GET (obtener), POST (crear), PUT (actualizar) y DELETE (borrar)

Logout

Para eliminar la sesión de usuario se debe acceder a:

`GET /v1/logout`

Ilustración 21. API de logout

El servidor responde que la sesión ya no existe. Esto se puede comprobar accediendo otra vez a la API **GET /v1/logged:**

Ilustración 22. Comprobando que la sesión no existe

Se ha visto cómo se utilizan las API's de usuario y la filosofía de Servidor de API's JSON que implementa Evented.io. A continuación se muestra cómo extender el servidor con nuevos controladores y modelos de datos, es decir **nuevas API's.**

Extendiendo el servidor con nuevos controladores y modelos

Para extender la funcionalidad del framework, se pueden hacer dos cosas:

- Crear un nuevo controlador con nuevas API's
- Crear un modelo de datos para la base de datos

Debido a l diversidad de funcionalidades exigidas por las aplicaciones, esta extensiones prácticamente indispensable en el 100% de las mismas.

Creando un nuevo modelo de datos.

El modelo es necesario crearlo tanto para API's JSON como para colecciones *WebSocket*.

Para crear un nuevo modelo de datos se deben seguir los siguientes pasos. En la carpeta de nuestro servidor, existe una carpeta llamada **`/models`** , y es ahí donde se define el nuevo modelo.

Se crea el modelo de una **Tarea**, a modo de **checklist**. Es decir, una tarea que tendrá un nombre, un campo que indicará si está realizada o no, y otro campo indicando el identificador del usuario al que pertenece.

El modelo se va a definir utilizando un módulo conocido como **Mongoose**[24]

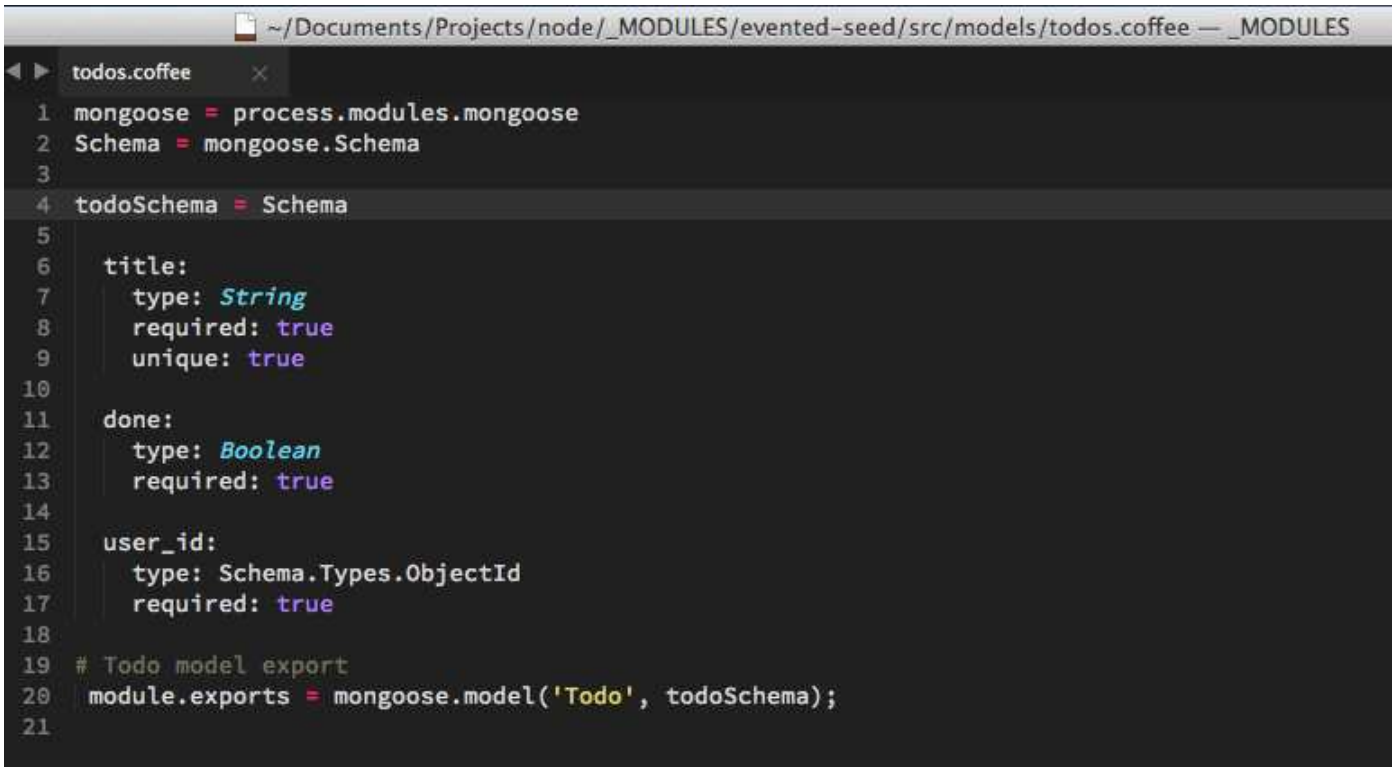


Ilustración 23. Definición de un modelo

[24] Mongoose, es un módulo de Node.js que permite definir modelos que actúan sobre las colecciones de MongoDB proporcionando un esquema de datos.

Nótese la última línea de código, donde se exporta el modelo creado con el esquema definido.

Creando un nuevo controlador

El siguiente paso es hacer accesible este modelo de datos mediante una llamada al servidor.

Para ello se crea una API dentro de un controlador.

En la carpeta del servidor, existe una carpeta llamada /**`controllers`** , que contiene los controladores en distintos archivos.

Si por ejemplo se quisiera crear una API que devuelva todos los *Todo's* (Tareas) que pertenecen a un mismo usuario, se haría del siguiente modo.

```coffee

# API para obtener todos los usuarios del modelo 'Todo'

Todo = process.server.mongodb.models.Todo

getTodos =
    method: 'GET'                                       # En esta linea, especificamos
                                                        # el metodo HTTP
    path: "/todo/:user_id"                                  # Especificamos la ruta,
                                                        # y sus parametros
    version: 1                                          # La versión de la API

    description: 'Get the list of Todos of a user.'     # Descripción de la API

    docURL: "http://api.domain.com/todoController#GET_USER_TODOS"   # Ruta en la documentación

    allowedUserKinds: []                                # Tipos de usuario permitidos
    roles: []                                           # y tambien los parametros
    params: {                                           # (En este caso ninguno)
        required: []
        optional: []

    callback: (req, res, completeCall) ->               # Accion que realiza la llamada

        userId = req.params.user_id                     # Tomamos el parametro userId

        Todo.find {user_id: userId}, (err, todos) ->    # Hacemos una query a mongoDB

            if err then completeCall {httpStatus: 500}              # Si hay error devuelve 500
            if todos is null then completeCall {httpStatus: 400}    # Si no encuentra todos, devuelve 400

            else completeCall {todos: todos}                        # Devuelve los Todo's

module.exports = (app) ->

  return [getTodos]                                     # Exporta una lista de acciones
                                                        # que serán leidas y registradas
                                                        # por el framework
```

Ilustración 24. Definición de una nueva API

En este punto se puede hacer **`GET /v1/todo/:user_id`** , donde **`:user_id`** es el identificador del usuario al que pertenecen y de esta forma se devuelven los *Todo's* de un usuario.

A este controlador le faltarían acciones para añadir todos, actualizar, eliminar, etc.. Es decir, muchas veces, básicamente necesitamos operaciones CRUD (crear, leer, actualizar y eliminar).

Por ello el framework también integra una forma sencilla de crear recursos REST utilizando un módulo denominado **`restifier`** .

```
1  restifier = require 'evented.io/restifier'
2
3  id_field = '_id'
4  name = 'todo'
5  version = 1
6  model = process.server.mongodb.Todo
7
8  module.export = (app) ->
9
10   return restifier(id_field, name, version, model)
```

Ilustración 25. Módulo restifier para la creación de recursos REST

Siguiendo estos sencillos pasos, es fácilmente realizable un servidor basado en colecciones de datos accesibles desde URL's utilizando un protocolo REST o simplemente manejando las colecciones mediante ***Websockets***.

5. Líneas Futuras

La tecnología web está cambiando mucho en los últimos tiempos. Esto se debe principalmente a lo activa que es la comunidad OpenSource.

Continuamente sus contribuyentes bombardean a la comunidad con nuevas ideas y formas de hacer las cosas. Lo que lo convierte la web en el contexto más volátil de la programación.

Existen estándares HTML5, CSS3, JavaScript (ECMAScript 5) pero siguen con la misma filosofía que desde el principio del internet.

El cambio es lento, pero están apareciendo interesantes formas de hacer las cosas.

5.1 Meteor

Por un lado tenemos Meteor. Meteor ha llamado la atención de la comunidad Web por su simplicidad. Un framework JavaScript OpenSource Realtime.

Promete desarrollar absolutamente toda nuestra aplicación: tanto el cliente y el servidor, como la comunicación en puro JavaScript, con un potente sistema de plantillas y cuya piedra angular son los ***WebSockets*** y la **reactividad**, como en **Evented.io.**

Aún es bastante joven, pero algunas empresas ya lo han desplegado en producción. La idea es tener todo **absolutamente centralizado** y el framework se hace cargo de los componentes más complejos.

Es un framework **Full-Stack,** lo cual implica que no se puede integrar con otros frameworks ni librerías de una forma sencilla, sino que pretende dar una solución total.

Sus próximas versiones prometen mucho y está siendo desarrollado por un equipo muy veterano que está teniendo mucho apoyo y aceptación

5.2 Famo.us

Famo.us es hasta ahora, la aparición más novedosa dentro de este contexto.

Abandona completamente la filosofía de HTML he impone una nueva forma de crear aplicaciones más parecida a la de los SDK privativos como Android e iOS.

Con el **objetivo**, de optimizar el **rendimiento** y la **interacción, p**ermite construir una aplicación mediante la definición de vistas y superficies.

El equipo de Famo.us ha rediseñado el **sistema de eventos del DOM, el árbol de render** (ya no se define una página HTML) y **el proceso de renderizado** también, permitiendo crear hasta aplicaciones 3D que llegan a funcionar en un iPad a 60 imágenes por segundo. Todo un hito en el rendimiento de las aplicaciones web.

6 Conclusiones

El contexto de aplicaciones de internet está cambiando mucho. En los últimos años se está viendo el impacto que ha tenido el boom de los smartphone. Por estadística se sabe que existen activos 4 terminales móviles por cada persona del mundo, lo cual parece increíble, pero nos da información sobre el alcance que tienen las aplicaciones móviles.

Se empezaron viendo aplicaciones en el iPhone, mas tarde salió Android al mercado, pero la tecnología que prevalecerá y superará sobre las otras, será sin duda las aplicaciones Web.

Las aplicaciones web permiten la implementación de una aplicación "una sola vez" y un despliegue para todos los tipos de dispositivo que existen y existirán en el futuro. Tan solo es necesario que el dispositivo tenga un navegador. Televisiones, teléfonos móviles, ordenadores, tabletas, consolas... etc., una aplicación web puede llegar a todos ellos.

La mayor virtud de estas aplicaciones web, es que forman parte de un ecosistema que hoy día es estándar: HTML5, CSS3 y JavaScript. Lo que necesita y aún es un defecto, es la falta de estandarización de los desarrollos con estas tecnologías, problema que Evented.io trata de resolver.

6.1 Cliente

Las prácticas en el cliente parten del diseño de las aplicaciones en el entorno nativo: la aplicación está enteramente implementada en el cliente es programada completamente en JavaScript utilizando un patrón MVC.

Los nuevos estándares que están apareciendo en los navegadores mas recientes abren un abanico de posibilidades que dotan al navegador web de capacidades antes inimaginables. Por ejemplo, el mas novedoso, se conoce como WebGL, y promete una API directa para crear gráficos por la GPU, es decir, como los videojuegos.

Evented.io proporciona un proyecto base extensible siguiendo el patrón MVC para crear una aplicación completamente ejecutada en el cliente y con capacidad para comunicarse con un servidor de API's mediante websockets.

Está basado en la librería Angular.js, que al ser una librería de Google bastante madura, tiene cierta garantía de que es un código bueno para producción y se evitan varios problemas de rendimiento y de compatibilidad comunes a las aplicaciones JavaScript.

6.2 Servidor

En el contexto de servidores, Node.js y sus derivados son bastante jóvenes, pero están alcanzando un nivel de madurez que los hace muy apropiados para su utilización en un entorno de producción.

En cualquier caso es necesario migrar a nuevas tecnologías y abrir un poco más lamente sobre cómo debe crearse una aplicación web, página o servidor de datos.

Node.js esta llegando al mercado de la industria y es seleccionado por la mayoría de los nuevos startups que están surgiendo.

En su caso, lo que lo hace único es que se basa en el moderno motor JavaScript V8 de Google, escrito en C, siendo su mayor característica el rendimiento y que su I/O (operaciones de entrada/salida) es asíncrona, delegando gran parte del trabajo que suponen este tipo de operaciones al sistema operativo.

A su vez, al estar diseñado en JavaScript y funcionar con un ciclo de eventos, Node es idóneo para crear servidores web porque su rendimiento y el hecho de utilizar solamente un lenguaje de programación a lo largo de toda la aplicación hacen muy corta la curva de aprendizaje.

Node.js corre en un solo proceso, pero puede extenderse para que trabaje con múltiples procesadores. Su carga, en cuanto a recursos de memoria y CPU es mínima y es capaz de gestionar miles de conexiones por segundo.

Evented.io, ha sido desarrollado encima de Node.js, y además de todas sus virtudes, tiene las siguientes especificas:

- Posibilidad de utilizar *Websockets* para una comunicación Full-Duplex.
- Posibilidad de utilizar las Colecciones de Evented.io para manejar los datos desde el cliente.
- Con una configuración mínima obtenemos un servidor preconfigurado y preparado para funcionar.

Es decir, Evented.io proporciona una plantilla de servidor lista para ser desarrollada en conjunto con el cliente. Está escrita en JavaScript, y las librerías desarrolladas permiten hacer las actividades mas comunes, de una forma sencilla.

Además, Evented.io proporciona un patrón MVC orientado a API’s, lo cual le proporciona una sencilla escalabilidad simplemente creando nuevos recursos (REST). También le permite integrarse con más de una aplicación distinta dado que en síntesis se compone de: Colecciones de datos, APIs de acceso a estas colecciones, y *Websockets* para manejar las mismas colecciones.

6.3 Comunicación

Por último, en el contexto de la comunicación podemos concluir que la comunicación tiende a ser Full-Duplex.

Antes de existir HTTP las aplicaciones utilizaban sockets TCP. Es decir, eran conexiones Full-Duplex. La razón de que HTTP no cree conexiones de este tipo, es que en sus orígenes no esperaban la interactividad que se espera de las aplicaciones de hoy en día.

Por eso, como se ha descrito varias veces en este proyecto de fin de carrera, la comunicación está tendiendo a utilizar una comunicación Full-Duplex mediante un nuevo estándar conocido como *Websockets*.

Los *Websockets* proporcionan una interfaz de eventos que pueden generarse tanto en el servidor como el cliente, y los host responden a ellos a través de una conexión permanente (un socket).

En los años venideros, dado que los servidores empiezan a implementar este estándar, las aplicaciones prácticamente implementarán solamente este tipo de comunicación. Sin ir mas lejos, Meteor (descrito en las líneas futuras) implementa solamente este tipo de comunicación, en vez de HTTP, utiliza DDP, un protocolo de distribución y sincronización de datos basado en *Websockets*.

7 Agradecimientos

Me gustaría dar las gracias a todas las personas que han contribuido con su apoyo a mi esfuerzo.

En primer lugar me gustaría agradecer a mi familia por todos sus ánimos y apoyo durante este último año. Así como a mis amigos, principalmente a Daniel Biedma, que ha trabajado conmigo codo con codo en varios proyectos.

Quiero agradecer también a la comunidad de GitHub y Google Groups, por la contribución que realizan al mundo opensource, y todo lo que me han ayudado.

También le estoy agradecido a Marko Galarza por la oportunidad que me ha dado de presentar el proyecto en este departamento.

Por último, la última persona a la que quiero agradecer el estar ahí, es a mi chica Mónica Sarasa, porque gracias a ella y sus ánimos, todo me resulta más fácil y no me dejan de faltar las ganas de seguir aprendiendo y formándome.