Manual de BackboneJS

Manual para entender el framework Javascript BackboneJS y aprender a usarlo para el desarrollo de proyectos basados en patrones de diseño con Modelos, Vistas y otra serie de utilidades esenciales en aplicaciones web con uso intensivo de Javascript.

Autores del manual

Este manual ha sido realizado por los siguientes colaboradores de DesarrolloWeb.com:

Erick Ruiz de Chavez
Desarrollador web, Geek "de hueso colorado", Marido feliz.
http://erickrdch.com/
(7 capítulos)

Miguel Angel Alvarez
Director de DesarrolloWeb.com y EscuelaIT
http://www.desarrolloweb.com
(3 capítulos)

Qué es BackboneJS

BackboneJS es una librería Javascript para programación del lado del cliente que nos ofrece diferentes ventajas y utilidades en la programación, atendiendo a patrones, de una manera flexible.

BackboneJS es una librería para el desarrollo de sitios web, principalmente usada para aplicaciones web con bastante interacción con el cliente, donde se hace un uso intensivo de Javascript, Ajax, etc. Te permite desarrollar en Javascript atendiendo a patrones, con una variante del paradigma MVC, el mismo que se ha introducido con fuerza en la programación del lado del servidor y que, desde hace menos tiempo, viene empezando a ser costumbre en el desarrollo del lado del cliente.

Se trata de una heramienta muy indicada en aplicaciones de una sola página, aunque también es útil para otros tipos de sitios, multipágina, donde se encuentran interfaces de usuario avanzadas. Facilita la ordenación del código y la sincronización de las vistas con los datos de la aplicación. Todo ello redunda en una programación más modularizada y con mayor separación entre las partes de la aplicación, como presentación, datos, etc.

Utilizando palabras sencillas, BackboneJS es un conjunto de objetos que tienen métodos (o funciones, si lo prefieres llamar así) que van a proveer de estructura a nuestro código, facilitar el trabajo con datos, vistas para producir nuestro HTML, las interacciones para implementar comportamientos que deba realizar frente a acciones del usuario y los servicios para comunicar con cualquier tecnología que usemos del lado del servidor, ya sea NodeJS, PHP, Ruby, .NET, etc.

En definitiva, es un paquete de utilidades que nos permite hacer una programación de mayor calidad y aplicaciones web del lado del cliente más fácilmente mantenibles, sobre todo es útil cuando ya nos encontramos ante aplicaciones web de cierta complejidad.

> **Nota:** Este artículo y los siguientes están extractados del *hangout* Conocer Backbone.js #BackboneIO que nos ofreció Erick Ruiz de Chavez en uno de los eventos *online* de DesarrolloWeb.com.

Puedes encontrar el sitio oficial de BackboneJS en backbonejs.org

Facilidades de BackboneJS

Tal como se definen ellos mismos, Backbone.js ofrece una estructura para aplicaciones que hace uso intensivo de Javascript y que provee de modelos llave / valor enlazables mediante eventos, colecciones con la posiblidad de realizar diferentes utilidades por medio de una API, vistas con manejadores de eventos declarativos y conexión a interfaces REST nuevas o ya disponibles anteriormente.

De manera resumida, las facilidades de la librería Backbone son las siguientes:

- Permite la programación atendiendo al paradigma MV...C? (Existe un debate si es MVR porque en realidad no son controladores sino "Routeadores").
- Te ayuda a crear estructuras bien definidas para los datos de tu aplicación y facilita la creación de eventos cuando los datos cambian. Esto permite olvidarnos de la revisión de las variables o la propagación de los cambios en la aplicación cuando los datos de dichas estructuras cambian.
- Simplifica el uso de vistas, que te ayudan a pintar o "renderizar" interfaces de usuario en la página.
- Permite el uso de colecciones de modelos, en los que tendrás acceso a diferentes operaciones sobre los mismos, como filtrar, buscar, recibir notificaciones cuando cambien, etc.
- Simplifica y ayuda a ordenar el código de las peticiones AJAX para realizar solicitudes al servidor.
- Etc.

Nota: Para los que no lo conozcan, MVC es un patrón de desarrollo en el que se trata de separar las responsabilidades del código fuente de las aplicaciones, separándolo en diversas capas o partes, que son los Modelos (para trabajar con los datos), Vistas (código que trabaja con la presentación) y los Controladores (para implementar la lógica del negocio, es decir, la forma en la que interaccionan tus vistas, o lo que ve el usuario, con tus modelos, o tus datos).

Qué tipo de librería Javascript es Backbone.js

Quizás merezca la pena explicar dónde encaja Backbone en el panorama de los *frameworks* y librerías Javascript para que las personas que llegan nuevas aquí puedan despejar una de las preguntas típicas al conocer Backbone.js.

¿Librería o framework? Se trata de un API de programación, por lo que podemos llamarlo librería sin lugar a equivocación. Además, te permite trabajar con una variante del MVC, por lo que también te impone una manera específica de organizar tu código, lo que es una de las características de los frameworks, sin embargo, como veremos, Backbone es bastante flexible, permitiendo aplicarlo a un sitio web completo o a un único módulo, algo que quizás nos aleja de la idea de framework.

No obstante, esto no es más que una nomenclatura. Lo que debe quedar claro es que no viene a sustituir a otras librerías del lado del cliente como pueden ser jQuery, Mootools, Ext.js... sino que viene a aportar algo adicional que éstas no ofrecen, y que es fundamental para crear código más mantenible en arquitecturas web medianamente complejas. BackboneJS en sí es un paquete que tiene a jQuery como dependencia, o en otras palabras, su construcción está basada en jQuery, por lo que se puede entender que no viene a sustituir, sino a complementar.

Entonces, BackboneJS lo podemos considerar como una nueva herramienta que viene a facilitarnos el desarrollo y la cual tenemos que aprender para hacer desarrollo de alto nivel, útil en cualquier proyecto, pues se trata de un paquete de ventajas adicionales al programar en Javascript.

Dónde usar BackboneJS

Lo puedes usar donde te apetezca, nunca está de más implementar mecanismos que te ayuden a crecer si ese proyecto va evolucionando, sin embargo, BackboneJS es una librería especialmente indicada para el uso en **aplicaciones web que hagan un uso intensivo de Javascript.**

¡No lo uses para asignar un comportamiento a un botón! o en cualquier página donde piensas tener pocos elementos con interacción con el usuario y donde las funciones manejadoras sean bastante sencillas. El usar Backbone aporta un poco de complejidad al desarrollo, por lo que no sería tan adecuado para pequeños proyectos o páginas bastante estáticas. Si tu código es muy pequeño y lo puedes tener en unas decenas de líneas de código, no se recomendaría usar Backbone.

Usa BackboneJS en una aplicación web en la que ya se entiende que vas a tener bastante lógica de negocio, bastante interacción, bastantes métodos y eventos trabajando. Allí ya es un buen entorno donde sería excelente usar Backbone.js. Es útil también en cualquier página, que aun siendo sencilla, tiene los mismos datos y se muestran en varios sitios distintos, replicados en diversos elementos, ya que te permite mantener la sincronización con todos esos datos de una manera sencilla y prácticamente automática, sin que tengas que estar revisando constantemente cuándo cambian tus datos para actualizarlos en todos los lugares donde aparecen.

Conclusión

Creo que con lo visto hasta el momento tendríamos una buena idea de lo que es BackboneJS, aunque seguro que te habrás quedado con ganas de más y sobre todo con interés de ver otros artículos más técnicos en los que nos metamos más en materia. En breve estaremos produciendo este contenido.

Pero recuerda, este artículo está extractado de un hangout que realizamos en DesarrolloWeb.com y que tienes disponible en el vídeo "Conocer BackboneJS". Fue un evento realizado en directo que nos ofreció mucho más contenido interesante y de utilidad. En este artículo solo hemos cubierto hasta el minuto 22 de ese programa, así que si te interesa seguir ya mismo, te recomiendo prestarle atención.

Nosotros vamos a seguir transcribiendo el contenido presentado en este hangout, así que si deseas saber más también puedes continuar esta lectura con el artículo Los eventos en BackboneJS.

Artículo por Erick Ruiz de Chavez

Introducción a los Eventos en BackboneJS

Repaso conceptual a lo que son los eventos y las posibilidades que nos ofrecen en BackboneJS.

En este segundo artículo para hablaros del *framework* Javascript BackboneJS, vamos a introducir un concepto fundamental para entenderlo, los eventos. De momento nos centraremos más bien en la parte más teórica, aclarando conceptos y explorando las posibilidades que nos abre Backbone para los desarrolladores.

Recuerda que puedes ver la primera entrega de este artículo en Qué es BackboneJS y también que estos textos los estamos extractando de los *hangouts* que se emitieron en DesarrolloWeb.com sobre Backbone.JS.

Eventos en general

Con toda certeza debes tener una idea de lo que es un evento en Javascript, así que a partir de ahí, veamos cómo son las mejoras en el sistema de eventos que podemos implementar ayudados de BackboneJS.

¿Qué es un evento? Es un acontecimiento que ocurre durante la ejecución de un programa. Cuando "pasa algo". Por decirlo de otra manera, es un "mensaje". El evento es el puro mensaje, con información asociada y relevante al evento, que nos puede dar datos adicionales al contexto donde se ha producido.

> **Nota:** Lo que pase después de producirse el mensaje ya no es conceptualmente el propio evento, sino otras cosas que podemos hacer con programación, así que centrémonos en el "mensaje".

Por ejemplo, recibes un email. Ese es el evento, que tiene información relevante, como el objeto que lo ha producido (remitente del mensaje), a dónde se ha enviado, la hora etc. Toda es información nos viene con el evento (recuerda entender al evento como un "mensaje"), para entender su contexto. Lo que ocurra después de recibir el correo ya no es relativo al evento como tal.

Utilidad de los eventos de BackboneJS

De manera adicional al concepto de evento que ya conocemos y que está implementado en Javascript nativo, la clase Event de BackboneJS nos permite lanzar eventos y escuchar eventos sobre cualquier objeto que queramos. Se puede tener un objeto normal y corriente en Javascript, asociarle la funcionalidad de Event de Backbone y a partir de entonces, en ese objeto vas a poder lanzar eventos, escucharlos, suscribirte a eventos que se produzcan en otros objetos.

Aquí es donde se inicia la magia de Backbone, en la capacidad de trabajar con eventos en cualquier lugar. A través de la clase Evento tienes una interfaz de programación que te permite trabajar en cualquier objeto con eventos y sobre todo, la potencia de propagarse automáticamente hacia otros objetos que también puedan estar interesados en esos mensajes.

La clase Event de Backbone es muy importante porque se puede usar en cualquier objeto, por lo que tanto vistas, modelos y cualquier otro objeto tienen esa misma interfaz de "mensajes" para comunicarse entre sí.

> **Nota:** Luego, cuando entremos en detalle para conocer los otros elementos de la arquitectura de Backbone, entenderemos bien toda la potencia del sistema de eventos y cómo nos sirven de pasarela para la comunicación entre todos los integrantes de la aplicación, de modo que puedan estar sincronizados y enviarse "mensajes" unos a otros cuando ocurran cosas, a fin de programar las acciones pertinentes como respuesta.

Implementar los eventos en BackboneJS

De momento nos vamos a centar en la parte conceptual y vamos a dejar el código para más adelante, donde veremos ejemplos en detalle. Sin embargo, os podemos adelantar que Backbone se apoya de jQuery (recordemos que es una de sus dependencias) y por ello utilizaremos también los métodos on() y off() de jQuery dentro de Backbone.JS.

No obstante, existen algunas mejoras interesantes. De momento nos debemos de quedar que podremos "bindear" (asignar) y hacer "trigger" (lanzar) eventos totalmente personalizados. La idea es exactamente la misma que en jQuery, pudiendo con el método on() asignar la función manejadora de eventos que necesitemos.

Además, existe una serie de eventos que vienen "de serie" en Backbone que se lanzan cuando ocurren cosas típicas sobre modelos, colecciones y vistas. Ejemplos de ello serían "add", "remove", "change", "request", "sync", "error" etc. Veremos más adelante todo esto con mayor detalle.

En el siguiente artículo podrás conocer otros de los integrantes de BackboneJS más fundamentales, como son los modelos y las colecciones.

Artículo por Erick Ruiz de Chavez

Modelos y colecciones en BackboneJS

Qué son los modelos y las colecciones, y qué ventajas nos ofrece BackboneJS para tratarlos.

A lo largo del Manual de BackboneJS estamos explorando las posibilidades de este *framework* Javascript. Hemos dedicado los artículos precedentes a las generalidades de esta herramienta y a las posibilidades de los eventos.

En este artículo nos centraremos en las posibilidades de los modelos, que no son más que la manera que usamos en Backbone para almacenar los datos de la aplicación. Además, dedicaremos también unas líneas para introducir otro de los integrantes de Backbone.JS estrechamente ligado a los modelos, como son las **colecciones**.

Modelos en BackboneJS

A continuación veamos qué son los modelos en el framework Javascript BackboneJS y para qué nos sirven, de modo que podamos entender las ventajas que podemos disfrutar al usarlos.

Los modelos, como en otros sistemas enfocados a MCV, se refieren a los datos de la aplicación. Si tengo una aplicación que trabaja con suscripciones de correo electrónico, el modelo será cada una de las cuentas de usuario del sistema. Si tengo una aplicación que se encarga de mostrar una base de datos de películas, el modelo será cada una de esas películas.

Los modelos en BackboneJS se pueden crear con cualquier objeto en notación JSON. En ellos se escriben los datos que podemos tener dentro del modelo. Al extenderlos con la funcionalidad de BackboneJS vamos a conseguir varias mejoras:

- Eventos en el modelo. Puedo, tanto escuchar eventos, como dispararlos.
- "Bindeo" de eventos y "bindeo" de cambios.
- Sincronización con el servidor y notificación de cambios.

Esto, en pocas palabras, hace posible que, si modificas un valor de un modelo, la aplicación puede reaccionar en varios otros objetos que estén suscritos a los eventos de cambios en los datos de un modelo.

Para entender la potencia de esto, supongamos que estás mostrando en algún lugar los datos de un usuario. Si modificas con Javascript los datos de ese usuario en un lugar, tienes que estar repasando a mano todos los otros lugares donde estás mostrando esos datos de ese usuario y actualizarlos manualmente. Usando Backbone todo esto es automático y tienes los modelos desacoplados de las vistas, comunicando a través de eventos. Cuando cambias algo en un modelo, se producen eventos y esos eventos los puedes estar escuchando en varias vistas diferentes que podrían actualizarse automáticamente

cuando se producen cambios en el modelo.

Esto te permite no solo automatizar muchos de los procesos de visualización de la información actualizada a lo largo del sitio web, sino que facilita la reutilización de los modelos en otros proyectos, manteniendo de una manera sencilla las automatizaciones.

Sincronización con el servidor

Otra de las genialidades de BackboneJS está en el "sync", pues si yo lo deseo, al crear el modelo puedo proveerle de una URL y hacer "save" en el modelo para llamar al servidor a esa URL y guardar sus cambios. Dicho de otra forma, desde un modelo puedo ejecutar una función de guardado que automáticamente hace una solicitud por AJAX a mi servidor, enviándole los datos nuevos que debe guardar.

Es decir, no tienes que hacer mucho, simplemente tienes tus modelos y al hacer el "save" de sus valores, BackboneJS llama por si solo al servidor por AJAX. En esa llamada automática Backbone ya envía los datos con un formato que facilita su recepción.

> **Nota:** Obviamente, esto requiere que en el *backend*, con programación del lado del servidor en cualquier lenguaje de tu preferencia, te programes la página que va a recibir esos datos y que va a actualizar las estructuras donde los guardas, como base de datos. Sin embargo, una vez las tienes creadas es muy fácil reutilizarlas en otros proyectos donde también utilices ese modelo.

Colecciones en BackboneJS

Ahora nos vamos a dedicar a entender qué son las colecciones en BackboneJS y qué procesos pueden facilitarnos.

Una colección, como su propio nombre indica, es un conjunto de datos. En BackboneJS generalmente una colección será un conjunto de modelos.

> **Nota:** Realmente por la arquitectura del framework en una colección no estamos obligados a guardar únicamente modelos. Sin embargo, como podremos observar en la práctica, en las colecciones acabarás guardando modelos y es allí donde las cosas tendrán más sentido al usar BackboneJS y podrás sacarle mayor partido a estas estructuras.

Con las colecciones, del mismo modo que ocurría con los modelos, tenemos a nuestra disposición la posibilidad de escuchar y producir eventos y la interfaz "sync" para la sincronización con el servidor.

Gracias a la sincronización, si tienes una colección y le has hecho cambios, podrás hacer un "save" y entonces los datos que hay en los modelos de la colección se enviarán al servidor mediante AJAX y de manera automática para el desarrollador. De manera similar, cuando carga la aplicación, también en las colecciones podremos hacer "fetch" para cargar datos en ellas que nos vienen del servidor. Automáticamente, la colección hace por AJAX el "get" al servidor y va a cargar en ella los datos que éste le devuelva.

Conclusión

La idea de este artículo ha sido más bien el aclarar conceptos y entender por qué BackboneJS es una herramienta tan útil. Todavía tenemos otras estructuras y clases que debemos identificar para poder observar el alcance global de backbone.JS y seguiremos con ello en el siguiente artículo.

> **Nota:**Las implementaciones en código fuente de modelos y colecciones los veremos un poco más adelante, cuando comencemos con las prácticas.

En el siguiente artículo podrás conocer con detalle qué son las vistas para Backbone.JS y cuáles son sus utilidades en el desarrollo web basado en este framework.

Artículo por Erick Ruiz de Chavez

Vistas en BackboneJS

Qué son las vistas y cómo nos ayudan a la hora de crear aplicaciones web y cómo BackboneJS las aprovecha para asociarlas a modelos y colecciones que contienen datos que pueden cambiar dinámica y automáticamente.

Ahora vamos a tratar de introducir las vistas y explicar cuáles son las ventajas que nos ofrecen en BackboneJS. Este es un artículo del Manual de BackboneJS, que entenderás si te has leído los capítulos anteriores.

Conociendo las vistas podremos entender finalmente cómo todos los "engranajes" que nos ofrece BackboneJS para el desarrollo web funcionan, ligados entre sí por el sistema de eventos, para realizar aplicaciones complejas con mayor facilidad.

Vistas en general

Las vistas contienen la parte del código que sirve para presentar la información. Como estamos desarrollando web, las vistas contendrán el código HTML que tenemos que generar para presentar los datos en la página.

Este es otro de los elementos clave de BackboneJS y de cualquier sistema que trabaje en base a patrones de diseño de software MVC. De hecho, en nuestras aplicaciones web, con lo que más vamos a trabajar es con los modelos y las vistas. Los modelos ya sabemos que sirven para almacenar los datos y las vistas que sirven para mostrarlos en código HTML.

Si lo queremos ver así, las vistas son una abstracción de cómo vamos a ver los datos en nuestra página. Pensando en un sitio web, podríamos tener una vista general que nos guarda la "plantilla" sobre el contenido que va a tener toda la página. Pero si lo deseamos hacer un poco más modular, podríamos tener una vista general y otra serie de vistas hijas, que contienen cada uno de los elementos de la página, por ejemplo, una para el buscador, otra para el recuadro de últimos artículos publicados, otra para el de *login* de usuarios, etc.

Potencia de las vistas en BackboneJS

El sistema de vistas, además, se enlaza con los modelos, de modo que podemos mostrar los datos de esos modelos en las vistas, pero además hay determinadas vistas que las podremos enlazar con colecciones en vez de modelos para mostrar un conjunto de datos. Por ejemplo, en los comentarios de artículos podríamos tener enlazada una colección con todos los modelos de los comentarios que deben aparecer. Es decir, en nuestra vista de comentarios tendríamos asociada una colección donde cada comentario sería cada uno de los modelos de la colección.

Siguiendo ese ejemplo de los comentarios, cuando un usuario hace un comentario en la página y pulsa en "submit", desde el punto de vista de BackboneJS, estaríamos haciendo un "save", que mandaría ese comentario al servidor. Además, podríamos estar haciendo un "fetch" en la colección que actualizaría los comentarios con los datos que haya en el servidor actualmente. Estos cambios en los modelos y las colecciones estarían lanzando eventos, los cuales a su vez estarían siendo escuchados por las vistas para actualizar su contenido.

En fin, si una vista detecta que ha habido cambios en los modelos o colecciones que tiene asociados podría actualizarse automáticamente, mostrando el contenido nuevo, donde puede que ya no haya los mismos elementos, pues quizás se agregaron nuevos o se eliminaron algunos antiguos. Todo esto nos permite tener vistas que son perfectamente dinámicas, sin tener que volvernos locos actualizando su contenido manualmente desde mil sitios en el código de la aplicación.

No es un motor de plantillas

Para no llevarnos a engaño, tal como están implementadas las vistas en BackboneJS, cabe señalar que no son lo que podríamos entender como un motor de plantillas. Es decir, con las vistas en Backbone no vamos a determinar el HTML o el CSS de nuestra página, eso lo podemos dejar en manos de nuestro motor de plantillas preferido, pero no en Backbone.JS.

Nota: Sistemas para hacerse valer de plantillas hay unos cuantos en Javascript. Podemos ver un artículo que nos ofrece una descripción de lo que son los motores de plantillas, por qué son útiles y cuáles son los más populares en Javascript.

Entonces ¿qué es exactamente lo que está implementado en las vistas de Backbone.js? Pues simplemente son unas clases que extiende el sistema de vistas genérico de Backbone y sobre el que indicamos los eventos que van a escuchar o producir, los modelos o colecciones con los que están enlazados, etc. Es decir, lo que básicamente nos sirve para implementar todo el sistema de sincronización con los modelos y propagación de eventos.

Aunque no sea un sistema de plantillas, las vistas siguen siendo las responsables de renderizar HTML en la página. Para ello existirá un método que se llama render() que sí tiene que ver con la creación de HTML y manipulación del DOM, al que se llama cada vez que tiene que refrescarse la vista. En ese método deberíamos producir el HTML de nuestra vista por los medios que nosotros deseemos, aunque lo lógico es usar algún *template engine* que nos guste y nos facilite la labor.

Más adelante nos dedicaremos a ofrecer ejemplos concretos con código y obtendremos más información de las vistas. De momento, lo importante es que hayamos podido aclarar conceptos y entender las posibilidades que nos ofrece el *framework* BackboneJS.

Artículo por Miguel Angel Alvarez

Route o manejar las URL con BackboneJS

Explicamos la clase Route de BackboneJS para manejar las URLs de las aplicaciones web, además de explicar lo que son las clases History y Sync.

Dentro del Manual de Backbone estamos describiendo todas las clases que componen este *framework* Javascript. Ahora le toca el turno al ruteador y además acabaremos esta parte teórica del manual describiendo también las clases "History" y "Sync".

Las rutas "route" de BackboneJS nos sirven para manejar las URLs de la aplicación, es decir, saber cuál es la dirección en la que estamos y poder actuar de maneras distintas dependiendo de ellas.

Para explicarlo bien, vamos a comenzar con un ejemplo sobre cómo manejaba las URLs anteriormente Twitter. Aunque hoy ya no nos muestra URLs como esta, todavía siguen funcionando perfectamente: https://twitter.com/#!/erickrdch

Los navegadores cuando reciben una URL que incluye el signo "#" (almohadilla, gato, numeral) entienden que es una referencia a un área determinada dentro de la página definida por lo que hay antes de la almohadilla. Si cambiamos lo que hay después de la "#" no navegan a una nueva URL, sino que llevan la página hacia arriba o hacia abajo para mostrarnos otra zona de la página. Esto no debe ser nada nuevo para los lectores, pues es justamente lo que ocurre con los enlaces internos, hacia otras partes de la misma página.

Javascript, y en concreto BackboneJS, se valen de este comportamiento para producir lo que se llaman "deep links" o sea, enlaces a una parte interna y específica de tu aplicación.

Para que entendamos para qué nos sirve esto de los "deep links" pensemos en otro ejemplo. Imagínate que tienes una rejilla, tabular, o si lo prefieres llámale sencillamente tabla con datos ("table grid" se suele usar en Inglés). Si pulsas sobre una de las líneas de esa rejilla, imagina que entras en la edición del registro que había en esa línea y Javascript te la transforma para mostrar un formulario donde puedes modificar los datos. Pues bien, generalmente, por medio de una URL simple no vas a tener la posibilidad de acceder directamente a ese estado de tu aplicación, es decir, podrás entrar en la página que muestra el *table grid*, pero no vas a poder especificarle a tu aplicación que quieres que te muestre directamente una línea determinada "abierta para edición".

Un enlace profundo es justamente para crear esas URLs en las que estás accediendo directamente a un estado complejo y determinado de tu aplicación. Vamos a suponer que tengo misitio.com que muestra esta lista de usuarios en un table grid. Si

tú haces misitio.com/usuario/23 te estoy dando una forma de mostrar la aplicación donde directamente verás la rejilla con todos los usuarios y donde puedas ver directamente el formulario para edición del usuario 23.

Cómo trabaja BackboneJS con los deep links

BackboneJS captura estas diferencias o cambios en las URLs, cuando se usan enlaces profundos, para poder realizar cosas específicas que se deseen implementar por medio de los deep links.

Volvemos a lo mismo de siempre, Backbone.JS utilizará los eventos para permitir que las diferentes partes de la aplicación se enteren que ha cambiado la URL y se sepa qué características tiene el nuevo enlace y poder realizar los cambios oportunos en los modelos, colecciones o vistas.

Lo genial de esto es que podrás usar enlaces dentro de tu aplicación web que vayan siempre a la misma página, con lo que tu navegador no la actualizará nunca. Esto produce lo que conocemos como aplicaciones web de una sola página, que se comportan parecido a las aplicaciones web de escritorio y que tienen un desempeño mucho mejor. Sin embargo, el detalle es que la URL siempre va cambiando, de modo que si un usuario guarda en favoritos tu aplicación, en un estado determinado, lo que está guardando es el "deep link", de modo que cuando vuelva a entrar en la aplicación, se le abrirá mostrando el mismo estado donde había sido guardada.

Esto de los deep links te vale para un gran abanico de situaciones, como por ejemplo cuando el usuario copia y pega el enlace actual para mandarlo por e-mail o tuitearlo, por ejemplo. Podrá mandar, no solo el enlace a la "home" de tu sitio, sino también a una sección o un estado concreto, lo que le aporta un gran valor a tu aplicación.

History API de HTML5, los deep links y la almohadilla o hash

En estos momentos en la mayoría de los navegadores no es necesario siquiera usar la almohadilla (gato, numeral o el símbolo "hash" como le llaman en inglés). Esto es porque ahora los navegadores modernos soportan lo que se llama "Session history management" o "History API" del HTML5, que nos permite cambiar la URL de la barra de direcciones del navegador, pero sin que éste actualice la página.

Por medio de pushState() tenemos un método que nos permite cambiar la dirección que se está mostrando en nuestro navegador mediante Javascript, pero sin que se realice una recarga del contenido que ya se está mostrando en el navegador. Pero además, este método trabaja directamente con el historial del navegador, de modo que podemos pulsar los botones de atrás y adelante del navegador y podemos retroceder y avanzar entre los distintos estados de la aplicación. Incluso podemos guardar ese estado en los favoritos perfectamente.

Pues bien, BackboneJS tiene la capacidad de trabajar con el pushState() en navegadores que lo soporten y en los casos en los que no hay compatibilidad, usar el viejo método de la almohadilla y los enlaces internos en la misma página, lo que produciría el mismo efecto. En la práctica, el uso de los hash ya no es necesario y paulatinamente va a ir desapareciendo en las aplicaciones web, en la medida que podemos producir los mismos comportamientos con URLs corrientes que no la contienen.

> **Nota:** La mayoría de los navegadores actuales es compatible con el API de historial del HTML5. Los que no tienen soporte son algunos navegadores para móviles y versiones de Internet Explorer 9 o inferiores.

Router de BackboneJS

Realmente todo esto se hace de manera muy transparente para el desarrollador, por medio del "ruteador" de Backbone.JS, la clase Route. Simplemente tienes que cambiar la dirección por medio de la mencionada clase Route y el framework ya se encarga de todo, es decir, ya te cambia la URL y gestiona el historial del navegador de modo que puedas ir hacia atrás y adelante entre los diferentes estados de tu aplicación.

Afortunadamente, como decíamos, en caso que el navegador no soporte el API History del HTML5, Backbone empieza a utilizar el hash de modo automático.

¿Las rutas son como los controladores del MVC?

Recordemos que BackboneJS cambió un poco el paradigma, o al menos la manera de referirse a los típicos MVC por MVR. O sea, cambió los "Controller" por "Route", pero ¿qué tienen que ver los unos con los otros?

Las diferencias dependen mucho del framework o librería que se esté usando. Pero en líneas generales, un Controller se encarga de la lógica de negocio, mientras que el Router es algo más sencillo, porque simplemente se encarga de dirigir las peticiones o las acciones de los usuarios a los lugares concretos donde está el código que tiene que realizar lo que se solicita. Si lo quieres ver así, el Router es como una manera de recibir parámetros, que te indican qué es lo que quieres hacer en ese momento o qué estado de la aplicación debe mostrarse.

En cierto modo, el ruteadores de backbone es algo parecido a los controladores que podemos conocer en los sistemas MVC, en el sentido que los controladores permiten generar URL de tu aplicación que realmente no existen como archivos dentro de tu *hosting*, sino que son gestionados por el software para emular que esas páginas están físicamente en el servidor.

Gracias a BackboneJS gestionamos esas rutas como si fueran URL que existen físicamente en el servidor, aunque esos archivos realmente no se encuentren. Todo esto no es gratuito, es decir, implica que tendremos que configurar nuestro Apache, o el servidor web que estemos usando, para que interprete esas rutas y las redireccione hacia archivos que sí que existan físicamente, pero de manera transparente para el navegador. Todo eso se verá más adelante, pero si conoces los MVC seguramente ya sepas por dónde van los tiros.

History en BackboneJS

Para poder finalizar con la descripción de las clases disponibles en Backbone, tenemos que mencionar, el History y el Sync.

History es otro de los módulos de Backbone.JS de los que hemos hablado ya de pasada al describir el funcionamiento de los Route. Realmente, es la parte de Backbone que permite gestionar el historial del navegador, de modo que puedas ir pasando entre secciones de tu aplicación y que se vayan guardando en el historial de navegación.

Hemos dicho que esto lo hace BackboneJS de manera automática, pero además tienes una interfaz de código para poder hacer cosas adicionalmente a las que ya hace el framework por si solo.

Sync en BackboneJS

Es simplemente la comunicación de las colecciones y modelos con el lado del servidor, que permite hacer llamadas para leeer o guardar estados de los datos en el servidor. Todo, por supuesto, por medio de AJAX, usando el método jQuery.ajax y JSON como formato de datos para la comunicación cliente/servidor.

Existe un modelo estándar implementado en Backbone que nos facilita mucho la sincronización de los datos, pero también si tienes un sistema diferente, de manera sencilla puedes implementar esa pasarela de escrituras y lecturas desde el cliente al servidor.

Conclusión

Hasta aquí hemos conocido Backbone.js desde un punto de vista puramente teórico. Esperamos que con la información ofrecida hasta el momento hayamos podido entender bien qué es lo que ofrece este framework y hacernos a la idea de las ventajas que nos puede proporcionar el utilizarlo.

Aunque esta información todavía debería complementarse con explicaciones más técnicas sobre la implementación de las diferentes clases que componen BackboneJS, creemos que resulta imprescindible para entender más adelante cómo se usa el framework.

En los próximos artículos nos dedicaremos a conocerlo más a fondo y a realizar un ejercicio práctico que nos permita ver algo de código y entender cuáles son los mecanismos para el desarrollo de sitios basados en BackboneJS.

Artículo por Erick Ruiz de Chavez

Instalar Backbone.js en un proyecto web

Cómo dar los primeros pasos con BackboneJS e instalar el framework en un sitio web mediante la inclusión de distintas librerías.

A lo largo del Manual de Bakcbone.js hemos podido obtener una introducción teórica de lo que podemos hacer con este *framework*. Ahora que ya sabemos qué nos ofrece podemos ponernos manos a la obra y **empezar a ver algo de código**. Estoy seguro de que la mayoría estará con ganas, así que comencemos aprendiendo cómo podemos hacer disponible el framework BackboneJS en una página web.

Antes de comenzar, podemos hacer un análisis y es que BackboneJS es especialmente útil para "onepagesites", sitios que solo tienen un único documento HTML y donde toda la interacción con el usuario se implementa con Javascript, haciendo uso de AJAX para comunicar con el servidor. Por ello, en la mayoría de los proyectos tendremos un único archivo index.html. En ese index.html tendrás que incluir las librerías necesarias, con *scripts* que tendrás descargados en archivos .js, o utilizando los correspondientes CDN.

Descarga Backbone.js y Underscore.js

De momento vayamos a la página de BackboneJS para hacer la descarga del framework. Ofrecen el archivo en diversas versiones, para desarrollo o producción. Actualmente, el archivo de descarga para producción, "Gzipeado" y minimizado, ocupa 6.3kb, en la versión 1.0.0.

Además, BackboneJS tiene una dependencia con Underscore.js, que es una librería que amplía el API de Javascript nativo con muchas funciones y métodos adicionales, básicos para realizar tareas habituales en Backbone.js.

Puedes obtener esta dependencia de Backbone en la página de la librería Underscore.js, obra de los mismos autores que Backbone.js.

Nota: Habíamos anunciado ya al principio del Manual de Backbone.js que estaba basado en la librería jQuery, pero en realidad UnderscoreJS es la única paquetería imprescinible para manejarse en BackboneJS. Para utilizar otras partes del framework (como manipular el DOM desde las vistas) vas a tener que cargar también jQuery igual o superior a 1.7, o la librería Zepto. Zepto.js es una librería compatible con el API de jQuery, pero resumida en funcionalidad, lo que la hace mucho más ligera. También, para navegadores antiguos podrías necesitar de un "polyfill" que implemente JSON, si el navegador del usuario no tiene soporte nativo a la Notación de Objeto Javascript.

Instalando los scripts

Después de descargar estas librerías, crearás tu index.html incluyéndolas en el código con las siguientes etiquetas SCRIPT.

```
<script src="jquery.js"></script>
<script src="underscore.js"></script>
<script src="backbone.js"></script>
```

Esta parte no tiene misterios. A partir de ahora podremos disfrutar de Backbone.js en nuestro proyecto.

Creando un primer script con Backbone.js

Backbone.js no es una librería muy que se preste a hacer el típico "hola mundo" donde tengamos una idea buena de su potencia o flujo de trabajo habitual. Recuerda que si quieres aplicar un evento a un botón o hacer todo tipo de cosas simples, este framework no es para ti. Por eso, pensar en un primer script sencillo y representativo se hace un poco difícil.

No obstante, vamos a hacer unas pocas líneas de código en las que crearemos un modelo y ejecutaremos algunos métodos disponibles. La idea no es explicar todavía todos los detalles de estas líneas de código, así que te pedimos algo más de paciencia si piensas que dejamos en el aire explicaciones importantes.

> **Nota:** El siguiente código se ofreció de muestra en el *hangout* de DesarrolloWeb.com #BackboneIO primer ejemplo BackboneJS, minuto 35.

Con las siguientes líneas creamos la clase de un modelo y un objeto de esa clase/modelo recién creado, al que le damos unos datos.

```
//creamos la clase del modelo
var Usuario = Backbone.Model.extend({});
//creamos una instancia de ese modelo
var miUsuario1 = new Usuario({
   nombre: "Erick"
});
```

Con esto no conseguimos nada representativo, ningún resultado aparente ni ningún mensaje, pero ahora fíjate en las siguientes líneas de código.

```
//asignamos un valor a la propiedad "url"
miUsuario1.url = "/usuarios";
```

La propiedad "url" la veremos más adelante, pero podemos decir ya que es la dirección donde entiende Backbone se realizarán las operaciones de guardado o lectura de los datos del modelo.

```
//guardo este usuario
miUsuario1.save();
```

Entonces verás en la consola de tu navegador que Backbone.js intenta hacer un "POST" de ese usuario al servidor. Esto lo hace Backbone por AJAX sin que tengamos que configurar nada, simplemente enviando los datos a la URL que habíamos definido anteriormente.

> **Nota:** Ese POST lo podrás ver si tienes abierta la consola de Javascript. Realmente lo que apreciarás es que intenta hacer el POST, pero como no existe la URL "/usuarios" se produce un error 404.

Luego, si yo quisiera recuperar los datos de ese usuario desde el servidor, podemos invocar a fetch() y Backbone ejecutará una operación GET, por AJAX, también de forma automática y sin que tenga que hacer nada explícitamente.

```
//recupero datos de un usuario
miUsuario1.fetch();
```

Los errores que podrás ver en la consola, como resultado de ejecutar estas sentencias llamando a los métodos save() y fetch() del modelo, te los muestra Backbone.js porque intentas acceder a la supuesta URL (que no existe), donde se supone debería realizarse la gestión de los usuarios. En concreto, podrás apreciar un mensaje en la consola de errores Javascript parecido al siguiente.

Failed to load resource: the server responded with a status of 404 (Not Found).

Aunque nos arroje un error, podemos comprobar hasta qué punto nos podrá simplificar la vida el trabajo con modelos y cómo es sencillo para volcarlos al servidor o traer sus datos actualizados. De todos modos, esto no es más que el principio.

En el siguiente artículo te explicaremos cómo implementar eventos en Backbone.js y podremos seguir aprovechando diversas de las cualidades más importantes de este framework.

Artículo por Erick Ruiz de Chavez

Implementar y disparar eventos en Backbone.js

Cómo implementar eventos con el módulo Backbone.Event de Backbone.js, asignar manejadores de eventos personalizados con on() y ejecutarlos con trigger().

Este capítulo del Manual de Backbone.js lo vamos a dedicar a la implementación de eventos en el *framework*. En estos momentos ya debemos tener funcionando Backbone.js en nuestra página web y gracias a los conocimientos del artículo dedicado a Instalar backbone.js en la página, hemos podido poner en marcha nuestro primer *script*.

Si has leído la documentación publicada sobre BackboneJS hasta este punto en DesarrolloWeb.com, habrás podido entender la parte teórica de los eventos y por qué son tan fundamentales y útiles. Como sabes, la "magia" de BackboneJS comienza con los eventos, por ello no es mala idea que empecemos nosotros también por este asunto, metiéndole mano al código fuente.

En resumen, los eventos nos sirven para mantener comunicados a todos los actores en nuestra aplicación, tanto modelos, colecciones, vistas... ya que podemos definirlos sobre cualquier objeto y también escucharlos desde cualquier objeto, para realizar acciones cuando se produzcan.

Backbone.Events

Este es el módulo que se usa para trabajar con eventos en el *framework*, Backbone.Events y puede implementarse sobre cualquier objeto, no necesariamente objetos de Backbone, sino cualquier objeto Javascript en general. Gracias a su funcionalidad tendremos la posibilidad, entre otras cosas, de crear eventos personalizados, asignando funciones manejadoras y de desatar (invocar, disparar) esos eventos personalizados.

Para definir un evento sobre un objeto tenemos primero que extender dicho objeto con la clase Backbone.Events, para que podamos asignarle luego esos eventos personalizados.

```
//creo un objeto cualquiera con Javascript, usando notación JSON
var objeto = {};
//extiendo ese objeto con la clase Backbone.Events
_.extend(objeto, Backbone.Events);
```

> **Nota:** si te fijas, para extender un objeto usamos el método _.extend(), que es perteneciente a la librería Underscore.js.

A partir de ese momento, sobre nuestro objeto Javascript podremos invocar los métodos de Backbone.Events.

Asignar manejadores de eventos con on()

Usamos el método on() para asignar una función manejadora de eventos sobre un objeto, para un evento cualquiera. Este método recibe al menos dos valores, siendo el primero el nombre del evento que queremos crear y el segundo la función manejadora que asocial, que será invocada cuando se produzca ese evento.

```
objeto.on("mi_primer_evento", function(msg) {
alert("Se ha desencadenado el " + msg);
});
```

En este caso "mi_primer_evento" es el nombre del evento que estamos creando. La función anónima que se envía como segundo parámetro es el manejador de evento que estamos creando, también llamada función "callback".

> **Nota:** En proyectos que tengan un número muy elevado de eventos nos recomiendan en la documentación de BackboneJS usar, por convención, el carácter dos puntos ":" para crear un espacio de nombres, por ejemplo "encuesta:empezar", "combociudades:desplegando".

Podemos asignar una misma función manejadora de eventos a varios eventos personalizados al mismo tiempo simplemente separando estos nombres de eventos con un espacio en blanco.

```
objeto.on("cambiar:hora cambiar:fecha", …);
```

Disparar eventos con trigger()

Una vez que hemos asociado una función manejadora con un objeto a ejecutar como respuesta a un evento personalizado, podemos "disparar" esos eventos con el método trigger().

El modo es muy sencillo, simplemente le indicamos el nombre del evento que queremos que se produzca y los parámetros

que debe recibir el manejador de eventos.

```
objeto.trigger("mi_primer_evento", "evento que acabamos de crear.");
```

Con el código anterior hemos producido el evento personalizado "mi_primer_evento" que acabábamos de crear. Además le hemos indicado a trigger() un segundo parámetro que es recibido por la función callback asociada a ese evento.

En realidad, trigger() solo requiere un parámetro, que es el nombre del evento que queremos disparar, el resto de parámetros que le pasemos se enviarán a la función manejadora de eventos que se esté ejecutando.

```
//pasando varios datos a la función manejadora
objeto.on("varios_datos", function(dato1, dato2){
alert("se produjo el evento 'varios_datos' y he recibido " + dato1 + " y " + dato2);
});
objeto.trigger("varios_datos", "valor1", "valor2");
```

Variable "this" como contexto dentro de las funciones manejadoras de eventos

Es importante destacar el papel de la variable "this" dentro de las funciones manejadoras de eventos, que nos da una referencia al contexto donde se está ejecutando este código; dicho de otro modo, el objeto sobre el que se ha producido el evento.

Esto no debería generar muchas dudas, pero veamos no obstante el siguiente código.

```
//creo un objeto nuevo, al que le cargo algunos datos
var persona = {
nombre: "Miguel"
};
//extiendo el objeto para que soporte eventos de backbone.js
_.extend(persona, Backbone.Events);
//asigno una función manejadora a un nuevo evento
persona.on("segundo_evento", function(){
alert("Manejador de 'segundo_evento'...");
alert("muestro this.nombre: " + this.nombre)
});
//disparo ese nuevo evento
persona.trigger("segundo_evento");
```

Como resultado de ejecutar ese código, comprobarás que se muestra el nombre de la persona "Miguel", que hemos obtenido a través de this.nombre. O sea, "this", es una referencia al objeto que ha recibido el evento que se acaba de producir.

El módulo de eventos contiene otras funciones interesantes que veremos un poco después, como off() o one(). Os citamos para próximas entregas en las que seguiremos conociendo las posibilidades de los eventos en este framework.

Artículo por Miguel Angel Alvarez

Eliminar eventos Backbone.js y cambiar su contexto

Más sobre los eventos en el framework Backbone.js, eliminarlos con off(), crearlos para ejecutarse una sola vez con once() y cambiar el contexto de ejecución o valor de la variable this.

Los eventos ya nos han ocupado un par de artículos en el Manual de Backbone.js . Hemos aprendido acerca del concepto de Evento, tal como lo entiende el framework, como nos ayudan a comunicar entre los elementos de la aplicación y también hemos visto una primera aproximación a su implementación en código fuente.

En este texto aprenderás más cosas sobre eventos y a utilizar algunos otros métodos como son "off()", "one()" o a cambiar el contexto de ejecución de un evento dado. La intención es aprender bien cuáles son las posibilidades de los eventos más esenciales para el desarrollo de sitios web con alto contenido en Javascript.

Desactivar eventos con off()

El método off() nos sirve para eliminar un manejador de eventos de un objeto. El mecanismo es muy sencillo.

La primera posibilidad es desactivar un evento, indicando el nombre del evento que queremos "apagar".

```
objeto.off("segundo_evento")?
```

Con eso estamos eliminando todas las funciones manejadoras de eventos asociadas al evento "segundo_evento".

> **Nota:** Para no liarnos con nomenclaturas cabe aclarar de nuevo que estos manejadores de eventos son también llamadas funciones *callback* en la documentación de Backbone.

La otra posibilidad es desenlazar una única función manejadora de eventos en un evento dado. Esto se consigue iniciando el nombre de la función que queres desenlazar, pero claro, para ello tienes que haber asociado el evento previamente con una función no anónima.

Veamos este ejemplo por pasos. Porque primero tenemos que crear la función y luego asociarla como manejadora de eventos.

```
function manejadora(){
alert("Estoy en la función Manejadora")?
alert("muestro this.nombre: " + this.nombre)
}
objeto.on("segundo_evento", manejadora)?
```

Ahora podemos eliminar esta función manejadora con off().

```
objeto.off("segundo_evento", manejadora)?
```

Ejecutar el evento una única vez con once()

Este método permite asociar una función manejadora de eventos a un objeto, igual que venimos haciendo con on(). La diferencia es que once() solo ejecutará la función una única vez cuando se produzca el evento. Es decir, la segunda y siguientes veces que ese evento se produzca no se ejecutaría la función callback.

Tal como nos comentan en la documentación, es útil cuando queremos decirle a un objeto "la siguiente (y solo la siguiente) vez que pase algo (algo es el evento) haz esto".

Cambiar el ámbito de ejecución de un evento en Backbone.js

Esta funcionalidad es interesante verla con mayor detalle, pues será útil para determinadas situaciones que veremos más adelante. Como sabemos, pues se explicó en el artículo anterior, dentro de una función manejadora de eventos podemos hacer uso de la variable "this", que es una referencia al objeto sobre el que se está ejecutando un método, en este caso un evento.

Bien, pues a la hora de asociar una función manejadora a un evento podemos definir un nuevo contexto para la variable this, no necesariamente el objeto sobre el cual se produjo el evento. Esto lo podemos ver con un ejemplo.

```
//creo un primer objeto
var fiatUno = {
marca: "Fiat"
}
//extiendo el objeto para que soporte eventos de Backbonejs
_.extend(fiatUno, Backbone.Events)?
//creo un segundo objeto
var seatToledo = {
marca: "Seat"
}
//creo una función
function mostrarMarca(){
alert("La marca del objeto: " + this.marca)?
}
//defino un evento sobre el primer objeto, cambiando el contexto de this
```

```
fiatUno.on("mievento", mostraMarca, seatToledo)?
//disparo el evento sobre el primer objeto
fiatUno.trigger("mievento")?
```

Como se puede observar, al definir el evento con el método "on()" se enviaron 2 parámetros. Los dos primeros ya los conocíamos: el nombre del evento "mievento" y la función manejadora "mostrarMarca". El tercer parámetro es el nuevo contexto de la variable this.

¿Qué ocurrirá? quizás lo hayas adivinado. Si no hubiéramos modificado el contexto de this, nos mostraría la marca del primer objeto "Fiat", pero como se cambió el contexto, pasando el objeto seatToledo como valor de this, la marca que se mostrará será "Seat".

Conclusión

En el próximo artículo del Manual de Backbone.js en DesarrolloWeb.com vamos a dejar de momento los eventos, introduciéndonos en el mundo de los modelos. Podremos hacer ejemplos más interesantes y poco a poco iremos sacándole un poco más de utilidad a todos estos conocimientos.

Sin embargo, cabe decir que nos queda una parte importante de los eventos que es conseguir que unos objetos escuchen los eventos que se produzcan en otros, algo muy útil para poder acompañar desde un objeto los acontecimientos que se están produciendo en otros objetos de la aplicación.

Además, veremos también más adelante que Backbone.js implementa una serie de eventos que se producen de manera predeterminada cuando ocurren determinadas cosas en los objetos de la aplicación.

Todo este conocimiento será conveniente aplazarlo cuando sepamos algo más del framework, como son los modelos o las vistas, lo que nos permitirá crear ejemplos más representativos.

Artículo por Miguel Angel Alvarez

Implementando modelos en BackboneJS

Cómo se implementan los modelos en BackboneJS, cómo podemos almacenar datos en ellos y acceder a ellos.

Durante el Manual de BackboneJS hemos ofrecido ya varios acercamientos a lo que son los modelos. El concepto y algunas de sus posibilidades lo aclaramos en el artículo sobre Qué son los modelos en BackboneJS. Además, en el primer artículo, donde empezamos con el código fuente, ya metimos mano a los modelos, aunque sin explicar mucho cómo se creaban estas estructuras.

Como ya hemos introducido de manera teórica los modelos, no vamos a entretenernos de nuevo explicando el concepto. Nos debe quedar claro, no obstante, que los modelos son los datos de nuestra aplicación. Ellos almacenan el "qué" (todo aquello que tengamos en la aplicación), dejando a parte el "cómo" (cómo deben ser mostrados los datos).

> **Nota:** Quizás esté de sobra mencionarlo, pero para entender todo esto, tendrás que estar familiarizado con la notación de objeto Javascript, pero si no fuese así te recomiendo ver el vídeo qué es JSON.

Creando una clase modelo con Backbone

Comencemos entonces viendo cómo se implementan los modelos. Teniendo en cuenta que todas las estructuras de BackboneJS aprovechan el paradigma de la Programación Orientada a Objetos (POO), entenderemos que los modelos se definen por medio de una clase y cada uno de los especímenes se crearán mediante objetos. Por ejemplo, las películas de un

videoclub (en general) serían prototipadas mediante una clase y luego, cada una de las películas independientes, serían distintos objetos.

Así pues, comenzamos creando la clase del modelo, el tipo o definición, sobre el que podremos instanciar los objetos del modelo más tarde.

```
var Pelicula = Backbone.Model.extend();
```

Como has visto, las clases de los modelos en BackboneJS se definen haciendo un "extend" del modelo básico de Backbone y para ello usamos el método extend() que pertenece a Backbone.Model. Esto nos devuelve otra clase, que extiende a los modelos básicos, para implementar objetos con los datos de nuestra aplicación.

> **Nota:** Observa que hemos escrito "Pelicula" con la "P" mayúscula, eso es por convención. En la POO se utiliza la primera letra mayúscula en los nombres de clases. Fíjate también que Backbone y Model también tienen las primeras en mayúscula porque son también clases y no objetos.

Es interesante comentar que en BackboneJS, el modelo de prototipos de Javascript se mantiene de unas clases a sus extendidas. Por ello, podríamos perfectamente extender Película para crear otras clases que hereden de ésta.

```
var PackPelicula = Pelicula.extend();
```

Creando los especímenes objetos modelo en BackboneJS

Una vez tenemos nuestra clase modelo, podemos crear nuevos objetos de esa clase, especímenes, instancias o datos de nuestra aplicación. Para ello instanciamos un objeto de esa clase que acabamos de crear.

```
var pelicula1 = new Pelicula();
```

Como en la mayoría de los lenguajes de programación orientados a objetos, utilizamos la palabra reservada "new" para crear una instancia de un objeto a partir de su clase. Así tenemos nuestra película, pelicula1, creada a partir de la clase Pelicula. Esta película ya es un especimen, es decir, un objeto concreto que hemos creado a partir de un prototipo. Ella podrá tener sus propios datos, como título, duración o cualquier otro dato que necesitemos que las películas alberguen.

> **Nota:** Fíjate que esta pelicula1 tiene la primera "p" en minúscula por ser un objeto, mientras que la clase Pelicula tiene la primera "P" mayúscula por ser una clase. Esto es por convención también de la POO.

Introducir y recuperar datos de un modelo: set() y get()

Este objeto película1 es como cualquier objeto Javascript. Yo si quisiera podría introducirle datos de la misma manera que hacemos con otros objetos en este lenguaje.

```
pelicula1.titulo = "Lo que el viento se llevó";
```

Luego podríamos hacer un console.log() para ver ese dato de mi objeto en la consola.

```
console.log(pelicula1.titulo);
```

Observaremos que en la consola aparece el título de esta película, sin embargo, esta manera de introducir datos no es del todo la adecuada en BackboneJS, porque, para que se produzcan los automatismos que Backbone nos ofrece con respecto a los modelos, necesita que nosotros hagamos las cosas de una manera un poco diferente.

Para definir los atributos o propiedades de los modelos utilizaremos el método set(), que recibe dos parámetros, el nombre del atributo que queremos cargar y el valor que queremos asignar.

```
pelicula1.set("titulo", "Alguien voló sobre el nido del cuco");
```

Así, hemos cargado de manera correcta el título de nuestra primera película, dejando a Backbone la responsabilidad de almacenarlo y hacer todo lo que necesite hacer sobre nuestro modelo.

La manera de recuperar un dato previamente almacenado con set() en un modelo es usar el método get(). A él le indicamos el nombre de la propiedad a la que queremos acceder.

```
pelicula1.get("titulo");
```

Eso nos devolverá el dato que habíamos almacenado anteriormente por medio de set(). Si observamos con detalle, será independiente del que cargamos simplemente asignando un valor a pelicula1.titulo. Esto lo veremos con un ejemplo detallado más adelante, pero si tomas todo el código de este artículo y lo pones en ejecución lo podrás comprobar por ti mismo.

```
// creo la clase Pelicula
var Pelicula = Backbone.Model.extend();
// creo un objeto de la clase Pelicula
var pelicula1 = new Pelicula();

// asigno una propiedad de manera habitual con Javascript
pelicula1.titulo = "Lo que el viento se llevó";
// asigno una propiedad ahora de la manera útil para aprovechar BackboneJS
pelicula1.set("titulo", "Alguien voló sobre el nido del cuco");

//muestro ambos datos y veo que son independientes
console.log(pelicula1.titulo);
console.log(pelicula1.get("titulo"));
```

Ventajas de usar set() y get()

Quizá quieras saber más sobre las ventajas de usar el método set() en vez de usar las propiedades de los objetos de Javascript nativo. Lo veremos en seguida, pero os adelantamos que al hacer un set() estaremos consiguiendo que BackboneJS ponga todos esos mecanismos que nos facilitarán la vida.

Entre otras cosas, provocando que se generen eventos específicos en el modelo, a los que podremos asociar funciones manejadoras para hacer cosas. También podremos detectar esos eventos desde cualquier otro objeto de la aplicación que esté escuchando.

Otra de las ventajas es que tendremos acceso a funciones de validación creadas en los modelos, o a serialización en JSON. Por ejemplo, mira este código:

```
pelicula1.toJSON();
```

Esto nos devolverá una serialización de los datos almacenados en el modelo, en notación JSON, que podremos usar para aquello que necesitemos, como almacenar los datos en LocalStorage, o enviarlos a un servidor por AJAX. Para el caso de la película creada anteriormente tendremos este JSON generado:

```
{titulo: "Alguien voló sobre el nido del cuco"}
```

Esperamos que lo visto hasta ahora te haya dado una primera idea, no solo sobre lo que te proporcionan los modelos, sino cómo se crean en Backbone.JS. Aclaramos que hemos sido muy escuetos en cuanto a código y que hay muchas otras utilidades y mecanismos que aún tenemos que ver. En próximos artículos continuaremos explorando estas posibilidades.

Artículo por Erick Ruiz de Chavez

Gestionar propiedades en los modelos BackboneJS

De qué manera podemos almacenar propiedades y métodos en los modelos de BackboneJS y qué diferencias tienen las distintas posibilidades.

En el artículo anterior estuvimos mostrando cómo crear nuestros primeros modelos y conociendo algunos de los primeros métodos disponibles en BackboneJS. En esa línea, seguiremos hablando de características de los modelos, viendo cuáles son las posibilidades de almacenamiento de propiedades y métodos.

Sin abandonar la sencillez de estas primeras tomas de contacto con el *framework*, con lo aprendido será suficiente para

comenzar a realizar ejemplos un poco más elaborados, que nos aportan mayores variantes para trabajar.

Especificar métodos y propiedades de instancia al crear la clase del modelo

Al crear un modelo con el método "extend()" podemos especificar una serie de propiedades, ya sean funciones o datos, que se copiarán en todas las instancias creadas sobre ese modelo.

> **Nota:** Como debemos saber si leímos el anterior artículo del Manual de BackboneJS, extend() te devuelve una clase (podríamos llamarla "clase modelo") a partir de la que podemos crear múltiples instancias de ese modelo.

Pues bien, extend() nos permite indicarle como parámetro un objeto Javascript y con ello conseguiremos que todas las instancias del modelo tengan las propiedades y métodos de ese objeto Javascript que le estamos pasando.

```
var ClaseModelo = Backbone.Model.extend({
dato: "Yeah!!",
metodo: function(){
alert("funciona!!");
}
});
```

Creada esa clase "MiModelo", todas las instancias que creemos a partir de ella, tendrán la propiedad "dato" y el método "metodo".

Esto quiere decir que podríamos instanciar un objeto y que sobre esa instancia podremos acceder a sus propiedades y métodos definidos al hacer el extend().

```
var objClaseModelo = new ClaseModelo();
alert(objClaseModelo.dato);
objClaseModelo.metodo();
```

Los datos asociados de esta forma a los objetos son como propiedades nativas de Javascript, de modo que podemos acceder por medio del operador punto ".", como a cualquiera de los métodos y atributos de objeto nativos en Javascript.

> **Nota:** Sin embargo, ten en cuenta que esas propiedades no estarán accesibles desde el método "get()" visto en el anterior artículo. Es decir, si hacemos objClaseModelo.get("dato") nos devolverá "undefined", ya que esa propiedad es "nativa" (pertenecía originalmente al objeto que se extendió para convertirlo en la clase del modelo). Como consecuencia de ésto, lo que hemos incluido de esta manera a los modelos no está dentro del circuito de atributos que Backbone monitoriza para generar eventos, o que usa cuando serializamos un modelo, por ejemplo.

Especificar métodos y propiedades de clase al crear la clase del modelo

Por medio del método extend() también podemos indicar propiedades y métodos de clase (llamados habitualmente en POO como propiedades o métodos estáticos), que se asociarán a la clase del modelo, en lugar de las instancias del modelo.

Para definir estos métodos y propiedades de clase tenemos que usar el segundo parámetro del método extend().

```
var ClaseModelo = Backbone.Model.extend({
propiedadInstancia: "esta no es la que nos interesa ahora"
}, {
propDeClase: "Prop estatica",
metodoDeClase: function(){
alert("este es un método estático o método de clase");
}
});
```

Los métodos y propiedades estáticos los podemos acceder e invocar a partir del nombre de la clase. Por ello, para acceder en este caso no necesito siquiera tener una instancia, simplemente lo hago a partir del nombre "ClaseModelo".

```
ClaseModelo.metodoDeClase();
console.log("propiedad de clase: " + ClaseModelo.propDeClase);
```

Especificar propiedades y métodos al crear el objeto de modelo

Hemos visto que BackboneJS nos permite crear propiedades y métodos para todas las instancias o la clase, al definir la clase del modelo. Sin embargo, al instanciar modelos también podemos definir propiedades y métodos, siendo éstos propios del ejemplar que se está instanciando.

Es más fácil de ver con un ejemplo.

```
// Creo una clase de modelo
var Modelo = Backbone.Model.extend({});
// instancio un objeto de ese modelo
var objModelo = new Modelo({
dato: "Esto se guardará dentro del modelo"
});
```

La diferencia en este caso, aparte de que este "dato" solo se está guardando en la instancia creada, es que estará dentro de Backbone como propiedad del modelo y no como propiedad nativa Javascript del objeto.

Siendo una propiedad del modelo controlada por Backbone.js deberemos acceder a través de las funciones "get()" o "set()" para recuperar su valor o cambiarlo, tal como se ve a continuación:

```
objModelo.get("dato");
objModelo.set("dato", "nuevo valor");
```

Al estar dentro de la arquitectura de BackboneJS, el framework estará al tanto de este dato, si es que tiene que serializar el modelo. Si se modifica, también estará al tanto para generar el correspondiente evento.

> **Nota:** Luego veremos más posibilidades de creación de propiedades en las instancias, en el artículo que dedicaremos a inicialización de los modelos.

Ejemplo que nos ayudará a diferenciar cómo Backbone aloja los datos en los modelos

Veamos ahora de nuevo esto que acabamos de aprender, pero con un ejemplo un poco más completo. Tenemos tres versiones de un código parecido:

1) En el primer caso hemos creado propiedades de instancia, asociadas al modelo. De este modo, todos los objetos de modelo que se creen tendrán estas propiedades que funcionarán como nativas Javascript.

```
var MiModelo = Backbone.Model.extend({
dato: "hola mundo MiModelo",
holaModelo: function(){
alert(this.dato);
}
});

var objMiModelo = new MiModelo();
objMiModelo.holaModelo();
```

Fíjate que dentro del método "holaModelo()" accedemos a la propiedad "dato" con "this.dato".

2) En el segundo caso hemos hecho propiedades de instancia, pero definidas al instanciar un modelo. Estas propiedades solo existirán en el objeto que acabamos de instanciar.

```
var MiModelo = Backbone.Model.extend({
holaModelo: function(){
alert(this.get("dato"));
}
});

var objMiModelo = new MiModelo({
dato: "Hola mundo Backbone"
});
objMiModelo.holaModelo();
```

Aquí podrás observar que para acceder al "dato" cargado al crear la instancia se tiene que utilizar el método get().

3) Ahora hemos hecho un tercer ejemplo, en el que ponemos junto el código del primer y segundo caso, de modo que nos demos cuenta que el "dato" creado en un lugar y otro son en realidad distintas variables, aunque se llamen igual.

```
var MiModelo = Backbone.Model.extend({
dato: "hola mundo MiModelo",
holaModelo: function(){
alert(this.dato);
alert(this.get("dato"));
}
});

var objMiModelo = new MiModelo({
dato: "Hola Al mundo Backbone!!"
});
objMiModelo.holaModelo();
```

Ahora fíjate que en el método holaModelo() estamos haciendo dos alert() y mostrando dos variables, tanto this.dato como tnis.get("dato") y que ambas variables tienen valores diferentes, lo que nos hace entender que son cosas distintas.

Espero que este ejemplo, aunque sencillo y nuevamente poco útil por sí mismo, nos haya hecho entender cómo podemos asociar datos a un modelo y qué diferencias existen si lo hacemos de una u otra manera.

Artículo por Erick Ruiz de Chavez