
Backbone.js

Otto Colomina <<otto@dccia.ua.es>>

Tabla de contenidos

1. Hola MVC en Javascript, Hola Backbone	5
1.1. MVC y las aplicaciones Javascript	5
MVC en la web (servidor)	5
MVC en la web (cliente)	6
1.2. <i>Frameworks</i> MVC en Javascript. Backbone	7
Características de Backbone	7
Estructura conceptual de una aplicación Backbone	7
1.3. Un ejemplo básico de aplicación Backbone : el <i>widget</i> del tiempo	8
El modelo	9
La vista	11
Eventos	12
1.4. Ejercicios de introducción a Backbone	14
Modificación del <i>widget</i> del tiempo (0.4)	14
UAdivino (0.6 puntos)	14
2. Modelos y colecciones	16
2.1. Modelos. Funcionalidades básicas	16
Atributos	16
Métodos y propiedades de un modelo	17
Inicializador y valores por defecto	17
Validación de datos	18
2.2. Persistencia con APIs REST	19
Create (POST)	19
READ (GET)	20
UPDATE (PUT)	21
DELETE (DELETE)	21
2.3. Colecciones	21
Navegar por las colecciones	22
Ordenación y filtrado	22
Manipulación básica	23
Persistencia con APIs REST	23
2.4. Eventos	23
Tratar con eventos desde objetos Javascript	24
Suscribirse/desuscribirse a eventos	24
Eventos para gestionar operaciones asíncronas	25
Emitir eventos de manera manual. Eventos propios	26
2.5. Configuración de la comunicación con el API REST	26
Configuración del identificador y/o la URL del modelo	26
Parseo "a medida" de la respuesta del servidor	27
Autenticación	28
APIs no REST. LocalStorage.	29
2.6. Ejercicios de modelos y colecciones	30
Star Wars API (0,4)	30
Comunicación con un API REST completo (0,6)	31
3. Vistas y templates. Routing	34
3.1. Vistas	34

La propiedad "el"	34
<i>Rendering</i>	35
Eventos	36
3.2. Vistas y modelos	37
Relación entre vista y modelo	37
Data binding	37
Data binding con eventos	37
Data binding automático	38
3.3. <i>Templates</i> (plantillas). El lenguaje de plantillas Mustache	39
Sintaxis básica	40
Plantillas en el lado del cliente	41
Uso típico de plantillas en Backbone	42
3.4. Routers	44
Routers básicos	44
Rutas con partes variables	45
Rutas por defecto	46
Navegación en el código	46
URLs completas	47
3.5. Ejercicios de vistas	48
Formulario para dar coches de alta (0,5 puntos)	48
Listado de coches (0,5 puntos)	48
4. Jerarquías de vistas	49
4.1. Listados dinámicos	49
Subvistas en Backbone	50
La vista global	50
Subvistas con Marionette	52
4.2. Composición genérica de vistas	54
Composición de secciones con Marionette	54
Secciones anidadas con Marionette	55
4.3. Ejercicios de jerarquías de vistas	57
Vistas y subvistas con Backbone (0,5 puntos)	57
Vistas y subvistas con Marionette (0,5 puntos)	57
5. Miniproyecto de aplicación con Backbone y Marionette	58
5.1. Requerimientos	58
5.2. Implementación de los requerimientos "iniciales"	58
La "lógica de negocio"	58
Estructura de la interfaz	59
Vista de un solo comic: <code>js/views/VistaComic.js</code>	59
Vista de lista de comics	60
Vista Global	60
Vista de búsqueda (0,5 puntos)	61
De nuevo a la vista global (0,25)	62
Ver detalles de comic (0,25)	62
Cerrar la vista de detalles y volver al listado de comics (0,25 puntos)	64
5.3. Requerimientos "adicionales" (1 punto en total)	64
6. Interfaces web con ReactJS	66
6.1. ¿Por qué ReactJS?	66
6.2. ¡Hola React!. Nuestros primeros componentes	66
Componentes personalizados. Clases.	68
6.3. Componentes sin estado. Propiedades	68
6.4. Componentes con estado	71
6.5. Interactividad. Eventos	73
6.6. JSX	75

De JSX a JS	76
"Gotchas" de JSX	77
El operador "spread"	78
6.7. Manejo de formularios	79
Componentes controlados	79
refs	80
6.8. Ejercicios de React (I)	81
El retorno del Widget del tiempo (0,4)	81
Componente tabla de datos	81
7. Interfaces web con ReactJS (II)	83
7.1. Composición de componentes	83
Dónde colocar el estado	84
Comunicación de abajo a arriba	85
7.2. Ciclo de vida de un componente	87
Ejemplo: carga de datos con AJAX	88
Mejora del rendimiento con <code>shouldComponentUpdate</code>	90
7.3. React y Backbone	90
Un componente con un modelo asociado	90
Un componente con una colección asociada	91
7.4. Ejercicios de React (II)	93
Composición de componentes y ciclo de vida (0,5 puntos)	93
Comunicación en la jerarquía de componentes (0,75 puntos)	93
8. Introducción a la arquitectura Flux para aplicaciones React	94
8.1. ¿ Por qué Flux?	94
8.2. Flux a grandes rasgos	95
Stores	96
Actions	96
Dispatcher	96
Flujo de datos	96
8.3. Un ejemplo sencillo	97
Librerías para implementar Flux	97
El <i>Dispatcher</i>	97
Las acciones	98
Los <i>stores</i>	99
Los componentes React	101
Coordinación entre <i>stores</i>	103
8.4. Ejercicio de Flux (1,25 puntos en total)	105
Acciones (0,4 puntos)	105
Stores (0,4 puntos)	105
Componente React (0,45 puntos)	105
9. Apéndice: Herramientas para gestionar el flujo de trabajo en el desarrollo <i>frontend</i>	106
9.1. Gestión de paquetes con <code>Bower</code>	107
9.2. Creación de plantillas con <code>Yeoman</code>	108
9.3. <code>npm</code> + <i>bundler</i>	108
Uso de <code>browserify</code>	109
<code>browserify</code> y <code>babel</code>	110
10. Apéndice: <i>testing</i> con Backbone	112
10.1. Introducción a Jasmine	112
Suites y casos de prueba	112
Expectativas y <i>matchers</i>	113
Configuración de cada prueba	113
Ejecutar las pruebas	113
10.2. Pruebas con Jasmine en Backbone	114

Pruebas de lógica de negocio	114
Pruebas sobre HTML	114
Uso de "espías"	115
Pruebas con AJAX	116

1. Hola MVC en Javascript, Hola Backbone

En esta primera sesión vamos a hacer una breve introducción al patrón MVC (Modelo/Vista/Controlador) y cómo en los últimos años se ha desplazado del servidor hacia el cliente. También veremos los conceptos básicos de Backbone e implementaremos una pequeña aplicación, que muestre cómo se aplican estos conceptos en la práctica. En el resto de sesiones de la asignatura iremos profundizando en las distintas funcionalidades de Backbone.

1.1. MVC y las aplicaciones Javascript

El patrón **MVC** o **Modelo/Vista/Controlador** es uno de los patrones de diseño arquitectónicos más conocidos y usados en la actualidad. La idea básica consiste en que deseamos separar el **modelo**, esto es, los datos de nuestra aplicación, de la **vista**, es decir, de su presentación en la interfaz de usuario. Como veremos esta idea básica admite multitud de variantes, motivo por el cual en esta definición básica no hemos introducido al **controlador**. Según la variante de MVC cambia el papel exacto que debe desempeñar el controlador, o cómo se pueden comunicar entre sí los tres componentes.

MVC en la web (servidor)

MVC es un patrón omnipresente en el lado del servidor. Existe en todas las plataformas web: JSF, Struts o Spring MVC en JavaEE, ASP.NET MVC en .NET, CakePHP, Symphony, Codeigniter y otros en PHP, Rails en Ruby, Django en Python,...

A finales de los 90, Sun propugnó lo que dio en llamar "modelo 2" como patrón básico de arquitectura para aplicaciones web basadas en servlets y JSPs. Con mayores o menores modificaciones, este modelo fue la base de Struts y otros *frameworks* MVC del mundo JavaEE como Spring MVC. En el *modelo 2*, las peticiones del cliente las recibe un *servlet*, que hace el papel de **controlador**, y que delega la lógica de negocio en un conjunto de JavaBeans (el **modelo**). Finalmente el control se transfiere a un JSP (la **vista**), que muestra los resultados al usuario. No obstante, Sun nunca llegó a estandarizar un API o un *framework* para implementar MVC en nuestras aplicaciones.

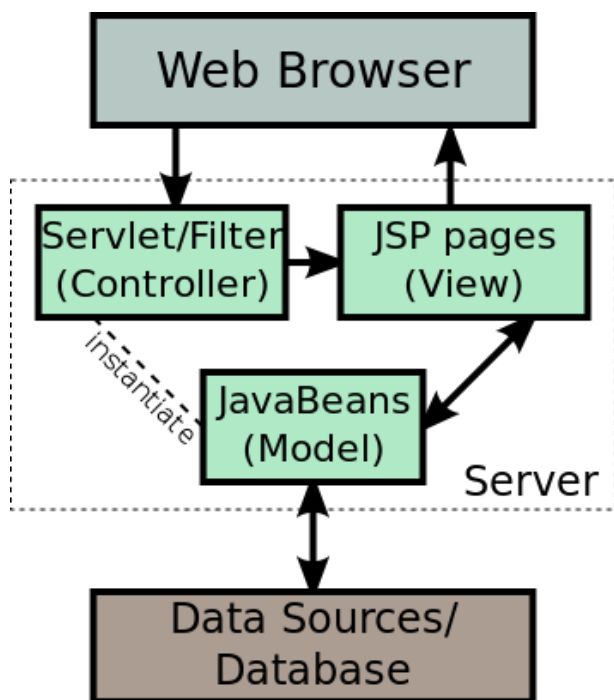


Figura 1. Modelo 2 de aplicaciones JSP. (By Libertyernie2 - Own work. CC BY-SA 3.0 via Wikimedia Commons)



En realidad, JSF podría considerarse el *framework* MVC estándar de JavaEE, pero su filosofía es muy distinta a la de otros como Struts o Spring MVC. JSF está orientado a *componentes*, mientras que los otros están orientados a *acciones* (aquí podéis ver [una comparación](#)¹). Además MVC en realidad es solo una pequeña parte de JSF, siendo su parte más importante todo el tema de componentes de usuario. Por ello se está en proceso de elaboración de un JSR para elaborar un [estándar de MVC en JavaEE](#)².

En la actualidad, en un mundo de aplicaciones web convertidas en "simples" APIs en el lado del servidor, la antigua preponderancia de MVC en el servidor parece haberse difuminado un poco. Lo que es lógico, ya que la vista se ha trasladado al cliente. Así, la "necesidad" de usar MVC para estructurar la aplicación ha acabado trasladándose también al lado del cliente.

MVC en la web (cliente)

Cuando el uso de Javascript se limitaba a cosas como validación de formularios, pequeños cálculos y algunos efectos visuales MVC en el cliente no hacía una gran falta. El interfaz ya venía construido desde el servidor en forma de plantillas, y asimismo el servidor ya generaba casi todos los datos que se le mostraban al usuario. Pero en la actualidad se tiende a ir hacia SPAs (Single Page Applications), en las que básicamente **la interfaz se construye dinámicamente con Javascript**, lo que incluye también el formateo y presentación de los datos que está viendo el usuario, y la gestión de los nuevos que crea. Esto implica que desde Javascript también debemos poder manipular el modelo y ejecutar lógica de negocio. El servidor se queda como una especie de fuente de datos remota. Como vemos, prácticamente todo el esquema del antiguo "modelo 2" se ha trasladado al cliente.

¹ <http://www.oracle.com/technetwork/articles/java/mvc-2280472.html>

² <https://jcp.org/en/jsr/detail?id=371>

1.2. Frameworks MVC en Javascript. Backbone

En una época en la que parece que hay que usar un *framework* para todo, el interés de llevar MVC al cliente provocó la aparición de multitud de *frameworks* MVC para Javascript. Surgieron tantas alternativas diferentes que en una cierta época era realmente difícil poder [decidirse por uno de ellos](#)³. En [TodoMVC](#)⁴ se usa una idea interesante que es escribir una aplicación de referencia (la típica lista de tareas) en cada uno de los *frameworks* para que el código hable por sí mismo.

Características de Backbone

Backbone fue uno de los primeros *frameworks* MVC en hacerse popular. Su filosofía va en la línea de lo que los anglosajones llaman "**non opinionated**", es decir, un *framework* que da libertad al desarrollador para hacer las cosas con su propio estilo, y no impone cierta forma de trabajar. Esto si lo queremos ver desde el punto de vista negativo, hace que el proceso de aprendizaje esté mezclado con cierta inseguridad para el desarrollador, ya que nunca acaba de tener claro "si lo está haciendo bien".

Otro aspecto que define el carácter de Backbone es la **simplicidad**. Es pequeño en términos de número de líneas de código y por tanto las funcionalidades que ofrece "tal cual" son limitadas. No ofrece facilidades "automágicas", casi todo está bajo control del desarrollador. Esto ha hecho que surjan multitud de *plugins* para cubrir las funcionalidades que Backbone no tiene y sí tienen otros *frameworks* más complejos.

Estructura conceptual de una aplicación Backbone

Cuando se habla de MVC, siempre surge la duda de exactamente de qué tipo de MVC se está hablando. Desde que apareció la [versión original](#)⁵ del patrón en los 70, en el contexto de aplicaciones de escritorio desarrolladas en Smalltalk, han surgido [multitud de variantes](#)⁶, en las que cambian los *roles* que desarrolla exactamente cada uno de los componentes "Modelo/Vista/Controlador" o el flujo de información que hay entre ellos. Incluso hay versiones en las que alguno de los componentes del trío original desaparece y es sustituido por otros, como MVP (Model/View/Presenter), MVVM (Model/View/ViewModel),...

Backbone no es exactamente MVC, más que nada porque directamente carece de controladores. En cuanto a [qué es entonces, exactamente](#)⁷ no lo vamos a responder aquí, más que nada porque es una [discusión probablemente infructuosa](#)⁸ y que en cualquier caso no va a ayudar a comprender mejor su funcionamiento. Nos conformaremos con llamarlo MV*, como dicen los anglosajones, MV- *whatever*, o MV- *loquesea*.

Teniendo presente lo que acabamos de decir, la siguiente figura mostraría una **posible estructura** conceptual de una aplicación Backbone (posible porque también podríamos estructurar las cosas de otro modo y tampoco "estaría mal"). La figura está tomada del libro de [Addy Osmani](#)⁹ "[Developing Backbone Applications](#)"¹⁰, que además está [disponible en Github](#)¹¹

³ <http://www.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/>

⁴ <http://todomvc.com/>

⁵ <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#History>

⁶ <http://kasparov.skife.org/blog/src/java/mvc.html>

⁷ <http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/>

⁸ <http://stackoverflow.com/questions/10745782/is-backbone-js-really-an-mvc>

⁹ <https://twitter.com/addyosmani>

¹⁰ <http://shop.oreilly.com/product/0636920025344.do>

¹¹ <http://addyosmani.github.io/backbone-fundamentals/>

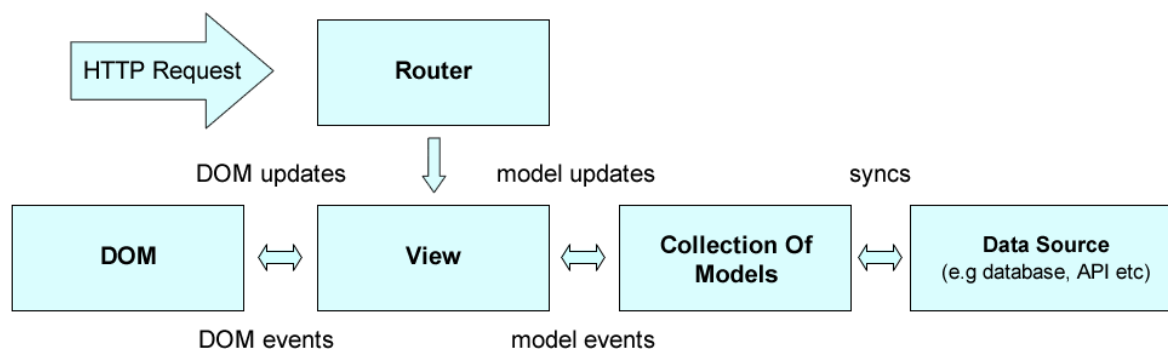


Figura 2. Típico flujo de información en una aplicación Backbone

Como vemos en la figura, en el "corazón" de Backbone están los **modelos** y las **vistas**. Además, para reflejar el hecho de que típicamente en una aplicación vamos a manejar más de una instancia del mismo tipo de modelo (clientes, libros, mensajes,...) Backbone modeliza también la idea de **colección** de modelos.

Las colecciones de modelos interactúan (se sincronizan, *sync*) con una fuente de datos, típicamente un API REST en el servidor.

Por otro lado, la vista interactúa con el HTML de la página. La vista actualiza el DOM, modificando así el HTML "en tiempo real" y en el sentido contrario, los eventos del DOM se procesan en la vista.



Hablar de DOM (o árbol HTML) y de vista como elementos separados nos puede dar la idea de que **el concepto de vista** no es igual en Backbone que en otros muchos *frameworks* MVC, en los que la vista es precisamente la interfaz, que en nuestro caso sería el HTML. Lo veremos con más profundidad en la sesión 3.

Vista y modelo se comunican mediante **eventos**. Esto se hace así para reducir el acoplamiento entre ambos. Es normal que la vista tenga que conocer ciertos detalles del modelo para poder interactuar con él, pero en general el modelo no debería tener que conocer cómo es la vista para comunicarse con ella. Así podremos reutilizar los modelos cambiando una vista por otra. Para solucionar este problema se usa el paradigma de comunicación "Publicar/Suscribir" (*Publish/Subscribe* o *Pub/Sub*). En este paradigma el objeto que quiere comunicarse con otro sin acoplarse con él no lo hace directamente sino *emitiendo eventos*. El objeto interesado se suscribe a esos eventos. En Backbone veremos que la vista se suscribe a los eventos que le interesan del modelo.

Para terminar, el **router** es un componente que asocia URLs con código Javascript. La idea es que cada operación o cada estado de nuestra aplicación debería identificarse con una URL, lo que permitiría que el usuario creara sin problemas sus *bookmarks*.

1.3. Un ejemplo básico de aplicación Backbone : el *widget* del tiempo

En lugar de seguir discutiendo de manera abstracta sobre las funcionalidades, vamos a introducir los aspectos básicos del desarrollo en Backbone con un ejemplo sencillo.

Queremos implementar un *widget* donde se pueda consultar el tiempo que hace en una determinada localidad. Algo al estilo de lo que se muestra en la siguiente imagen:



Figura 3. El aspecto del *widget* terminado

Como vemos, simplemente hay un campo de texto para teclear la localidad y al pulsar el botón aparecen los datos meteorológicos. Para obtener los datos reales usaremos un [servicio externo](#)¹².

El modelo

Como ya hemos visto, el modelo es el *conjunto de objetos que forman el dominio de nuestra aplicación* y por tanto depende enteramente de su naturaleza: en un "campus virtual" tendremos profesores, alumnos, notas, ... mientras que en una red social tendremos usuarios, mensajes, fotos, ...

Los modelos son realmente las mismas entidades que usamos en la capa de negocio de la aplicación. En ellos encapsulamos por tanto dos aspectos: los *datos* y la *lógica de negocio*. En nuestro ejemplo del tiempo los datos serán los parámetros que definen el estado actual del tiempo (temperatura, humedad, descripción en modo texto: - "soleado", "nublado", ...-). La lógica de negocio se ocuparía de la comunicación con el servicio web remoto que nos ofrece los datos.

Backbone nos ofrece una "clase" base, `Backbone.Model`, que podemos extender para crear nuestros propios modelos. No es necesario especificar por adelantado las propiedades del modelo, se pueden crear en cualquier momento, igual que con los objetos Javascript convencionales

```
var DatosTiempo = Backbone.Model.extend(); ❶
var miTiempo = new DatosTiempo({"localidad":"Alicante"}); ❷
console.log(miTiempo.get("localidad")); ❸
miTiempo.set("localidad", "San Vicente del Raspeig");
```

- ❶ Creamos la clase para representar nuestro modelo.
- ❷ Creamos una instancia de dicha clase, y le asignamos una propiedad "localidad" con valor "Alicante".
- ❸ Como vemos, la clase `Model` nos proporciona *getters* y *setters*.

Antes de ver cómo implementamos la lógica de negocio, necesitamos saber cómo funciona el API del servicio web. Básicamente hay que hacer una petición GET a <http://api.openweathermap.org/data/2.5/weather> con el parámetro `q` igual al nombre de la localidad y el valor de nuestra API KEY como valor del parámetro `APPID` (podemos obtenerla [registrándonos en OpenWeatherMap](#)¹³). Podemos hacer pruebas provisionales con `APPID=1adb13e22f23c3de1ca37f3be90763a9`.

Si además añadimos los parámetros `units=metric&lang=es` obtendremos el resultado en español usando unidades del sistema métrico. La respuesta será un JSON del estilo

¹² <http://openweathermap.org>

¹³ https://home.openweathermap.org/users/sign_up

```

"coord": {
  "lon": -0.48,
  "lat": 38.35
},
"sys": {
  "message": 0.1941,
  "country": "ES",
  "sunrise": 1423292456,
  "sunset": 1423330269
},
"weather": [
  {
    "id": 800,
    "main": "Clear",
    "description": "cielo claro",
    "icon": "01n"
  }
],
...

```

Como vemos, la descripción del tiempo está en el campo `weather[0].description`. El `weather[0].icon` es el icono que lo representa gráficamente. Como indica la [documentación](#)¹⁴, para obtener el icono hay que ponerle delante a este nombre una URL base.

Con esto ya podemos implementar la llamada al servicio web desde nuestro modelo. La lógica de negocio la implementaremos normalmente con propiedades de tipo `function()`. Como la funcionalidad la deben tener todas las instancias de la clase, le asignaremos la propiedad a la clase:

```

var URL_API = "http://api.openweathermap.org/data/2.5/weather?
APPID=1adb13e22f23c3de1ca37f3be90763a9&units=metric&lang=es";
var URL_BASE_ICONO = "http://openweathermap.org/img/w/"

var DatosTiempo = Backbone.Model.extend({
  actualizarTiempo: function () { ❶
    var callback = function (data) { ❷
      this.set('descripcion', data.weather[0].description);
      var icono_url = URL_BASE_ICONO + data.weather[0].icon
+ ".png";
      this.set('icono_url', icono_url);
      this.set('dt', data.dt);
      console.log("Se ha leído el tiempo del servicio web");
    }
    $.getJSON( ❸
      URL_API,
      {q: this.get('localidad')}, ❹
      callback.bind(this) ❺
    );
  }
});
var miTiempo = new DatosTiempo();

```

¹⁴ <http://openweathermap.org/weather-conditions>

- ❶ Como vemos, la propiedad es una función, así que luego haremos `miTiempo.actualizarTiempo()` cuando queramos disparar la actualización
- ❷ Creamos un *callback* para la petición AJAX, que recibirá el JSON ya parseado. Aquí es donde rellenamos los datos del modelo con los recibidos del servicio web, la `descripcion` del tiempo, la `icono_url` que la representa gráficamente, y un atributo `dt` que es un *timestamp* indicando cuándo se han obtenido los datos. Así, si el *timestamp* no cambia no va a ser necesario refrescar el HTML.
- ❸ Usamos jQuery para hacer más compacto el código.
- ❹ Pasamos el parámetro `q=` nombre de la localidad buscada.
- ❺ Y aquí es donde viene el truco necesario para que el código funcione. En el *callback* usamos `this` para referirnos al modelo. Sin embargo si usamos jQuery, en el *callback* `this` va a ser el objeto jQuery usado para hacer la petición. Con `bind` forzamos a que `this` sea lo que necesitamos.

Podemos probar el funcionamiento del código anterior tecleando algo como lo que sigue en la consola Javascript:

```
var miTiempo = new DatosTiempo();
miTiempo.set('localidad', 'Alicante');
miTiempo.actualizarTiempo();
//Hay que dar tiempo a que la petición AJAX acabe antes de teclear esto
console.log(miTiempo.get('descripcion'));
```



Mucho cuidado con el código anterior: `miTiempo.actualizarTiempo()` dispara una petición AJAX **asíncrona**, con lo que después de ejecutar esta línea tendríamos que esperar a que aparezca el mensaje `Se ha leído el tiempo del servicio web` que se imprime al final del *callback* para asegurarnos de que se ha procesado ya la respuesta. Luego veremos cómo se arregla esto en la versión completa.

La vista

La vista en Backbone tiene la misión de generar el HTML que represente el modelo en pantalla. Es decir, de dibujar el *widget*. También debe responder a los eventos del usuario. En nuestro caso el único evento es la pulsación en el botón "ver tiempo".

Las vistas heredan de la clase `Backbone.View`, y deben tener asociada una instancia de un modelo (también podrían tener varias instancias, como veremos en la siguiente sesión).

```
var TiempoWidget = Backbone.View.extend({
  render: function() { ❶
    this.$el.html('<input type="text" id="localidad">' +
      '<input type="button" value="Ver tiempo" id="ver_tiempo">' +
      '<div> <img id="icono" src=""></div>' +
      '<div id="descripcion"></div>');
  },
  renderData: function() { ❷
    $('#icono').attr('src', this.model.get("icono_url"));
    $('#descripcion').html(this.model.get("descripcion"));
  },
  events: { ❸
    "click #ver_tiempo": "ver_tiempo_de"
```

```

    },
    ver_tiempo_de: function() { ❷
      this.model.set("localidad", $("#localidad").val());
      this.model.actualizarTiempo();
      this.renderData();
    }
  })

  var miTiempo = new DatosTiempo();
  var miWidget = new TiempoWidget({model: miTiempo}); ❸
  miWidget.render(); ❹
  $('#tiempo_widget').html(miWidget.$el) ❺

```



Esta versión de la vista no va a funcionar correctamente por el motivo que se discutirá a continuación. ¡No hagáis esto tal cual en casa!

- ❶ Esta función se encarga de generar el HTML de la vista. `this.$el` es un nodo de jQuery que representa la "raíz" de la vista. Manipulando su HTML estamos manipulando el HTML de la vista.
- ❷ Esta función se encarga de actualizar únicamente el icono del tiempo y la descripción textual. La vista solo hará falta dibujarla completa la primera vez, las siguientes bastará con esto.
- ❸ Esta propiedad se encarga de vincular los eventos producidos sobre la vista con manejadores de evento. La propiedad debe llamarse `events` y es un conjunto de pares `clave:valor` donde la *clave* es un nombre de evento + selector CSS y el valor el nombre de la función a asociar.
- ❹ Tal y como se ha definido `events`, esta sería la función que se dispararía al hacer *clic* sobre el botón, que tiene el id `ver_tiempo`. Aquí tomamos la `localidad`, que estará escrita en el campo de texto con id `localidad`, llamamos al `actualizarTiempo` del modelo y luego a `renderData` para actualizar gráficamente la información del tiempo. Pero **en realidad esto no va a funcionar** ya que al ser `actualizarTiempo` asíncrono deberíamos esperar a que terminara para llamar a `renderData()`. Ahora veremos cómo resolverlo.
- ❺ Creamos una instancia de la vista y le asociamos una instancia del modelo.
- ❻ Llamamos al `render` de la vista para generar su HTML, pero este todavía no está en la página, solo en la propiedad `$el` de la vista.
- ❼ Finalmente incluimos el HTML de la vista en la página usando el API de jQuery

Eventos

Como ya hemos dicho, tenemos un pequeño problema: ¿cómo hacemos que el modelo avise a la vista de que `actualizarTiempo` ya ha acabado y que por tanto se puede ejecutar el `renderData()`? podría ejecutarlo el propio modelo, pero necesitaría mantener una referencia a la vista y esto haría que dejara de ser genérico y se "atará" a la vista (asumiera que siempre va a estar asociado a una vista que tiene un método `renderData`).

La solución más limpia para comunicar del modelo hacia la vista es no tocar el código del modelo en absoluto y usar la idea de "Publicar/Suscribir". Por defecto, los modelos de Backbone emiten eventos cuando se dan ciertas situaciones, por ejemplo que cambia una propiedad, o que el modelo se sincroniza con el estado del servidor. Lo único que debe hacer la vista es encontrar el evento apropiado y suscribirse a él. En este caso, el evento que nos viene

que ni pintado sería que la propiedad `dt` del modelo adquiriera un nuevo valor. Recordemos que esta propiedad es un *timestamp* que nos indica cuándo se han obtenido los datos.

En el `initialize` de la vista, que se usa para inicializar valores por defecto y otros elementos, podemos suscribirnos al evento del modelo. Esto se puede hacer con el método `listenTo`, indicando a qué objeto queremos suscribirnos, qué evento nos interesa, y cuál va a ser el manejador de evento:

```
.....  
var TiempoWidget = Backbone.View.extend({  
  initialize: function() {  
    this.listenTo(this.model, 'change:dt', this.renderData)  
  },  
  ...  
  ver_tiempo_de: function() {  
    this.model.set("localidad", $("#localidad").val())  
    this.model.actualizarTiempo()  
  }  
}
```

```
.....
```

El resto del código de la vista quedaría igual que antes. Como vemos, la función que dispara la actualización del tiempo no necesita llamar a `renderData` ella misma. Si la operación de actualización cambia el atributo `dt` del modelo se llamará a `renderData` automáticamente.

1.4. Ejercicios de introducción a Backbone



Como norma general de la asignatura, para cada ejercicio crearemos una carpeta con su nombre e incluiremos en ella todo lo necesario: el HTML, el JS propio, las librerías JS usadas (jQuery, Backbone, ...). Aunque repitamos los archivos, así lo tenemos todo de manera independiente. En las plantillas de la asignatura tenéis una plantilla genérica de aplicación con Backbone, `plantilla_backbone`, podéis usarla como base para los ejercicios.

Modificación del *widget* del tiempo (0.4)

Este ejercicio debes entregarlo en una carpeta llamada `mi_tiempo_backbone`.

En este ejercicio vamos a modificar el modelo del *widget* del tiempo para incluir también la temperatura actual, y crearemos una nueva vista que incluya esta información.

Modificación del modelo

Modificar la clase del modelo `DatosTiempo` para que cuando se reciba la respuesta del servidor se incluya también la temperatura actual, en una nueva propiedad `temp`. Este dato está en el campo `main.temp` del JSON recibido del servidor.

Comprobad, usando la consola Javascript, que la temperatura se almacena correctamente en el modelo, llamando manualmente a `actualizarTiempo` y luego mostrando la propiedad `temp`.

Creación de una nueva vista

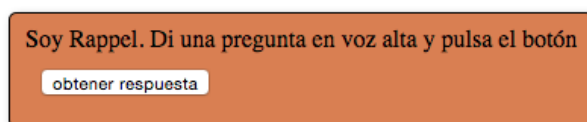
Crear un nuevo tipo de vista `TemperaturaWidget` similar a `TiempoWidget` pero que únicamente mostrará la temperatura actual. Insertarlo en el HTML y comprobar que funciona.

UAdivino (0.6 puntos)

Este ejercicio debes entregarlo en una carpeta llamada `UAdivino`.

Crear una aplicación de Backbone que funcione al estilo de la conocida "bola 8 mágica", a la que se le "hace una pregunta" en voz alta y nos responde algo al azar.

El *widget* tendrá un aspecto similar al siguiente:



- El modelo

- # Tendrá una propiedad llamada "nombre", con el nombre del adivino (Rappel, Zoltar, ...)
- # Tendrá un único método de lógica de negocio llamado `obtenerRespuesta()`, que devolverá una respuesta al azar de entre las predefinidas.



Podéis guardar las respuestas predefinidas en un array dentro del objeto `defaults`, que en Backbone se usa para guardar valores por defecto

```
var Adivino = Backbone.Model.extend({
  defaults: {
    respuestas: ["Sí", "No", "Ni de coña", "¡Claro que sí!"]
  },
  //Resto del modelo...
  ...
});
```

- La vista

- # Inicialmente muestra el nombre del adivino y un botón para obtener respuesta
- # Podéis mostrar la respuesta con un `alert` para simplificar, o bien insertarla en el HTML del *widget*.

2. Modelos y colecciones

2.1. Modelos. Funcionalidades básicas

Un modelo en nuestra aplicación no es más que una clase propia que hereda de la clase `Backbone.Model`. Para la herencia se usa el método `extend`. Una vez creada la clase del modelo podemos crear instancias del mismo con `new`, como es habitual en Javascript.

```
var Usuario = Backbone.Model.extend({})
var u1 = new Usuario() //un usuario
var u2 = new Usuario() //otro
```



Ya sabemos que en Javascript (o en ECMAScript hasta la versión 5 inclusive, para hablar con algo más de propiedad) no existen las clases como tales, ni tampoco la herencia al estilo Java o C++, sino los objetos y la herencia basada en prototipos. No obstante Backbone al igual que muchos otros *frameworks* "imita" el enfoque clásico de la POO basada en clases, instancias y herencia entre clases. Aunque no sea totalmente correcto hablar de la "clase Usuario" a partir de ahora vamos a usar esta terminología para simplificar. Si queréis más información sobre cómo se implementan las clases y la herencia en Backbone podéis consultar este [tutorial](#)¹⁵ o directamente el propio código fuente anotado de Backbone, en el [apartado "Helpers"](#)¹⁶.

El método `extend` admite como parámetro un objeto en el que podemos encapsular diversas propiedades del modelo, más tarde veremos su uso. En el ejemplo hemos usado un objeto vacío. (`{}`).

Atributos

Los modelos en Backbone siguen la filosofía de Javascript: son dinámicos y podemos añadir y eliminar atributos sobre la marcha. Para añadir un atributo, o cambiar su valor si este ya existe, usamos `set(nombre, valor)`. Para obtener el valor, `get(nombre)`. Continuando con el ejemplo anterior:

```
var Usuario = Backbone.Model.extend({})
var u1 = new Usuario()
u1.set("nombre", "Pepe")
u1.set("fecha_nac", new Date(1990, 0, 1)) //1 de enero de 1990
```

También podemos fijar los valores de los atributos al instanciar el objeto con `new`. Se los pasamos a este método en forma de *hash*:

```
u1 = new Usuario({nombre: "Pepe", fecha_nac: new Date(1990, 0, 1)})
```

Si deseamos eliminar un atributo podemos usar `unset(nombre)`, aunque este método lo único que hace es borrar el atributo usando `delete`. Podríamos hacer lo mismo accediendo

¹⁵ <http://dailyjs.com/2012/08/09/mvstar-5/>

¹⁶ <http://backbonejs.org/docs/backbone.html#helpers>

directamente a la propiedad de la clase llamada `attributes`, que es la que contiene los atributos en sí

```
u1.unset("fecha_nac") //es lo mismo que delete u1.attributes("fecha_nac")
```



aunque podemos acceder a los atributos directamente modificando `attributes`, se recomienda hacerlo siempre a través de `get/set`.

Podemos comprobar si un objeto tiene un determinado atributo con `has(nombre)`, que devolverá un valor booleano indicándolo.

Métodos y propiedades de un modelo

Como ya hemos dicho, cuando creamos una clase que hereda de `Backbone.Model` podemos definir propiedades en forma de objeto Javascript, normalmente usando notación literal. De hecho podemos definir propiedades de instancia y propiedades de clase. Las primeras serían propias de cada instancia de nuestro modelo. Las segundas serían de la clase del modelo en sí. En Backbone ya vienen definidas por defecto unas cuantas propiedades de instancia. Por ejemplo cada objeto tiene un `cid` que es un identificador único y se va generando secuencialmente.



podríamos usar las propiedades especificadas en `extend` para definir variables miembro de nuestros objetos, pero lo habitual es usar atributos para esta tarea.

Lo habitual es **usar las propiedades especificadas en el `extend` para definir métodos**. Un método no va a ser más que una propiedad que resulta ser una función. Por ejemplo:

```
var Usuario = Backbone.Model.extend({
  toString: function() {
    return this.get("nombre") + ".Nacido/a el "
      + this.get("fecha_nac").toLocaleDateString()
  }
})
var u1 = new Usuario({nombre:"Pepe", fecha_nac: new Date(1990, 0, 1)})
console.log(u1.toString()) //Pepe. Nacido/a el 1/1/1990
```

Todos los modelos tienen una propiedad por defecto `cid` (*client id*) que actúa como identificador y cuyo valor va generando automáticamente Backbone. Como luego veremos, cuando el modelo se almacena en el servidor también pasa a tener una propiedad `id`, con valor asignado por este.

Inicializador y valores por defecto

Podemos ejecutar un determinado código cuando se cree el modelo, poniéndolo en el método `initialize`

```
var Usuario = Backbone.Model.extend({
  initialize: function() {
    console.log("Inicializando usuario...")
  }
})
```

```

    //como fecha de alta del usuario ponemos la actual
    this.set("fecha_alta", new Date())
  }
})
var u1 = new Usuario() //Imprime: inicializando usuario...
console.log(u1.get("fecha_alta")) //imprime la fecha actual

```

Aunque si lo que queremos es simplemente inicializar atributos con valores por defecto es más directo usar la propiedad `defaults`. A esta propiedad se le pasa un objeto en notación literal con los nombres de los atributos y sus valores por defecto:

```

var Usuario = Backbone.Model.extend({
  defaults: {
    'saldo': 0
  }
})
u1 = new Usuario()
console.log(u1.get("saldo")) //0

```



Recordemos que los objetos en Javascript se pasan por referencia, de modo que si usamos un objeto como valor por defecto todas las instancias referenciarán el mismo objeto. Y además modificar el contenido del atributo en una instancia lo modificará en todas, por ejemplo:

```

var Usuario = Backbone.Model.extend({
  defaults: {
    'fecha_alta': new Date()
  }
})
var u1 = new Usuario();
var u2 = new Usuario();
console.log(u1.get("fecha_alta")==u2.get("fecha_alta")) //true
u1.get("fecha_alta").setFullYear(2000)
console.log(u2.get("fecha_alta").getFullYear()) //2000!!

```

La solución es hacer que `defaults` sea una función que devuelva un objeto con los valores deseados, así, cada instancia tendrá su propia copia de valores por defecto.

```

var Usuario = Backbone.Model.extend({
  defaults: function() {
    return {'fecha_alta': new Date()}
  }
})
var u1 = new Usuario();
var u2 = new Usuario();
console.log(u1.get("fecha_alta")==u2.get("fecha_alta")) //false

```

Validación de datos

Backbone ofrece un método `validate()` para la validación de datos, pero el código tenemos que escribirlo nosotros por completo, no existe ningún tipo de validación declarativa.

Si la validación es correcta el método `validate()` no debería devolver nada. En caso de que sea incorrecta, corre por cuenta del desarrollador qué devolver, mientras se devuelva algo. Por ejemplo:

```
var Usuario = Backbone.Model.extend({
  validate: function (attrs) {
    var password = attrs.password;
    if (!password || password.length<6)
      return "Password no válido";
  }
});
```

`validate()` recibe como parámetro un objeto con los atributos que se están validando. Backbone llama automáticamente a `validate()` al guardar un objeto en el servidor. En ese caso los atributos recibidos en `validate()` son los actuales del objeto.

Además validará el cambio de valor de un atributo si pasamos la opción `validate:true`. En este caso los atributos recibidos en `validate()` son los nuevos valores que estamos intentando fijar.

```
//Continúa el código del ejemplo anterior
var unUsuario = new Usuario();
unUsuario.set({"password":""}, {validate:true});
//la propiedad "validationError" nos da el último valor devuelto por
"validate"
console.log(unUsuario.validationError) //"Password no válido"
```

2.2. Persistencia con APIs REST

Con Backbone podemos sincronizar de forma sencilla el estado local de un modelo con el estado en el servidor. El *framework* está preparado por defecto para comunicarse con el servidor empleando las convenciones REST habituales. Partiendo de la URL que referencia el modelo en el servidor, Backbone va a generar por nosotros las llamadas AJAX necesarias para hacer CRUD del modelo, ahorrándonos tener que escribir nosotros mismos el código.

Con la propiedad `urlRoot` fijamos la URL "base" del modelo. Es decir, será la URL de la "colección" en la que está incluido en el servidor. Por ejemplo un usuario en el servidor podría estar en una URL del tipo <http://miapp.com/api/usuarios/identificador>. Por tanto la URL base será solamente <http://miapp.com/api/usuarios/>

```
var Usuario = Backbone.Model.extend({
  urlRoot: 'miapp.com/api/usuarios/'
});
```

Una vez establecida la propiedad `urlRoot` podemos hacer CRUD del modelo de forma muy sencilla.

Create (POST)

Para **crear** el modelo en el lado del servidor llamaríamos al método `save()`.

```
//Continuando con el ejemplo anterior
var usuario = new Usuario()
usuario.set({'login':'experto', 'password': '123456'})
usuario.save()
```

La creación del objeto en el servidor implica una petición POST. Antes de hacer esta petición se llama a `validate()`, y si la validación falla, `save()` devuelve `false`.

Una vez hecha la petición, Backbone espera que el servidor le devuelva un JSON incluyendo al menos la propiedad "id" con el identificador del nuevo recurso creado. Si esto se cumple, la librería establece la propiedad `id` del modelo a este valor.

Si el servidor usara una propiedad con nombre distinto a `id` para devolver el identificador del objeto, podemos poner este nombre como valor del atributo `idAttribute` del modelo.



Por defecto Backbone **no sigue el "estándar"**¹⁷ que usan algunos API REST de devolver la URL del nuevo recurso en la cabecera `Location`. Backbone ignorará la cabecera y para extraer de ella el nuevo `id` tendríamos que sobrescribir el método `save()`.

Para tener más información sobre la respuesta devuelta por el servidor debemos pasar dos *callbacks* en el `save()`, uno para llamar en caso de éxito (código de estado en el rango 200-299) y otro en caso de error:

```
usuario.save(null, {
  success: function(model, response, options){
    console.log('Modelo guardado OK');
    console.log('Id: ' + model.get('id'));
  },
  error: function(model, xhr, options){
    console.log('Error al intentar guardar modelo');
  }
});
```

También podemos usar la sintaxis de promesas. En caso de superar la validación, `save` devuelve un objeto `jqXHR`¹⁸, que es un *wrapper* de jQuery para el `XMLHttpRequest` nativo que además implementa la interfaz de promesas, así que podemos simplificar un poco la sintaxis y por ejemplo hacer más sencillo el encadenamiento de operaciones:

```
usuario.save.then(function(){
  console.log('usuario guardado OK')
  return pedido.save();
}).then(function(){
  console.log('pedido guardado OK')
});
```

READ (GET)

El método `fetch()` le pide al servidor los datos del modelo, sobrescribiendo los actuales. Asume que la respuesta va a venir en forma de objeto JSON. Para poder usar este método

¹⁷ <https://github.com/jashkenas/backbone/issues/1660>

¹⁸ <http://api.jquery.com/jQuery.ajax/#jqXHR>

el objeto ya debe tener un `id` asignado, ya que la URL a la que se va a lanzar la petición `get` es la `urlRoot + /id`.

Si los valores de los atributos procedentes del servidor difieren de los actuales se disparará un evento `change`. Posteriormente veremos cómo hacer que un objeto determinado observe un evento que genera otro.



Las interacciones con el servidor son *asíncronas*, lo que significa que tras ejecutar `fetch()` se continuará con el resto del programa aunque todavía no se haya recibido respuesta del servidor. Esto puede dar lugar a *bugs* difíciles de depurar salvo que recordemos el carácter asíncrono de la operación. Por ejemplo, en el siguiente código:

```
var u = new Usuario();
u.set("id", 1);
u.fetch();
console.log(u.login); //undefined!!!
```

La última línea imprimirá `undefined` ya que no habrá dado tiempo a que el servidor responda y a rellenar el objeto con los valores de la respuesta. Sin embargo si depuramos el código ayudándonos de un *debugger* paso a paso, daremos tiempo a que se procese la respuesta y sí mostrará el login correctamente, con el consiguiente WTF! por nuestra parte. Para poder enterarnos de cuándo se ha rellenado la información del objeto tenemos que usar callbacks en `fetch()` o bien usar eventos, como veremos al final de la sesión.

UPDATE (PUT)

La actualización se dispara con el mismo método que sirve para crear un objeto en el servidor: `save()`. Backbone asume que un modelo que tiene valor asignado a la propiedad `id` ya existe en el servidor, y por tanto al llamar a `save()` lanzará un `PUT` a `urlRoot + /id`.

DELETE (DELETE)

Para eliminar un objeto del servidor se usa `destroy()`, que lanzará una petición `DELETE` a `urlRoot + /id`, salvo que todavía no haya sido guardado en el servidor (no tenga `id`), en cuyo caso no hará petición y devolverá `false`.

2.3. Colecciones

De la mayor parte de los modelos de nuestra aplicación normalmente no habrá una única instancia, sino una colección de ellas: *posts*, *tags* o *categorías* en un blog, *mensajes*, *hilos* o *usuarios* en un foro, ...

La clase `Collection` de Backbone representa precisamente una colección de modelos. Así podemos tratarlos conjuntamente, lo que facilita la realización de ciertas operaciones, como persistir los datos en el servidor o poder escuchar eventos en cualquier modelo de la colección.

El uso de `Collection` es muy similar al de `Model`. Primero extendemos la clase y luego creamos las instancias que sean necesarias. Al extender la clase habitualmente especificaremos con la propiedad `model` el tipo de los modelos que forman la colección.

```
var Usuario = Backbone.Model.extend();
var Usuarios = Backbone.Collection.extend({model:Usuario});
var u1 = new Usuario({'login':'experto', 'password':'123456'});
```

```
var u2 = new Usuario({'login':'master', 'password':'654321'});
var lista = new Usuarios([u1,u2]);
```

Como puede verse en el constructor de la instancia podemos pasar un *array* de modelos.

Navegar por las colecciones

Podemos obtener el modelo en una posición con el método `at()`. Como los arrays, las colecciones mantienen una propiedad `length` con el número de elementos.

Si conocemos el `id` o el `cid` del modelo, podemos obtenerlo directamente con `get()`.

para **iterar por la colección** podemos usar el típico bucle `for` que vaya incrementando un índice y usar `at()`, pero también podemos usar un *iterador*.

```
misUsuarios.forEach(function(usuario) {
  console.log(usuario.get("login"));
});
```

Al `forEach` se le pasa una función, que será llamada conforme se vaya iterando por la lista. Como argumento la función recibirá el objeto en la posición actual. Este y otros métodos de manejo de colecciones y eventos procede en realidad de la librería `underscore`, que como ya hemos comentado es un prerequisite de Backbone.



Underscore¹⁹ es una pequeña librería que proporciona diversos métodos típicos de programación funcional como `map`, `filter`, `invoke`,... Además tiene pequeñas utilidades como la posibilidad de especificar *binding* de funciones, un pequeño motor de plantillas,... Es interesante echarle al menos un vistazo ya que sus funcionalidades pueden ser realmente útiles en ocasiones.

Ordenación y filtrado

En principio el **orden de los elementos** al recorrer la colección es el de inserción, pero también podemos especificar un criterio de ordenación. Para casos sencillos podemos darle al atributo `comparator` el nombre del campo usado para clasificar.

```
misUsuarios.comparator = "login";
```

Si necesitamos usar un criterio más complejo le podemos asignar a `comparator` una función con un único argumento que a partir del objeto devuelva el criterio de ordenación.

```
//Ordenar por longitud del password
misUsuarios.comparator = function(usu) {
  return usu.password.length;
}
```



Aunque Backbone (en realidad Underscore) solo nos permite ordenar en sentido ascendente, podemos usar un pequeño truco para ordenar de forma descendente: multiplicar por -1 la función de ordenación.

¹⁹ <http://underscorejs.org>

```
//Ordenar por longitud del password, pero ahora de mayor a menor
misUsuarios.comparator = function(usu) {
  return -usu.password.length;
}
```

También podemos usar una función con dos argumentos que actúe como un *comparador*: dados dos objetos a comparar devuelve -1 si el primer argumento es menor que el segundo, +1 si es mayor y 0 si son iguales.



las colecciones no se reordenan automáticamente cuando un modelo cambia el valor de alguno de sus atributos. Puedes ordenarlas de nuevo llamando a `sort()`.

Podemos filtrar una colección ayudándonos de la función `filter` de underscore. Por ejemplo, aquí vemos cómo podríamos filtrar una colección de usuarios obteniendo solo los que tienen un password de menos de 6 caracteres.

```
var passwordsCortos = lista.filter(function(usu) {
  //devolvemos true si queremos quedarnos con el objeto
  return usu.get("password").length<6;
});
```

Manipulación básica

Podemos añadir un nuevo modelo o array de modelos a la colección con `add()`. El modelo se añadirá en la posición especificada por el criterio de ordenación actual. Si queremos añadir por la cabeza usaríamos `unshift()` y por la cola `push()`. Podemos eliminar un modelo o un array de ellos con `remove()`, o eliminar el de la cabeza con `shift()` y el de la cola con `pop()`.

El método `set()` se usa para "actualizar" una colección. Si un modelo de la nueva colección no existe en la actual se añadirá, si estaba en la antigua pero no en la nueva se eliminará, y si existe en ambas se mezclarán sus atributos (los que existan en antigua y nueva se actualizarán al valor de la nueva).

Persistencia con APIs REST

Para **obtener una colección del servidor** se usa el método `fetch()`, igual que con los modelos. Si la colección no está vacía no se elimina completamente, sino que se usa el método `set()` para actualizar la del cliente.

Para **guardar la colección** en el servidor, **actualizarla** o **eliminarla** tendremos que ir procesando los modelos uno a uno. No obstante en los modelos incluidos en colecciones no es necesario especificar la `urlRoot` de cada uno por separado, se usa automáticamente la `url` de la colección como URL base.

2.4. Eventos

Los eventos son la forma de comunicación principal entre componentes de Backbone. Cuando un objeto quiere comunicar al resto que ha sucedido algo interesante, emite un evento. El resto de objetos puede suscribirse al/los eventos que desee asociados a un objeto, de modo que

cuando este emita el evento se llamará a una función que actúe de *callback*. Como vemos, es un mecanismo análogo al de los eventos en Javascript, con la diferencia de que en Javascript la mayoría de eventos vienen asociados a acciones del usuario, y en Backbone se asocian típicamente con cambios en el modelo o en las colecciones.

La documentación de Backbone incluye una [referencia de todos los eventos](#)²⁰. La gran mayoría son emitidos por modelos y colecciones, salvo unos pocos que lo son por *routers* (otros componentes de Backbone, que ya veremos en su momento).

Tratar con eventos desde objetos Javascript

En Backbone cualquier componente (modelo, vista, colección o *router*) puede observar los eventos emitidos por cualquier otro componente. Pero también podemos hacer que cualquier objeto Javascript pueda emitir y recibir eventos. También podemos hacer que cualquier objeto Javascript sea capaz de observar eventos de Backbone haciendo un *mixin* del objeto con la clase `Backbone.Events`. Es tan sencillo como llamar al método `_.extend` de Underscore pasándole como parámetros el objeto y la clase `Events`:

```
_.extend(obs, Backbone.Events);
```



Un *mixin* es un mecanismo distinto a la herencia que permite incorporar funcionalidades nuevas a un objeto. Algunos lenguajes incorporan los *mixin* de forma nativa, por ejemplo Ruby o Scala (aunque en este último se denominan *traits*). Javascript no los tiene de forma nativa pero al ser dinámico es relativamente sencillo implementarlos copiando al objeto las funciones y propiedades que queramos incorporar. Esto es de hecho lo que hace el método `_.extend()`.

Suscribirse/desuscribirse a eventos

Hay varias posibilidades para suscribirnos a los eventos que nos interese. La más usada es el método `listenTo`, al que se le pasa como parámetro el objeto a observar, el nombre del evento y la función *handler*. Por ejemplo, supongamos que desde un modelo queremos observar cuándo cambia algún atributo de otro:

```
var Usuario = Backbone.Model.extend({urlRoot: 'http://localhost:4567/usuarios'});
var usuario = new Usuario();
var MiModelo = Backbone.Model.extend({
  handler : function(modelo) {
    console.log("handler del evento 'change'")
  }
});
var observador = new MiModelo({});
//Nos suscribimos al evento 'change' sobre el modelo 'usuario'
observador.listenTo(usuario, 'change', observador.handler)
```

Recordemos que si el observador no es un componente de Backbone, primero tenemos que hacer un *mixin* con `Backbone.Events`. Lo demás es idéntico.

²⁰ <http://backbonejs.org/#Events-catalog>


```

//El objeto que va a hacer de observador
var obs = {
  handler : function(modelo, opts) {
    ...
    console.log("handler del evento 'change'")
  }
  //Más funciones y propiedades
  ...
};
//Mixin con Backbone.Events
_.extend(obs, Backbone.Events);
//Nos suscribimos al evento 'change' sobre el modelo 'usuario'
obs.listenTo(usuario, 'change', obs.handler)

```

Para dejar de escuchar todos los eventos que emite un objeto podemos usar `stopListening` pasando como parámetro el objeto que queremos "ignorar" de ahora en adelante.

```

//Continuando con el ejemplo anterior, si nos "cansamos" de escuchar
obs.stopListening(usuario);

```

Habitualmente los observadores de los eventos no serán objetos propios como en nuestro ejemplo, sino componentes de Backbone. **Típicamente son las vistas las que observan el comportamiento del modelo, lo que permite comunicarlos sin introducir acoplamiento entre ambos.** El modelo puede indicar que ha cambiado para que la vista muestre los nuevos datos, sin necesidad de mantener una referencia a la vista, ni siquiera saber cómo se llama el método de la vista que procesa los cambios.



ECMAScript 6 añade el método `object.observe`, que permite a cualquier objeto observar directamente los cambios en otro. Es de esperar que cuando el método esté [implementado en los navegadores](#)²¹ actuales cambie el funcionamiento interno de la gestión de eventos en muchos *frameworks* MVC que ahora usan técnicas propias.

Eventos para gestionar operaciones asíncronas

Antes hemos visto el caso de la operación `fetch`, para actualizar un modelo/colección con los datos del servidor, que al ser asíncrona continúa la ejecución sin haber recibido todavía los datos. Podríamos saber cuándo se han recibido por ejemplo suscribiéndonos al evento `sync`, que se dispara cuando los datos locales se sincronizan con el servidor.

```

var Usuario = Backbone.Model.extend({urlRoot: 'http://localhost:4567/
usuarios'});
var u1 = new Usuario();
u1.set("id", 1)
var obs = {
  sync_handler : function(modelo) {
    console.log("Recibido el usuario con login " + modelo.get("login"));
  }
};

```

²¹ <http://caniuse.com/#feat=object-observe>

```
_.extend(obs, Backbone.Events);
obs.listenTo(u1, 'sync', obs.sync_handler)
u1.fetch();
```

Emitir eventos de manera manual. Eventos propios

Podemos también generar un evento manualmente, incluso eventos propios. En caso de ser un evento propio lo único que tenemos que hacer es inventar un nombre para el evento. Por convenio se usa el tipo de componente y el nombre dado al evento separados por `:`. Por ejemplo `model:miEvento`

```
var Usuario = Backbone.Model.extend({urlRoot: 'http://localhost:4567/
usuarios'});
var u1 = new Usuario();
var obs = {
  miEvento_handler : function(modelo, mensaje) { ❶
    console.log("evento sobre " + modelo.cid);
    console.log("el mensaje dice " + mensaje);
  }
};
_.extend(obs, Backbone.Events);
obs.listenTo(u1, 'model:miEvento', obs.miEvento_handler);
```

❶ En un momento veremos de dónde salen los dos parámetros del *handler*. Disparamos el evento llamando a `trigger` desde el objeto que emite el evento:

```
u1.trigger("model:miEvento", u1, "¡hola!")
```

`trigger` admite un número variable de argumentos. El primero es el nombre del evento a generar y el resto son los parámetros que se le pasarán al *handler*.

2.5. Configuración de la comunicación con el API REST

Backbone sigue por defecto algunas convenciones habituales en REST a la hora de comunicarse con el API, por ejemplo que las inserciones se hacen con POST, que la URL de un modelo se obtiene concatenando el `id` con la URL de la colección, etc. Sin embargo ¿qué ocurre si nuestro API REST no sigue alguna de estas convenciones?. Tendremos que sobrescribir alguno de los métodos de Backbone para adaptarlo a nuestras necesidades.

También es muy típico el caso en el que debemos autenticarnos enviando un *api key*, bien sea en una cabecera HTTP o bien como un parámetro de la petición. Es decir, que tenemos que enviar información adicional a la que envía Backbone por defecto. Vamos a ver cómo tratar también con estos casos.

Configuración del identificador y/o la URL del modelo

Ya hemos comentado que Backbone necesita que cada modelo tenga un `id` para poder identificarlo de manera única en el servidor. Si los objetos que devuelve nuestro API siguen la misma convención no tendremos que hacer nada en especial, pero hay algunas plataformas en las que el identificador no es el atributo `id` sino que se usa otro nombre. Por ejemplo como

veréis en la asignatura de NoSQL, en MongoDB se usa el campo `_id` como identificador. En ese caso lo único que tendremos que hacer es asignar a la propiedad `idAttribute` del modelo el nombre del atributo que actúa de identificador.

Si el API devuelve un identificador más complejo (por ejemplo formado por dos atributos, o por parte de un atributo) no podemos establecer esta simple correspondencia. En ese caso lo que podemos hacer es sobrescribir el método `url()`, que debería devolver la URL del modelo, y que por defecto se obtiene como la URL "base" más el `id`. La URL base de un modelo se define bien como la `url` de la colección, si el modelo está incluido en una, bien como el valor de la propiedad `urlRoot` del modelo (que por defecto es vacío y tenemos que especificar si lo deseamos).

Por ejemplo supongamos que un API usara como identificador el atributo `id` pero luego la URL de un objeto se formara concatenando la URL base + el `id` + el sufijo `/data` (de acuerdo, es un ejemplo un poco raro pero podría ser). En el modelo haríamos algo como:

```
var MiModelo = Backbone.Model.extend({
  url: function() {
    return 'http://miapi.com/api/' + this.id + '/data';
  }
});
```

Parseo "a medida" de la respuesta del servidor

Por defecto Backbone toma la respuesta del servidor como un objeto JSON y asigna sus propiedades "de primer nivel" como atributos del modelo. Esto es porque [la implementación por defecto de la función `parse\(\)`](#)²², que es la que se usa para analizar la respuesta del servidor, simplemente devuelve tal cual el cuerpo de la respuesta:

```
parse: function (resp, options) {
  return resp;
}
```

Sin embargo hay muchos APIs que en los listados "envuelven" los resultados en un objeto que actúa como *wrapper* y los resultados en sí están dentro de él. Esto es típico de las operaciones de búsqueda o listados, por ejemplo al [buscar repositorios en el API de GitHub](#)²³. En este caso lo que haría Backbone es guardar el *wrapper* dentro del modelo, que no es lo que queremos. Tendremos pues que sobrescribir `parse()`. En el ejemplo de búsqueda en GitHub, el *wrapper* tiene una propiedad `items` donde están los resultados como un array. De modo que si tuvieramos una colección `Repositorios` tendríamos que hacer algo como:

```
var Repositorios = Backbone.Collection.extend({
  ...
  parse: function(response) {
    return response.items;
  }
  ...
});
```

²² <http://backbonejs.org/docs/backbone.html#section-75>
²³ <https://developer.github.com/v3/search/>

Autenticación

Todos los APIs en los que podamos modificar información van a requerir que nos autentiquemos de una forma u otra. Incluso muchos APIs en los que solo se puede leer información requieren del uso de una *api key* para identificar al "usuario" y evitar que un mismo usuario haga un número de peticiones excesivo. Por defecto Backbone no incorpora ningún mecanismo de autenticación, así que tendremos que añadirlo de algún modo.

Autenticación BASIC

Algunos APIs REST usan autenticación BASIC (aunque está en desuso frente a estándares más modernos como OAuth). La autenticación BASIC implica que hay que enviar el usuario y el password en una cabecera HTTP en cada petición que requiera permisos. Esto se podría implementar como veremos en la siguiente sección, pero dado que HTTP BASIC [es un estándar](#)²⁴, ya hay *plugins* de terceros listos para usar en Backbone. El más conocido es probablemente [este](#)²⁵. Lo que hace es sobrescribir la función `sync` para añadir automáticamente las cabeceras HTTP adecuadas. Para usarlo, solo hay que incluir el `.js` en la página después de incluir el Backbone original. Veamos un ejemplo de uso:

```
var Modelo = Backbone.Model.extend({
  urlRoot: 'https://miapi.com/modelos'
});

var m = new Modelo();
//Fijamos usuario y password que se enviarán al servidor cuando hagamos
//GET/POST/PUT/DELETE del modelo
m.credentials = {
  username: 'pepe',
  password: 'pepe'
}
m.set('saludo', 'Hola Backbone');
m.save().then(console.log('salvado!!'));
```

Autenticación "A medida"

La mayoría de veces la información de autenticación hay que enviarla en forma de una cabecera especial, típicamente `Authorization`, con un *token* de sesión que el servidor nos debe haber devuelto previamente al hacer login.

Hay varias formas de enviar la cabecera adicional requerida. Una solución es sobrescribir el `sync` de Backbone. Se pueden implementar algunas "menos invasivas" pero menos elegantes aprovechando que internamente Backbone usa jQuery para hacer las peticiones AJAX, por lo que podemos usar los métodos estándar de jQuery para manipular la petición. Por ejemplo el método `$.ajaxPrefilter()` nos permite modificar una petición antes de que se envíe al servidor, cambiando sus opciones, que son las mismas que podemos usar en el típico `$.ajax()`.

Por ejemplo, si tenemos que autentificarnos o enviar datos adicionales mediante cabeceras especiales, podríamos hacer algo como:

```
$.ajaxPrefilter(function (opts, originalOpts, jqXHR) {
```

²⁴ https://es.wikipedia.org/wiki/Autenticación_de_acceso_básica

²⁵ <https://github.com/fiznool/backbone.basicauth>

```
var headers = originalOpts.headers || {};  
opts.headers = $.extend(headers, {  
  "Authorization": "ponemos_lo_que_haga_falta"  
  "X-Una-Cabecera-Arbitraria": "un_valor_arbitrario",  
});  
});
```

En ocasiones es necesario enviar una `api key` como un parámetro más de la petición. Podemos hacer esto en el `fetch`

```
//supongamos que el API pide que enviemos la clave en un parámetro HTTP  
llamado "apikey"  
this.fetch({data: $.param({apikey: MI_API_KEY})
```

APIs no REST. LocalStorage.

Para los APIs que no sean del todo REST tendremos que sobrescribir el método `sync()`, que es el "corazón" de la comunicación con el servidor. Evidentemente esto va a ser mucho más complicado que todas las configuraciones que hemos visto hasta ahora. No obstante, hay ciertos casos de uso típicos para los que se han desarrollado *plugins* de terceros.

Por ejemplo, hay APIs de terceros que permiten sincronizar los datos con el *LocalStorage* del navegador en lugar de con un servidor remoto. Esto es muy interesante para aplicaciones que puedan trabajar *offline* por ejemplo agendas, listas de tareas, notas, ... el más conocido es [Backbone localStorage Adapter](#)²⁶, que nos permite sincronizar una colección automáticamente con el LocalStorage, sin más que definir una propiedad `localStorage`

```
UnaColeccion = Backbone.Collection.extend({  
  
  localStorage: new Backbone.LocalStorage("UnaColeccion"), // Un nombre  
  único dentro de la aplicación  
  
  // ... todo lo demás es igual  
});
```

Incluso hay *plugins*, como [DualStorage](#)²⁷ que permiten trabajar con el API REST remoto por defecto y cambiar de manera transparente al LocalStorage cuando se detecta que estamos *offline*.

²⁶ <http://documentup.com/jeromegn/backbone.localStorage>

²⁷ <https://github.com/nilbus/Backbone.dualStorage>

2.6. Ejercicios de modelos y colecciones



Por el momento las aplicaciones que vamos a desarrollar no tendrán interfaz, solo modelos y colecciones. Así que la forma más sencilla de probarlas es a través de la **consola de Javascript** del navegador.

Star Wars API (0,4)

Este ejercicio debes entregarlo en una carpeta llamada `star_wars`.

Vamos a probar cómo comunicarnos mediante Backbone con el [API de Star Wars](#)²⁸ que ya has usado en otros ejercicios. Como sabes, el API solo permite hacer peticiones GET, por lo que vamos a centrarnos en listar y filtrar datos.

Tendremos que adaptar la persistencia REST que implementa Backbone por defecto a las peculiaridades de este API.

- **Define una clase modelo llamada `Personaje` y una clase colección `Personajes`** formada por instancias de la anterior.
 - # Define la `url` de la colección al valor que consideres apropiado.
 - # Si haces una [petición para listar personajes](#)²⁹ verás que el objeto JSON devuelto no es directamente la lista, sino que la lista está dentro de la propiedad `results`. **Sobreescribe el método `parse()`** de la colección `Personajes` para que rellene la colección adecuadamente.
 - # Finalmente comprueba en la consola del navegador que si creas una colección vacía y haces `fetch` se llena de resultados. ¡¡Recuerda que `fetch` es asíncrono y si lo pruebas interactivamente tendrás que esperar un poco a obtener resultados!!.



Como los listados del API están paginados, al hacer un `fetch()` solo vas a obtener los 10 primeros resultados. Para arreglar esto tendríamos que **sobreescribir el método `parse()`**³⁰ para que vaya haciendo `fetch()` mientras queden resultados. No es necesario que lo hagas, trabajaremos solo con 10 resultados.

- Recuerda que Backbone necesita identificar cada modelo de manera única y que por defecto lo hace con la propiedad `id`. No obstante si accedes a <http://swapi.co/api/people> verás que los personajes no tienen `id`. Tendrás que escoger un campo que sirva de identificador y **definir la propiedad `idAttribute` de `Personaje`** para indicar que este debe actuar de identificador. Para comprobar que funciona, crea de nuevo una colección vacía, haz `fetch()` y luego imprime el `id` del primer objeto, algo como:

```
var lista = new Personajes();
lista.fetch().then(function(){
  console.log(lista.at(0).id);
});
```

- Modifica la colección `Personajes` para hacer que **la colección esté ordenada** alfabéticamente por nombre de manera ascendente.

²⁸ <http://swapi.co/>

²⁹ <http://swapi.co/api/people>

³⁰ <http://stackoverflow.com/questions/13753540/appendng-data-to-same-collection-after-every-pagination-fetch>

- Añade un método a la colección llamado `buscarPorNombre(cadena)`, que la filtre devolviendo solo aquellos personajes cuyo nombre contenga la subcadena especificada.

Comunicación con un API REST completo (0,6)

Este ejercicio debes entregarlo en una carpeta llamada `alquiler_coches`. Seguiremos trabajando sobre la misma carpeta en más sesiones.

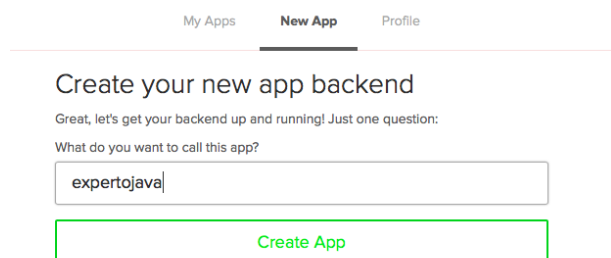
Vamos a ir creando en sucesivas sesiones una aplicación para una compañía de alquiler de coches. En concreto vamos a ir desarrollando solo la parte de administración en la que se podrá listar los vehículos, darlos de alta/baja, editarlos,...

Para no tener que programarnos el *backend* desde cero, ya que no es el objetivo de la asignatura, usaremos una plataforma de tipo *BaaS (Backend As A Service)*, con la que podemos crear un *backend* de tipo REST de manera sencilla.

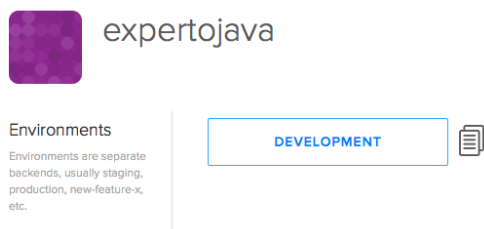
Uso básico de la plataforma BaaS

Usaremos una plataforma llamada [Kinvey](#)³¹. Aunque ofrece otros servicios, el que nos interesa es el de *DataStore*, con el que podemos hacer CRUD de objetos en el servidor mediante un API REST. Los objetos no son más que conjuntos de pares *propiedad-valor*, al igual que en Backbone.

Para usar la plataforma primero hay que [darse de alta](#)³² como desarrollador. Una vez dados de alta, pulsamos sobre "Get Started with your first app" y comenzamos a crear una app en el servidor. Lo primero será elegir un nombre, por ejemplo `expertojava`.



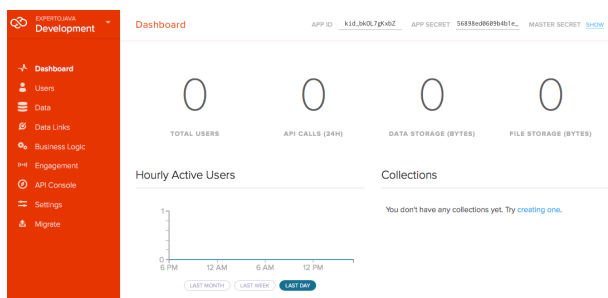
Una vez creada (tardará unos segundos), pasamos a editar el entorno de `Development`.



Aparecerá nuestro *dashboard*.

³¹ <http://kinvey.com>

³² <https://console.kinvey.com/sign-up>



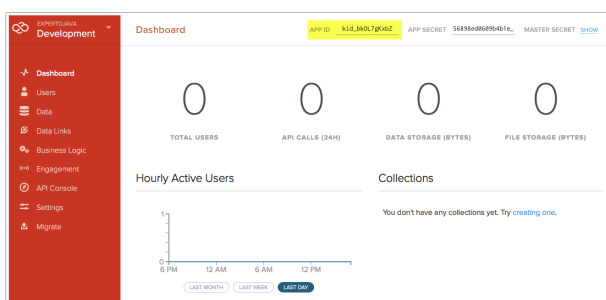
Lo primero que podemos hacer es crear una colección de entidades, el equivalente a una colección de Backbone pero en el lado del servidor. Si nos fijamos, en la parte derecha de la pantalla tenemos un apartado 'Collections' que nos invita a crear una nueva. Para crearla solo necesitamos darle un nombre, en nuestro caso `coches`.

Una vez creada la colección (vacía, por el momento) debemos crear un nuevo usuario, ya que desde el lado de Backbone siempre hay que autenticarse para hacer cualquier operación con el *backend*. En el menú lateral izquierdo, hacemos *click* sobre `Users`, y luego en la barra de herramientas de la parte superior, sobre el de `Add user`. Nos pedirá simplemente un `username` y un `password` (este no es nuestro usuario de Kinvey, es un usuario ficticio para poder probar nuestra *app*, valdrá cualquiera).

Ahora ya podemos probar el *backend*. Antes de ponernos con backbone podemos hacerlo desde la terminal de linux, con `curl`. Vamos a probar a crear un nuevo coche

```
curl -H 'Content-type:application/json' --user MI_USUARIO:MI_PASSWORD
-d '{"matricula":"1111AAA", "modelo":"Opel Corsa"}' https://
baas.kinvey.com/appdata/MI_APP_ID/coches
```

CUIDADO: hay que cambiar `MI_USUARIO` y `MI_PASSWORD` por los del usuario que acabamos de dar de alta para nuestra *app*, y `MI_APP_ID` por el identificador único para nuestra *app*, podemos verlo en la parte superior del *dashboard*.



una vez ejecutado el comando, si todo ha ido bien el servidor debería responder con un JSON con los datos del nuevo objeto creado, y al examinar la colección `Coches` desde el *dashboard* deberían aparecer los datos del coche dado de alta.

El modelo `Coche` en Backbone

Por el momento los coches tendrán una `matricula`, un `modelo`, un `kilometraje` y un valor `disponible` indicando si está disponible o por el contrario está alquilado.

- **Crea un modelo de Backbone** llamado `Coche`.

- Adapta el modelo para que pueda comunicarse con el API remoto, de manera similar a como hiciste en el API de Star Wars

Define el valor de la propiedad `urlRoot` para que se pueda sincronizar el modelo con el servidor aunque no esté dentro de una colección. Recuerda que antes sincronizábamos con el servidor una colección, y en ese caso la propiedad se llama `url`. Para un modelo se llama `urlRoot`. Recuerda que la URL sería `https://baas.kinvey.com/appdata/MI_APP_ID/coches`, sustituyendo `MI_APP_ID` por el de nuestra *app*.

Define como campo `id` el que Kinvey llama `_id`.

Usa el *plugin* de autenticación BASIC para backbone que hemos visto en los apuntes, ya que todas las peticiones a Kinvey deben estar autenticadas.

- Implementa un **método de validación** para comprobar al menos que la matrícula está formada por 4 dígitos seguidos de 3 letras.

Para probar lo implementado, en la consola Javascript crea manualmente coches, asignándoles valores a los campos y comprueba interactivamente que se pueden guardar, recuperar, y modificar. Comprueba que si la matrícula no es válida no se llega a hacer la petición al servidor.

- Crea la colección de Backbone* `ListaCoches`, formada por instancias del modelo anterior.

Define su `url` como proceda para poder comunicarse con Kinvey

Usa el *plugin* de autenticación BASIC

- Comprueba que al hacer `fetch()` de la colección obtienes los mismos modelos que ves en tu *dashboard* de la web de Kinvey.
- Crea un método `listarDisponibles` que filtre los coches mostrando solo los que están disponibles para alquilar.

3. Vistas y templates. Routing

3.1. Vistas

Las vistas son los componentes que se encargan de mostrar la información al usuario. En otros *frameworks* MVC las vistas son *plantillas*: mitad HTML, mitad variables e instrucciones, que son el esqueleto de lo que el usuario va a ver en su pantalla. Esto sucede por ejemplo en Rails (Ruby), en Spring MVC (Java),... En Backbone, por el contrario, **las vistas son código javascript**. Este código genera el HTML de la interfaz y encapsula los manejadores de evento que se ocupan de las acciones del usuario. En Backbone se pueden usar *templates* para generar el HTML, pero en cuanto a si usarlas o no o qué motor de plantillas usar, es totalmente "agnóstico".

La vista más simple que podemos crear en Backbone es la que aparece a continuación, aunque es un poco "aburrida", ya que está prácticamente vacía. Como se puede ver, la mecánica es similar a la de crear un modelo

```
//Creamos la "clase" Vista
var Vista = Backbone.View.extend();
//Instanciamos una vista
var unaVista = new Vista();
```

La propiedad "el"

Todas las vistas tienen una propiedad predefinida llamada `el`, **que representa el nodo del DOM que es la raíz del HTML de la vista**. La vista genera HTML y lo coloca dentro de `el` (luego veremos cómo se hace esto habitualmente). Después nosotros somos los responsables de tomar esa propiedad `el` e insertarla en el lugar que queramos del DOM.

```
var Vista = Backbone.View.extend();
var unaVista = new Vista();
//generamos el HTML y lo metemos en 'el' (todavía no aparecerá en
//pantalla)
//normalmente no se suele manipular 'el' desde fuera, esto es solo un
//ejemplo
unaVista.el.innerHTML('Hola soy una vista de Backbone')
//Añadimos el HTML generado al cuerpo de la página
document.body.appendChild(unaVista.el);
```

Por defecto, `el` es una etiqueta `<div>`. Si ejecutamos el código anterior veremos que por tanto se le añade un `<div>` a la página con el mensaje que hemos puesto.

Podemos darle el valor que queramos a `el` si no nos interesa el valor por defecto. De hecho podemos configurarla totalmente a nuestra medida con una serie de atributos

```
<div id="miVista">
</div>
<script type="text/javascript">
var Vista = Backbone.View.extend();
var unaVista = new Vista({
  tagName: 'span',
  className: 'vista',
```

```
    id: 'vista_principal',
    attributes: {'data-fecha': new Date()}
  });
  document.body.appendChild(unaVista.el)
</script>
```

Al ejecutar el código anterior al cuerpo de la página se le añadirá un HTML como este

```
<span data-fecha="Thu Jan 29 2015 11:39:38 GMT+0100
(CET)" id="vista_principal" class="vista"></span>
```

Hasta ahora hemos observado que la vista genera el HTML pero nosotros somos los responsables de incluirlo en la página. Hay otra posibilidad: darle al `el` como valor el `id` de algún nodo de la página. Así, al poner

```
<div id="miVista">
</div>
<script>
var Vista = Backbone.View.extend();
var unaVista = new Vista({el: '#miVista'});
unaVista.el.innerHTML = "Hola yo ya estoy en la página";
</script>
```

El contenido ya aparecería insertado en el DOM de la página actual.

Como ya hemos dicho en otras ocasiones Backbone facilita el trabajo con jQuery. En este caso tiene predefinida una propiedad `$el` que representa lo mismo que `el` pero es un objeto jQuery en lugar de un nodo DOM estándar, por lo que podemos usar el API de jQuery si nos resulta más cómodo:

```
unaVista.$el.html("Hola estoy dentro de la vista");
```

Ya hemos visto que la inclusión de la vista en el DOM se hace manualmente o bien poniendo como valor de `el` un nodo ya existente en el DOM. Para eliminar la vista del DOM se usa el método `remove()`.

Rendering

Hasta ahora hemos estado manipulando directamente el `el` para incluir contenido en la vista, pero esta forma de trabajar no es muy "limpia" que digamos. **La convención habitual en Backbone es sobrescribir el método `render()`**, que debería rellenar el contenido del `el`, generando el HTML de la vista. Y decimos convención ya que si examinamos los fuentes de Backbone veremos que el resto del código no llama a `render()` en ningún momento, y la implementación por defecto **no hace nada**³³ (salvo devolver `this`, hablaremos ahora sobre esto).

Así, los ejemplos que hemos puesto hasta ahora quedarían mejor como:

```
var Vista = Backbone.View.extend({
```

³³ <http://backbonejs.org/docs/backbone.html#section-135>

```

render: function() {
    this.$el.html("Hola soy una vista")
    return this; ❶
}
});
var unaVista = new Vista();
$('body').append(unaVista.render().$el) ❷

```

Nótese que:

- ❶ Por convenio `render()` devuelve la vista, lo que es cómodo porque permite encadenar las llamadas, al estilo jQuery: (`render().$el`).
- ❷ Debemos llamar explícitamente a `render()` para rellenar el contenido del `el`. Ni Backbone ni nadie lo va a hacer por nosotros.



Volvemos a recalcar que `render()` es simplemente una convención. Podríamos llamar al método que genera el HTML `pintar()` y funcionaría igual, ya que los responsables de llamarlo somos nosotros. No obstante todos los desarrolladores de Backbone suelen respetar la nomenclatura estándar. Así, cuando se lee código Backbone y se ve el `render()` uno ya sabe a qué atenerse. Por supuesto en una SPA es de esperar que haya formas de *renderizar* solo parte de la vista. Pero para eso ya no hay un estándar, definiremos los métodos propios que deseemos.

Eventos

Las vistas que solo muestran contenido estático no son muy divertidas. Normalmente nos interesará que sean interactivas y respondan a eventos. La gestión de eventos también es responsabilidad de la vista, y se define en un objeto en formato JSON llamado `events`. Las propiedades son nombres de eventos (y de manera opcional un selector CSS indicando el nodo o nodos DOM al que afecta). Los valores son cadenas con el nombre del manejador correspondiente. Por ejemplo:

```

var Vista = Backbone.View.extend({
  render: function() {
    this.$el.html("Ahora soy una vista interactiva <br>");
    this.$el.append('<input type="button" class="boton" value="Haz clic">');
    return this;
  },
  verMensaje : function() {
    console.log("Hola!!!");
  },
  events : {
    'click .boton' : 'verMensaje'
  }
});
var unaVista = new Vista();
$('body').append(unaVista.render().el);

```

El selector CSS se busca únicamente dentro de la vista. En el ejemplo, si en la página (fuera de la vista) hubiera otras etiquetas con `class="boton"` no se verían afectadas por esta gestión de eventos. Esto es interesante porque hace a las vistas *modulares* y *autocontenidas*.



En listados de datos es muy habitual, como veremos en la siguiente sesión, que cada elemento del listado sea una vista distinta. El manejo separado de eventos permite que todas puedan coexistir, cada una procesando sus propios eventos y sin interferir con las demás.

Podemos modificar dinámicamente la gestión de eventos llamando al método `delegateEvents()` y pasándole un objeto JSON con el nuevo valor a darle a `events`.

3.2. Vistas y modelos

Relación entre vista y modelo

Hasta ahora hemos hablado de vistas, pero ¿qué relación mantienen con los modelos?. La idea es que cada vista normalmente tiene una referencia al modelo o colección que representa. Hasta ahora en los ejemplos que hemos visto no había modelo, pero esto en realidad no es lo habitual. Es más habitual algo como:

```
var Libro = Backbone.Model.extend();
var unLibro = new Libro({titulo:"El mundo del río", autor:"P.J.Farmer"});
var Vista = Backbone.View.extend({
  render: function() {
    this.$el
      .append("<b>"+this.model.get("titulo") + "</b>")
      .append("<br> <em>"+this.model.get("autor") + "</em>")
    return this;
  }
});
var unaVista = new Vista(model: unLibro);
$('body').append(unaVista.render().$el)
```

Código³⁴ en JSbin.com

Con la propiedad `collection` podemos pasarle una colección a la vista.

Data binding

El *data binding* es la vinculación entre ciertos componentes del modelo y de la vista, de modo que cuando cambia uno de ellos el otro se actualiza automáticamente. La vinculación puede ser solo en un sentido o en ambos. La de un solo sentido suele funcionar del modelo hacia la vista (si cambia el primero se actualiza la segunda) pero no al contrario. La bidireccional se suele usar en formularios, cuando estamos editando los datos del modelo.

Backbone no tiene *data binding* propiamente dicho, ya que el único momento en que están vinculados los datos del modelo y la vista es justo cuando se hace un *render* de la vista.

Data binding con eventos

En Backbone es habitual vincular el modelo con la vista usando eventos. Las vistas pueden suscribirse a eventos del modelo. Al recibir el evento la vista debe hacer un *rendering* parcial, modificando únicamente la parte que no varía. Backbone no va a ayudarnos en esto último, tendremos que hacerlo nosotros mismos. Por ejemplo, supongamos que tenemos un *widget* que monitoriza el estado de un servidor y debe actualizar la vista automáticamente cuando cambie éste:

³⁴ <http://jsbin.com/rovak/1/edit>

```

var Servidor = Backbone.Model.extend();
var miServidor = new Servidor({estado:"funcionando"});
var VistaServidor = Backbone.View.extend({
  initialize: function() {
    this.listenTo(this.model, 'change:estado', this.renderEstado) ❶
  },
  render: function() {
    this.$el.html('El servidor está: <span id="estado">' ❷
      + this.model.get('estado') + '</span>');
    return this;
  },
  renderEstado: function() { ❸
    $('#estado').text(this.model.get('estado'))
  }
});
var miVista = new VistaServidor({model: miServidor});
$('body').append(miVista.render().$el);

```

- ❶ Suscribimos a la vista a los cambios de la propiedad `estado` de su modelo asociado.
- ❷ Marcamos una parte del HTML con el `id="estado"` para luego poder cambiar su valor directamente.
- ❸ El método `renderEstado` solamente cambia el HTML que muestra directamente el estado del servidor, no toda la vista.

Si ahora cambiara el valor de la propiedad `estado` del modelo el estado se actualizaría sin tener que redibujar totalmente la vista. Podemos probarlo de manera sencilla tecleando en la consola del navegador:

```
miServidor.set("estado", "parado")
```

Data binding automático

Aunque ya hemos dicho que Backbone tal cual no tiene *binding automático*, existen varios *plugins* que proporcionan esta funcionalidad. Uno de los más conocidos es `stickit`³⁵, que vamos a ver aquí brevemente. Hay otros como por ejemplo `backbone UI`³⁶, que además incluye `widgets` o `backbone baguette`³⁷.

Veamos un ejemplo de cómo conseguir *data binding* automático desde el modelo hacia la vista. Hasta cierto punto la idea es similar a lo que hacíamos antes con los eventos: en la vista debemos tener ciertas secciones del HTML marcadas indicando que ahí van los datos que queremos vincular. La diferencia es que `stickit` los actualizará automáticamente por nosotros sin necesidad de gestionar los eventos ni implementar el *rendering* parcial.

```

var Libro = Backbone.Model.extend();
var unLibro = new Libro({'titulo':'Juego de tronos', 'autor':'George R.R.
  Martin'});
var VistaLibro = Backbone.View.extend({
  render: function() {
    this.$el.html('<b id="titulo"></b>, de <em id="autor"></em>'); ❶
  }
});

```

³⁵ <http://nytimes.github.io/backbone.stickit/>

³⁶ <http://perka.github.io/backbone-ui/>

³⁷ <http://spacenic.github.io/backbone-baguette/>

```

    this.stickit(); ❷
    return this;
  },
  bindings: { ❸
    '#titulo': 'titulo',
    '#autor' : 'autor'
  }
});
var miVista = new VistaLibro({model:unLibro});
$('body').append(miVista.render().$el)

```

- ❶ En el HTML de la vista marcamos (en este caso usando `id`) las partes donde luego queremos que se coloquen los datos. Esto elimina la necesidad de colocar incluso el valor inicial del dato, ya que `stickit` se encargará de ello automáticamente.
- ❷ Para que funcione correctamente `stickit` debemos incluir esta línea al final del método `render`.
- ❸ Definimos un conjunto de pares "propiedad":"valor" llamado `bindings` y muy similar en formato al `events` de Backbone. Pero en este caso la propiedad es un selector CSS que identifica en la vista dónde está un dato y el valor es el nombre del atributo del modelo que queremos colocar allí.

Si ahora modificamos el modelo, la vista se actualizará automáticamente, por ejemplo podemos ejecutar en la consola Javascript la siguiente línea para ver cómo se actualiza la vista

```
unLibro.set("titulo", "Tormenta de espadas")
```

`Stickit` soporta también el *data binding* bidireccional. Lo único que hay que hacer en el ejemplo anterior es cambiar las etiquetas `` y `` por campos de formulario de tipo texto, por ejemplo. Podremos observar que cuando se modifica el contenido del campo el atributo del modelo refleja el cambio.

Los ejemplos anteriores son con la configuración de la librería por defecto. Podemos forzar el tipo de vinculación que queramos (por ejemplo solo de una dirección en campos de formulario), configurar los eventos de vista que disparan los cambios en el modelo, incluir nuestros propios *handlers*,... La librería es bastante completa y flexible, aquí solo queremos mostrar una pequeña introducción a cómo funcionaría el *data binding* integrado con Backbone.

3.3. *Templates* (plantillas). El lenguaje de plantillas *Mustache*

Con el último ejemplo podemos intuir que cuanto más se complique el HTML que debe generar la vista más engorroso va a ser el código, hasta llegar a un punto que lo haga inmanejable para vistas complejas. La solución es la misma a la que se llegó en el lado del servidor hace ya años, en aplicaciones "clásicas" en las que el servidor debe enviar al cliente la página totalmente formada: usar plantillas (*templates*). Ejemplos clásicos de lenguajes que podríamos considerar de plantillas son JSP, ASP, PHP,...

Al igual que las del servidor, las *templates* del cliente son fragmentos de HTML con variables intercaladas, y suelen incluir secciones condicionales y secciones repetidas. Aunque las plantillas del lado del servidor pueden incluir típicamente instrucciones arbitrarias de algún lenguaje de programación, dicha posibilidad nunca ha sido muy bien vista desde una perspectiva "purista", ya que acaba mezclando lógica con presentación. Las plantillas definidas en el cliente no suelen usar esta funcionalidad, limitándose habitualmente a condicionales y bucles sencillos.

Backbone en sí no incluye ningún lenguaje de *templates* ni facilita especialmente la integración con ninguno en concreto. Eso sí, la librería `underscore`, que es un requisito de Backbone, incluye un [pequeño lenguaje de plantillas](#)³⁸ que es una elección razonable para casos sencillos.

Nosotros veremos aquí un lenguaje de plantillas algo más sofisticado que el de `underscore` (no mucho más) pero que es mucho más usado en la web: Mustache.

[Mustache](#)³⁹ es un lenguaje de plantillas del que existen implementaciones en los entornos y lenguajes de programación más variopintos. No solo Javascript, sino también Java, Ruby, Python, Scala, .NET, Android,... Como puede deducirse de esta lista, se puede usar tanto en el lado del cliente como del servidor.

La baza principal de Mustache es la simplicidad: aunque se pueden mostrar partes de manera condicional y se puede iterar por listas de valores, no se hace explícitamente con sentencias condicionales o con bucles. Todo se hace con lo que en Mustache se llaman etiquetas o *tags*, que no son precisamente como las de HTML.

Existen implementaciones alternativas a la "de referencia" que incluyen algunas funcionalidades adicionales: muy conocidas son por ejemplo [Handlebars.js](#)⁴⁰ o [Hogan](#)⁴¹.

Sintaxis básica

La idea básica es que una plantilla más un "objeto" formado por pares propiedad-valor va a generar el resultado final. En la plantilla se toma todo como literal excepto las partes entre dobles llaves (`{{ }}`), que representan variables o indican secciones especiales, como ahora veremos.

La sintaxis del lenguaje se puede consultar en el [manual online](#)⁴². Vamos a ver un ejemplo que incluye todas las características típicas que vamos a necesitar aquí:

```

<!-- Plantilla -->
<p>Bienvenido a <b>{{lenguaje}}</b>, {{#usuario}}{{nombre}}{}/usuario}}.
  Vamos a usar:</p> ❶
<ul>
  {{#frameworks}} ❷
    <li>{{nombre}} ({{lenguaje}})</li>
  {}/frameworks}}
</ul>
{{#aviso}}Este curso puede ser peligroso para tu salud{}/aviso}}

```

```

//Datos
{
  "lenguaje": "Mustache",
  "usuario": {"nombre": "ExpertoJava", "curso": "2014-15"},
  "frameworks": [ ❶
    {"nombre": "Backbone", "lenguaje": "JS"},
    {"nombre": "Angular", "lenguaje": "JS"},
    {"nombre": "REStEasy", "lenguaje": "Java"},
  ]
}

```

³⁸ <http://underscorejs.org/#template>

³⁹ <https://mustache.github.io/>

⁴⁰ <http://handlebarsjs.com/>

⁴¹ <http://twitter.github.io/hogan.js/>

⁴² <https://mustache.github.io/mustache.5.html>


```

],
  aviso: false ❷
}

```

```

<!-- Resultado final -->
<p>Bienvenido a <b>Mustache</b>, ExpertoJava. Vamos a usar:</p>
<ul>
  <li>Backbone (JS)</li>
  <li>Angular (JS)</li>
  <li>RESTEasy (Java)</li>
</ul>

```

- ❶ cuando aparece un identificador entre dobles llaves se sustituye por el valor de la correspondiente propiedad.
- ❷ También se pueden usar *secciones*, que se marcan convencionalmente como `{{#seccion}}...{{/seccion}}`.
- ❶ Si la sección se corresponde en los datos con una *lista*, se va iterando por ella. El objeto que se corresponde con la sección es ahora el que marca el contexto de donde tomamos las propiedades.
- ❷ Podemos usar una sección que se corresponda con una propiedad *booleana* para implementar partes condicionales. En el resultado final no aparece el último mensaje porque la propiedad correspondiente es `false`.

Plantillas en el lado del cliente

Estando en el lado del cliente, y usando HTML+JS veamos dónde almacenaríamos la plantilla, de dónde sacaríamos los datos y cómo uniríamos ambos elementos para obtener el resultado final.

Rendering

En primer lugar, **la plantilla no es más que una cadena**, por tanto la podríamos almacenar en una variable Javascript:

```

var plantilla = "<p>Bienvenido a <b>{{lenguaje}}</b>, {{#usuario}}
{{nombre}}</usuario}}. Vamos a usar:</p>";
plantilla += "<ul> {{#frameworks}} <li>{{nombre}} ({{lenguaje}})</li> {/
frameworks}}";
plantilla += "{{#aviso}}Este curso puede ser peligroso para tu salud{/
aviso}}</ul>"

```

Este código es muy engorroso, en un momento veremos algunas formas de solucionarlo. Por el momento vamos a usarlo tal cual.

La lista de pares propiedad-valor no es más que un objeto Javascript, por lo que solo nos queda unir las dos partes para obtener el resultado final. En [la implementación Javascript](#)⁴³ de Mustache esto se hace con el método `Mustache.render`:

```

//opcionalmente podemos ejecutar esta línea, que "compilará" la plantilla
y la cacheará

```

⁴³ <https://github.com/janl/mustache.js>

```
Mustache.parse(template)
//unir plantilla y datos para generar texto resultante
var html = Mustache.render(template, datos);
```

Dónde almacenar la plantilla

Es evidente que este enfoque se va haciendo inmanejable conforme crece el tamaño y complejidad de la plantilla. Una posibilidad sería **almacenar la plantilla en un archivo aparte** y acceder a ella con una petición AJAX. Por ejemplo, usando jQuery:

```
$.get('plantilla.mustache', function(template) {
  var res = Mustache.render(template, datos);
  ...
});
```

Otra posibilidad es **almacenar la plantilla en la propia página**, pero necesitamos que el navegador lo ignore para que no lo muestre tal cual en la página. Un truco muy usado es incluir la plantilla dentro de una etiqueta `<script>` con un `type` no estándar (cualquiera, una cadena inventada al estilo `type="text/x-tmpl-mustache"`). Si el navegador no reconoce el valor de dicho atributo simplemente ignorará el contenido de la etiqueta, asumiendo que es un *script* en algún extraño lenguaje de programación para el que no tiene intérprete:

```
<script id="miTemplate" type="text/x-tmpl-mustache">
  <p>Bienvenido a <b>{{lenguaje}}</b>, {{#usuario}} {{nombre}} {{/
usuario}}. Vamos a usar:</p>
  <ul>
    {{#frameworks}}
      <li>{{nombre}} ({{lenguaje}})</li>
    {{/frameworks}}
  </ul>
  {{#aviso}}Este curso puede ser peligroso para tu salud{{/aviso}}
</script>
```

y ahora accederíamos al nodo correspondiente del DOM, por ejemplo usando jQuery:

```
var res = Mustache.render($('#miTemplate').html(), datos);
```

Uso típico de plantillas en Backbone

Como Backbone en sí no prevé ni aporta nada con respecto a las plantillas, en realidad no hay una "forma correcta" de usar plantillas en el código Backbone, pero hay algunos fragmentos de código típicos que se repiten más o menos literalmente en muchas aplicaciones Backbone. Vamos a ver un par de casos de uso.

Una vista que se corresponde con un único modelo

En este caso, la única diferencia con lo que veníamos haciendo hasta el momento es llamar a `Mustache.render` dentro del `render` de Backbone en lugar de generar "manualmente" el HTML

```

<script id="template_libro" type="text/x-tmpl-mustache">
  <b>{{titulo}}</b>, por <em>{{autor}}</em>
</script>
<script type="text/javascript">
  var Libro = Backbone.Model.extend();
  var unLibro = new Libro({titulo:"Juego de Tronos",
                          autor:"George R.R. Martin"});
  var Vista = Backbone.View.extend({
    template: $('#template_libro').html(),
    render: function() {
      var res = Mustache.render(this.template, this.model.toJSON())
      this.$el.html(res)
      return this;
    }
  });
  var unaVista = new Vista({model:unLibro})
  $('body').append(unaVista.render().$el)
</script>

```

Una vista que se corresponde con un listado de modelos

Como veremos en la siguiente sesión, cuando tenemos listas editables es mucho mejor hacer que cada elemento del listado sea una subvista de la vista principal, pero cuando simplemente queremos listar datos que van a ser "estáticos" en la página, podemos usar una única vista para todos. Si usamos Mustache no va a haber prácticamente diferencia con lo anterior, ya que las secciones `{{#}}` y `{{/}}` nos permiten iterar implícitamente por los datos.

La plantilla sería algo como

```

<script id="template_lista" type="text/x-tmpl-mustache">
  {{#.}}
  <b>{{titulo}}</b>, por <em>{{autor}}</em> <br>
  {{/.}}
</script>

```

Nótese que si la sección por la que vamos a iterar se corresponde con el "nivel superior" del objeto y no con una propiedad entonces podemos usar el símbolo "." Eso nos permite iterar por un array JSON: `[{'titulo':'Tormenta de espadas', 'autor':'George R.R. Martin'}, {'titulo':'Beginning Backbone', 'autor':'James Sugrue'}]`.

Además de esto la diferencia en el Javascript es que en lugar de serializar en JSON un único modelo serializamos una colección.

```

var Libro = Backbone.Model.extend();
var unLibro = new Libro({'titulo':'Juego de tronos', 'autor':'George R.R.
  Martin'});
var otroLibro = new Libro({'titulo':'Tormenta de
  espadas', 'autor':'George R.R. Martin'});
var Biblioteca = Backbone.Collection.extend({
  model:Libro
});
var miBib = new Biblioteca([unLibro, otroLibro]);

```

```

var VistaBiblioteca = Backbone.View.extend({
  template: $('#template_lista').html(),
  render: function() {
    this.$el.html(Mustache.render(this.template, this.collection.toJSON()));
  }
});

```

3.4. Routers

Todos los *frameworks* web en el lado del servidor implementan de un modo u otro la idea de mapeado de **rutas**: especificamos qué se va a ejecutar cuando se reciba una petición a determinada URL. Habitualmente la URL no tiene por qué ser literal sino que se pueden usar variables, expresiones regulares, etc. Como ya sabemos, en JavaEE las rutas se pueden configurar en el `web.xml` o bien especificar directamente en el código con la anotación `@Path`.

En las SPAs por su propia naturaleza no hay cambio de URL cuando el usuario va realizando operaciones en la aplicación. Hasta el momento nosotros no hemos tenido que asociar rutas con ningún caso de uso. Pero esto representa un problema desde el punto de vista de la *usabilidad*. En la web el usuario depende de la URL para poder volver en otro momento a acceder a la información, pero ahora mismo tal y como funcionan nuestras aplicaciones, los *bookmarks* son inútiles: todos apuntarían al HTML que se cargó originalmente con la aplicación, pero no al estado actual de la misma.

Los **routers** intentan resolver este problema. Permiten asociar a una URL un código a ejecutar.

Routers básicos

Para crear un *router* debemos extender la clase `Backbone.Router`. Dicha clase tiene una propiedad básica, `routes`, que es un conjunto de pares clave/valor, al estilo del `events` de las vistas. La clave es la ruta, y el valor el nombre de la función a ejecutar. Por ejemplo:

```

var MiRouter = Backbone.Router.extend({ ❶
  routes: {
    'hola' : 'holaRouter'
  },
  holaRouter: function() {
    console.log("Hola Router");
  }
});
var unRouter = new MiRouter(); ❷
Backbone.history.start(); ❸

```

- ❶ Extendemos la clase `Backbone.Router` y definimos la propiedad `routes`
- ❷ Creamos una instancia de router
- ❸ Esta instrucción es necesaria para que Backbone "escuche" los cambios en la URL. Suponiendo que la página que contiene el código anterior fuera `index.html` si en la barra de direcciones del navegador cambiamos la URL por `index.html#hola`, veremos aparecer el mensaje en la consola Javascript.

Nótese que en las rutas tal y como las ve el usuario, **lo que cambia entre una ruta y otra es el hash fragment**, es decir, la parte que va detrás del símbolo `#`. Para usar URLs convencionales habría que configurar la parte del servidor, como veremos luego.

Rutas con partes variables

Podemos definir partes variables en una ruta poniéndoles un nombre precedido del símbolo `!`. La función Javascript asociada a la ruta recibirá tantos parámetros JS como partes variables tenga la ruta.

```
var MiRouter = Backbone.Router.extend({
  routes: {
    'hola/:nombre' : 'holaRouter'
  },
  holaRouter: function(nom) {
    console.log("Hola " + nom);
  }
});
```

Podemos poner varias partes variables en una ruta. La siguiente ruta encajaría con `#hola/Pepe/Pérez`.

```
routes: {
  'hola/:nombre/:apellidos' : 'holaRouter'
}
```

Para especificar alguna parte como opcional la pondríamos entre paréntesis:

```
routes: {
  'hola/:nombre(/:apellidos)' : 'holaRouter'
}
```

En el ejemplo anterior, la ruta encajaría tanto con `#hola/Pepe/Pérez`, como simplemente con `#hola/Pepe`. En este último caso, el parámetro Javascript asociado a los apellidos sería `null`.

Las partes variables de las rutas se tratan al estilo de las expresiones regulares. Por ejemplo podemos poner una parte fija mezclada con la variable, o usar el símbolo `*` para indicar cualquier secuencia de caracteres. Esto último es útil si queremos recoger un *path* completo, o sea una cadena que contenga también el carácter `/`.

```
routes: {
  'hola/Pep:sufijo' : 'holaRouter'
  'adios/*var' : 'adiosRouter'
}
```

La ruta `#hola/Pepito` encajaría con la primera expresión, por lo que la variable adquiriría el valor `ito`. Si fuéramos por ejemplo a `#adios/mas/cosas/por/aqui`, el parámetro JS asociado a la variable `var` tomaría el valor `mas/cosas/por/aqui`.



si la URL encaja con más de una ruta se usará la primera ruta que encaje.

Rutas por defecto

Es conveniente definir un par de rutas por defecto en cualquier aplicación: la ruta vacía `''`, que se usa cuando el *hash fragment* está vacío, y `*default`, que se usará con la URL actual si esta no encaja con ninguna de las definidas en el *router*.

```
var MiRouter = Backbone.Router.extend({
  routes: {
    '' : 'vacía',
    '*default': 'defecto'
  },
  defecto: function(path) { ❶
    console.log('Ruta por defecto: ' + path);
  },
  vacía: function () {
    console.log('Ruta vacía');
  }
});
```

❶ En el caso de la ruta por defecto, el parámetro JS asociado contendrá el *path* completo.

Navegación en el código

En cualquier momento **podemos navegar a una URL determinada** con el método `navigate` de la clase `Router`:

```
miRouter.navigate('hola/Pepe')
```

Esta operación añadirá la nueva URL al historial del navegador. No obstante, navegar a una URL **por defecto no disparará la función asociada a la ruta** correspondiente, salvo que lo especifiquemos con `{trigger:true}`

```
miRouter.navigate('hola/Pepe', {trigger: true})
```

Al contrario, podemos **detectar en nuestro código que se ha disparado una ruta determinada** respondiendo a los eventos con prefijo `route:`. El nombre completo del evento se obtiene añadiendo a este prefijo el nombre de la función asociada. Por ejemplo:

```
var MiRouter = Backbone.Router.extend({
  routes: {
    'hola': 'holaRouter'
  },
  holaRouter : function () {
    console.log("Hola Router")
  }
});
```

```
var unRouter = new MiRouter();
Backbone.history.start();
unRouter.on('route:holaRouter', function() {
  console.log("Se ha disparado la función holaRouter");
})
```

URLs completas

Usar una URL con *hash fragments* simplifica la gestión para Backbone, ya que en realidad no estamos cambiando de página. Pero puede parecer "algo rara" para el usuario. Podemos configurar Backbone para usar URLs convencionales, pero hay que solucionar dos pequeños problemas:

- El navegador debe ser compatible con el *history API* de HTML5. Este API permite manipular el historial de navegación y Backbone lo usa para cambiar la URL sin tener que hacer nuevas peticiones HTTP.
- Debemos configurar el servidor para que todas las peticiones se redirijan a la misma página, la de nuestra SPA. La configuración del servidor queda fuera del ámbito de estos apuntes.

3.5. Ejercicios de vistas

En los ejercicios de esta sesión vamos a implementar un interfaz rudimentario para la aplicación de alquiler de coches. Por tanto seguirás trabajando sobre la misma carpeta `alquiler_coches`.

Formulario para dar coches de alta (0,5 puntos)

Implementa un formulario que sea una vista de Backbone y que permita dar de alta un nuevo vehículo. - El modelo asociado a la vista será una instancia de `Coche` - Puedes crear una plantilla Mustache para la vista, pero ten en cuenta que será totalmente estática (solo HTML), ya que no sirve para mostrar datos, sino para introducirlos. - La vista debería escuchar el evento `sync` sobre el modelo, que indicará que el alta en el servidor se ha producido correctamente. En ese caso se debería mostrar un mensaje indicándolo (muéstralo en algún lugar del HTML, y pon algún botón o enlace para eliminar luego el mensaje. Recuerda que es más sencillo con jQuery). Escucha también el evento `error` que indica un error, y muestra también un mensaje adecuado en el HTML.

Listado de coches (0,5 puntos)

Implementa una vista de Backbone que muestre un listado con todos los coches. El listado será totalmente estático salvo por el hecho de que cuando se inserte un nuevo coche en el formulario de alta debe aparecer también aquí. Para ello:

- Cuando se dé de alta el modelo, debes añadirlo también a la colección.
- La vista de listado debe escuchar el evento `add` sobre la colección y si se produce, añadir al HTML ya existente el del nuevo coche.



también se podría redibujar la lista entera, pero eso no sería demasiado eficiente, así que no lo hagas de ese modo.

4. Jerarquías de vistas

Hasta ahora todos los ejemplos de Backbone que hemos tratado contenían una única vista. Es fácil ver que en aplicaciones reales esto no va a ser así: la interfaz de una SPA está compuesta de una serie de secciones diferenciadas, y es lógico pensar que cada una de ellas podemos modelarla como una vista de Backbone. De hecho, probablemente haya vistas compuestas a su vez de otras vistas más pequeñas, o vistas "hijas". Vamos a tratar aquí diferentes patrones de organización de vistas en web, y cómo podemos tratarlas de modo eficiente en Backbone. Veremos que como Backbone "se queda corto" cuando las vistas alcanzan una cierta complejidad los desarrolladores suelen usar extensiones de Backbone como [MarionetteJS](http://marionettejs.com/)⁴⁴.

4.1. Listados dinámicos

Este patrón se da cuando tenemos que mostrar un listado de elementos dinámicos, sobre los que se puede realizar una serie de operaciones (ver detalles, editar, borrar...). Es una situación muy común en multitud de aplicaciones web.

La organización más habitual de un listado de este tipo en Backbone es como una vista que engloba un conjunto de *subvistas*, una por cada elemento del listado:

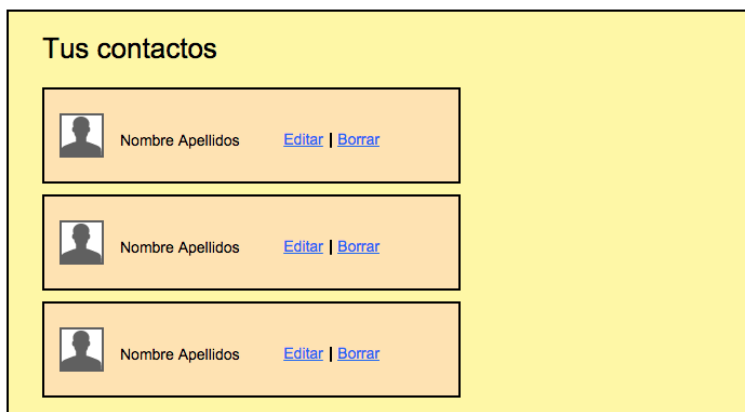


Figura 4. Vista Backbone compuesta a su vez de subvistas

La vista global, además de servir como "contenedor", se encarga de algunas operaciones que no son propias de ningún elemento en concreto. En el ejemplo anterior se podría encargarse de crear un nuevo contacto si le añadiéramos el correspondiente formulario.

Hay algunas razones por las que esta organización es adecuada:

- Representar cada elemento del listado como una subvista nos ayuda a "componentizar" y organizar mejor el código.
- Podremos gestionar de modo más sencillo los eventos para editar, borrar, ... Al estar cada elemento de la lista en una subvista diferente, cada vista se tiene que responsabilizar únicamente de procesar sus propios eventos. Esto simplifica el código.

Vamos a ver cómo implementaríamos la lista de contactos que vemos en la figura anterior. Para simplificar, las únicas operaciones que podemos realizar son crear un nuevo contacto y borrar un contacto. La primera de ellas es de tipo global y corresponde a la vista principal. La segunda corresponde a cada subvista por separado.

⁴⁴ <http://marionettejs.com/>

Subvistas en Backbone

Cada subvista será una instancia de la clase `VistaContacto`, y será responsable de *renderizarse* "ella misma" y procesar sus eventos. El código Javascript sería algo como lo que sigue:

```

var VistaContacto = Backbone.View.extend({

  //queremos que la etiqueta de la que "cuelga" la vista tenga la
  class="contacto"
  //Así podremos darle un estilo apropiado con CSS
  className: 'contacto',

  //plantilla Mustache
  template: $('#contacto_tmpl').html(),

  render: function() {
    //Usamos el toJSON() de Backbone en vez del stringify estándar
    this.el.innerHTML =
    Mustache.render(this.template, this.model.toJSON())
    return this
  },

  borrar: function() {
    this.model.destroy()
    this.remove()
  },

  //Cada contacto tiene su propio botón de borrar
  events: {
    'click .boton_borrar' : 'borrar'
  }
})

```

La plantilla Mustache asociada sería la siguiente:

```

<script id="contacto_tmpl" type="text/x-mustache-template">
  <b>{{nombre}} {{apellidos}}</b> <br>
  <em>{{telefono}}</em> <br>
  <input type="button" class="boton_borrar" value="Borrar">
</script>

```

Esta vista no tiene nada sustancialmente diferente de las que hemos usado hasta el momento. La clave, pues, está en la vista global.

La vista global

La plantilla Mustache asociada a la vista no es excesivamente interesante, únicamente contiene la parte "externa" a la lista de contactos en sí: un título y un formulario para dar de alta un nuevo contacto:

```

<script id="listado_tmpl" type="text/x-handlebars-template">

```

```

<h1>Lista de contactos</h1>
<label for="nombre_edit">Nombre:</label>
<input type="text" id="nombre"> <br>
<label for="apellidos_edit">Apellidos:</label>
<input type="text" id="apellidos"> <br>
<label for="telefono_edit">Teléfono:</label>
<input type="text" id="telefono"> <br>
<input type="button" id="boton_nuevo" value="Nuevo">
</script>

```

La parte realmente interesante es el código del método `render()`. Además de renderizar la plantilla asociada, debe ir creando una subvista por cada elemento del listado, haciendo el `render()` de dicha subvista y añadiendo el resultado al HTML propio.

```

render: function() {
  this.$el.html(this.template) ❶
  this.collection.each(this.renderContacto) ❷
  return this
},

renderContacto: function(contacto) { ❸
  var vc = new VistaContacto({model: contacto}) ❹
  this.$el.append(vc.render().$el) ❺
},

```

- ❶ Lo primero que hacemos es *renderizar* la plantilla global propiamente dicha
- ❷ Usando el iterador `each` de Underscore, iteramos por la colección de contactos, y para cada uno de ellos llamamos a la función `renderContacto`
- ❸ Esta función se encarga de *renderizar* el contacto. El `each` hace que automáticamente reciba como parámetro el objeto correspondiente a la iteración actual.
- ❹ Creamos la subvista asociada al contacto
- ❺ Renderizamos la subvista y la añadimos al HTML de la vista global



Este código es lo que los anglosajones llaman *boilerplate*, lo vamos a encontrar de manera casi literal en muchos proyectos de Backbone. Hasta tal punto es típico, que como veremos a continuación algunos *frameworks* basados en Backbone (como Marionette) lo incorporan automáticamente, para que no haya que escribirlo "a mano".

El código anterior tiene un pequeño problema. Ya hemos visto alguna vez que **cuando desde una función de una clase de Backbone llamamos a otra, `this` no tiene como valor el objeto actual, sino el objeto global (window)** Para resolverlo, en la inicialización de la vista enlazamos (*bind*) la función `renderContacto` con la vista:

```

initialize: function() {
  _.bindAll(this, "renderContacto")
}

```

En el código Backbone típico se suele usar el `_.bindAll` de Underscore para vincular una función con un objeto, más que nada porque es cómodo y nos asegura el soporte en navegadores antiguos. También podríamos haber usado el `bind` de Javascript estándar:

```
initialize: function() {
  this.renderContacto = this.renderContacto.bind(this)
}
```

Subvistas con Marionette

Marionette es una extensión de Backbone. Le añade una serie de funcionalidades interesantes para el trabajo cotidiano con el *framework*. Por ejemplo ya hemos visto que Backbone "aporta poco" en cuanto a la gestión de las vistas, prácticamente lo tenemos que hacer todo nosotros. Marionette automatiza mucho más el trabajo con las vistas, implementando un `render` que a diferencia del de Backbone sí hace algo por defecto: serializa el modelo en JSON, aplica la plantilla, Además incluye clases pensadas para representar explícitamente jerarquías de vistas. Enseguida veremos una pequeña introducción a estas funcionalidades. Por otro lado Marionette también incluye funcionalidades no relativas a vistas como por ejemplo la definición de módulos, que nos permiten organizar mejor nuestro código, o la ampliación del sistema de eventos de Backbone.



En lugar de dedicar una sesión entera a Marionette vamos a ir viendo sus características poco a poco. Conforme vayamos explicando funcionalidades de Backbone iremos viendo en qué se "quedan cortas" y cómo nos puede facilitar Marionette el trabajo. Podéis consultar la documentación de Marionette y ver algunos tutoriales y *screencasts* interesantes en su [sitio web](#)⁴⁵.

Marionette tiene dos tipos de vista pensados para resolver nuestro problema. La clase `ItemView` nos sirve para representar un único modelo, mientras que `CollectionView` representa una colección. Un `CollectionView` tendrá un conjunto de `ItemView` como vistas "hijas".

Continuando con el ejemplo del apartado anterior, para representar cada contacto usaríamos un `ItemView`. Marionette implementa un `render` por defecto que:

- Serializa automáticamente el modelo asociado a la vista.
- Aplica la plantilla, que debe estar asociada a una propiedad `template` de la vista.
- Actualiza el `el` de la vista con el resultado.

```
Marionette.Renderer.render = function(template,data) { ❶
  return Mustache.render(template,data);
}
```

```
var contacto_tmpl = ❷
  '<b> {{apellidos}}, {{nombre}} </b> <br>' +
  '<em>{{telefono}}</em>'
```

```
var VistaContacto = Marionette.ItemView.extend({ ❸
  template: contacto_tmpl
})
```

```
//Vamos a probar cómo funciona
```

⁴⁵ <http://marionettejs.com>

```

var c1 = new Contacto({nombre:"Pepe", apellidos:"Pérez Martínez",
  telefono:"966123456"}); ❷
var vc = new VistaContacto({model:c1});
$('body').append(vc.render().$el); ❸

```

- ❶ Por defecto Marionette está configurado para usar *templates* de Underscore. Podemos configurarlo para usar otro motor de plantillas sobrescribiendo el método `Marionette.Renderer.render`. Este método acepta como parámetros una plantilla y unos datos y debe devolver el resultado de combinar ambos. En el ejemplo, lo configuramos para usar Mustache.
- ❷ Definimos la plantilla de Mustache para mostrar un contacto, no hay diferencia con Backbone.
- ❸ Definimos la clase de la vista, que hereda de la clase `ItemView`. Le asignamos a la propiedad `template`, propia de Marionette, la plantilla asociada. Nótese que no tenemos que implementar el `render` ya que Marionette lo hace por nosotros.
- ❹ Definimos una instancia de un modelo y de una vista, asociada al modelo.
- ❺ Añadimos el HTML de la vista al cuerpo de la página, como vemos es idéntico a como se hace en Backbone.

Una vez hemos definido la vista para un item de la lista, ya podemos definir la vista "global" para la lista de elementos. En Marionette para esto se usa un `CollectionView`, que incluirá como vistas "hijas" varias `ItemView`. Siguiendo con nuestro ejemplo, para presentar la lista de contactos: (mostramos solo lo que hay que añadir al código anterior)

```

var VistaAgenda = Marionette.CollectionView.extend({
  childView: VistaContacto,
});

//Vamos a probar cómo funciona
var c1 = new Contacto({nombre:"Pepe", ...
var c2 = new Contacto(...
var miAgenda = new Agenda([c1,c2]);
var va = new VistaAgenda({collection: miAgenda})
$('body').append(va.render().$el);

```

Como vemos, lo único que necesitamos para definir la vista asociada a la colección es especificar qué clase va a actuar como vista para cada elemento. En nuestro caso es la clase `VistaContacto`. El `render` de la `CollectionView` creará automáticamente una `VistaContacto` por cada elemento de la colección si es necesario, llamará a su `render` y concatenará todos los HTML resultantes, es decir, lo mismo que antes teníamos que hacer de modo manual.

Nótese que una `CollectionView` no tiene HTML "propio", su HTML es el formado por la concatenación del de sus vistas hijas. Si queremos que la vista "madre" contenga información propia deberíamos usar una `CompositeView`, que es como una `CollectionView` pero se le puede asociar también un `model` y una `template`.

Marionette hace un *re-render* automático cada vez que se modifica la colección asociada a una `CollectionView`. Si nos vamos a la consola Javascript y tecleamos

```

va.collection.add(new Contacto({nombre:"Luis Ricardo",
  apellidos: "Borriquero", telefono:"965656565"}))

```

veremos cómo se redibuja automáticamente la vista y aparece el nuevo contacto. Es decir, el `CollectionView` implementa un *data binding* unidireccional, del modelo hacia la vista.

4.2. Composición genérica de vistas

En una aplicación web es muy común dividir la página en diferentes secciones. Estas secciones se suelen representar en el HTML con etiquetas `<div>` (o `<section>`, `<nav>`, `<article>`, ... si usamos HTML5) y se marcan con distintas clases o identificadores. Así se les puede dar estilo con CSS y pueden ser manipuladas dinámicamente con Javascript. Por ejemplo aquí tenemos la típica página con contenido, pie y barra lateral:

```
<!-- Falta el CSS que haga aparecer las cosas "en su sitio" -->
<div id="sidebar">
<div>Esto es la barra lateral</div>
</div>
<div id="main">
<div>Esto es el contenido principal</div>
</div>
<div id="footer">
<div>Y esto es teóricamente el pie</div>
</div>
```

En Backbone podemos crear una vista por cada sección, pero el *framework* no nos da ninguna facilidad para coordinar las vistas entre sí ni estructurarlas si necesitamos que a su vez una sección se componga de subsecciones. Para trabajar con este tipo de estructuras de manera más sencilla podemos usar Marionette.



si necesitamos trabajar con jerarquías de vistas pero no queremos usar Marionette porque no nos hacen falta sus otras funcionalidades, una alternativa más "ligera" es un *plugin* de Backbone muy conocido llamado [Layout Manager](#)⁴⁶.

Composición de secciones con Marionette

En Marionette podemos definir una vista que sea una composición de otras extendiendo la clase `LayoutView`. Esta clase tiene una propiedad `regions` en la que daremos una lista de las secciones (o como las llama Marionette, *regiones*) que componen la vista. Para cada sección especificamos un nombre simbólico y un selector que identifique la región dentro de la página.

```
var VistaGlobal = Marionette.LayoutView.extend({
  el: 'body',
  regions: {
    barra: '#sidebar',
    principal: '#main',
    pie: '#footer'
  }
});
var laVistaGlobal = new VistaGlobal();
```

⁴⁶ <https://github.com/tbranyen/backbone.layoutmanager>

Una vez creada la instancia de `LayoutView` podemos acceder a sus regiones por identificador y mostrar una vista en cada una de ellas con `show`. Podemos eliminar la vista con `empty`.



las regiones son zonas más o menos permanentes de la página, mientras que las vistas habitualmente se mostrarán y eliminarán de manera dinámica con `show` y `empty`.

```
//Suponemos que "MiVista" y "miModelo" ya están definidos
var unaVista = new MiVista({model:miModelo});
//mostramos una vista
laVistaGlobal.getRegion('principal').show(unaVista);
//La ocultamos a los dos segundos
setTimeout(2000, function() {
  laVistaGlobal.getRegion('principal').empty();
})
```

Secciones anidadas con Marionette

Para anidar vistas dentro de otras podemos hacer que la vista mostrada en una región sea a su vez una `LayoutView`. Supongamos que ahora queremos que la región principal se divida a su vez en un título y en otra subregión para el texto del cuerpo principal.

Vamos a usar plantillas para modularizar las regiones:

```
<script id="global_tmpl" type="text/x-template">
  <div id="sidebar">
    Esto es la barra de navegación
  </div>
  <div id="main">
  </div>
  <div id="footer">
  </div>
</script>

<script id="principal_tmpl" type="text/X-template">
  <div><h1>Esto es el título del contenido principal</h1></div>
  <div id="main_content">
  </div>
</script>

<script id="texto_principal_tmpl" type="text/X-template">
  Esto es el texto del contenido principal
</script>
```

Ahora el código sería algo como lo que sigue:

```
var miApp = new Marionette.Application(); ❶

miApp.addRegions({
  todo: "#all"
});
```

```
$(document).ready(function() {
  var vg = new VistaGlobal();
  miApp.getRegion('todo').show(new VistaGlobal());
});

var VistaGlobal = Mn.LayoutView.extend({ ❷
  template: '#global_tmpl',
  regions: {
    barra: '#sidebar',
    principal: '#main',
    pie: '#footer'
  },
  onBeforeShow: function() {
    this.getRegion('principal').show(new VistaPrincipal());
  }
});

var VistaPrincipal = Mn.LayoutView.extend({
  template: '#principal_tmpl',
  regions: {
    textoprincipal: '#main_content'
  },
  onBeforeShow: function() {
    this.getRegion('textoprincipal').show(new VistaTextoPrincipal());
  }
});

var VistaTextoPrincipal = Mn.ItemView.extend({
  template: '#texto_principal_tmpl'
});
```

-
- ❶ Creamos un objeto `Marionette.Application` al que podemos añadir regiones. Tenemos una única region que engloba la aplicación y que está dividida en subregiones (con vistas asociadas).
 - ❷ La `VistaGlobal` tiene tres regiones, de las cuales solo la llamada `principal` es dinámica. Igual que antes usamos el `onBeforeShow` de la región "madre" para pintar la "hija". Con la única diferencia de que ahora la "hija" a su vez es una `LayoutView` compuesta de subregiones.

4.3. Ejercicios de jerarquías de vistas

En esta sesión vamos a mejorar la interfaz implementada en la sesión anterior para permitir que se eliminen coches. Para ello vamos a crear una subvista por cada coche del listado, primero con Backbone y luego usando Marionette.

Vistas y subvistas con Backbone (0,5 puntos)

Para este ejercicio haz una copia de la carpeta `alquiler_coches` que ya tenías, y llámala `alquiler_coches_subvistas`.

Usando solamente Backbone, sin Marionette, cambia el listado "estático" que ya tenías y que usaba una sola vista por un listado dinámico en el que se puedan eliminar coches, y cada coche se muestre con una subvista. Cada coche debería aparecer con un botón o enlace "eliminar" que:

1. borre el modelo del servidor⁴⁷ (`destroy()`)
2. Si se elimina correctamente del servidor, elimine la vista⁴⁸ de la página (`remove()` de `View`).
3. Quite el modelo⁴⁹ de la colección (`remove()` de `Collection`)

Vistas y subvistas con Marionette (0,5 puntos)

Para este ejercicio deberás coger tu código JS (solo los modelos, no las vistas) y juntarlo con la plantilla de aplicación Marionette que tienes en las plantillas de la asignatura. Entrega el ejercicio en una carpeta `alquiler_coches_marionette`.

Usando Marionette implementar las mismas funcionalidades del ejercicio anterior. Ten en cuenta que como la vista global debe mostrar el formulario de alta no te valdrá con una `CollectionView` sino que debes usar una `CompositeView` que sí puede tener una `template` asociada.

⁴⁷ <http://backbonejs.org/#Model-destroy>

⁴⁸ <http://backbonejs.org/#View-remove>

⁴⁹ <http://backbonejs.org/#Collection-remove>

5. Miniproyecto de aplicación con Backbone y Marionette

En esta sesión vamos a desarrollar una pequeña aplicación algo más compleja que los ejercicios de las otras sesiones. El objetivo es trabajar con varias vistas de manera coordinada y algo más realista que lo que hemos hecho hasta ahora.

5.1. Requerimientos

Queremos desarrollar una pequeña aplicación de gestión de comics en la que podamos buscar comics de Marvel usando [su API REST](#)⁵⁰. Si estamos autenticados también podremos marcar comics como favoritos, y gestionar luego esta lista de favoritos.

Como requerimientos "iniciales":

- Se podrán buscar comics por título, listando los resultados de modo resumido
- Se podrán ver todos los detalles de un comic determinado

Como requerimientos "adicionales"

Se usará Kinvey como *backend* para realizar las siguientes operaciones:

- Se podrá hacer *login* y *logout*
- Si estamos logueados, se podrán marcar comics como favoritos, añadiéndolos a una lista de "mis comics".
- Se podrán eliminar comics de la lista "mis comics".

Fíjate que la aplicación trabaja con dos *backends*. El servidor de Marvel solo nos permite leer datos. En él buscaremos comics y obtendremos sus datos. El servidor de Kinvey nos permite también guardar información, así que ahí es donde guardaremos los datos de nuestros comics favoritos.

Daremos aquí instrucciones paso a paso de cómo implementar los requerimientos "iniciales", pero solo algunas guías genéricas de cómo implementar los "adicionales".



Para hacer peticiones al API de Marvel hay que tener un "API key". En la plantilla del proyecto se usa la API Key del profesor (tiene unas 3000 llamadas diarias de límite, seguramente más que suficientes). No obstante, si deseas usar tu propia API key, puedes [darte de alta como desarrollador](#)⁵¹ en la web de Marvel para obtenerla, accediendo luego a tu panel de control de desarrollador.

5.2. Implementación de los requerimientos "iniciales"

Aquí tienes la [plantilla de proyecto](#)⁵² que puedes usar como base para tu código.

La "lógica de negocio"



para que funcionen las llamadas al API de Marvel con la clave del profesor, la página debe ser servida por un servidor web que funcione en localhost (si abris el `.html` dando doble clic y probáis una llamada al API dará error HTTP 409). Podéis poner en marcha un servidor web

⁵⁰ <http://developer.marvel.com/>

⁵¹ <https://developer.marvel.com/pleasesignin>

⁵² <https://bitbucket.org/ottocol/marvel-miscomics/downloads>

simple yendo al directorio donde está la aplicación y ejecutando `python -m SimpleHTTPServer`, que empezará a funcionar por `http://localhost:8000`. También podéis instalar un servidor basado en Node con `sudo npm install -g http-server` y luego ejecutarlo con `http-server`.

La plantilla usa espacios de nombres y módulos al estilo "patrón módulo". Tienes ya implementado un modelo (`Comic`) y una colección (`Comics`). Esta última ya implementa la comunicación con el API de Marvel, para buscar comics por título. Puedes ir a la consola Javascript y teclear:

```
lista = new Marvel.Collections.Comics()
lista.buscar("Hulk")
```

Pasados unos segundos si examinas la variable "lista" debería contener un conjunto de modelos `Comic` cada uno con los datos de un comic. Aunque el API está paginado y podríamos irle solicitando resultados de 20 en 20 como mucho, para simplificar la aplicación nos "conformaremos" con listar solo los 20 primeros (podría haber menos), no es necesario implementar paginado.



Vamos a usar Marionette para la interfaz porque simplifica bastante el renderizado y la gestión dinámica de las vistas, pero también se podría implementar con Backbone "puro". Puedes hacerlo así si eres lo suficientemente masoquista.

Estructura de la interfaz

Dividiremos la pantalla en tres secciones, representadas con tres `div` en el `index.html`

- Parte superior ("cabecera"): aquí aparecerá el formulario de login, y si ya estamos autenticados, un botón o enlace para gestionar "mis comics".
- Parte media ("formBusqueda"): el formulario de búsqueda
- Parte inferior ("listado"): Esta parte irá cambiando. Si hemos buscado mostrará la lista de resultados de búsqueda. Si hemos elegido "mis comics" aparecerán los que hemos ido seleccionando. Si pulsamos sobre "ver detalles" de un comic se verá únicamente el cómic seleccionado.



aunque hablemos de parte superior, media e inferior, la colocación en pantalla es libre. Con el CSS adecuado podrías poner por ejemplo la página a tres columnas y colocar en cada columna una sección. Pero sí deberías respetar las tres secciones y el papel que desempeña cada una de ellas. Por otro lado, dale prioridad a la funcionalidad sobre la interfaz, no te preocupes demasiado del aspecto estético, no es el objetivo del ejercicio.

Estas tres secciones estarán controladas por una "vista global" de Marionette, que gestionará a las vistas "hijas". Pero vamos por partes. Primero vamos a ver y probar las vistas básicas, para ver un solo comic y un listado de comics.

Vista de un solo comic: `js/views/VistaComic.js`

La clase ya la tienes definida, ya que en Marionette solo hace falta referenciar dónde está la *template*, el *render* lo implementa Marionette. Eso sí, tendrás que definir tú la *template* usando

Mustache en el `script id="VistaComicTpl"`, que ahora está vacío. Muestra al menos el `title` y la `description` del comic. Además muestra una imagen a tamaño reducido. Las imágenes se obtienen concatenando:

- Una URL (propiedad `thumbnail.path`)
- Un tipo de imagen (que puede ser "standard_small", "standard_medium", "standard_large", ... [consulta aquí](#)⁵³ todos los tipos o *image variants* y escoge el que prefieras).
- Una extensión de archivo (normalmente `.jpg` pero puede variar, así que se usa el atributo `thumbnail.extension` para ser más genérico).

Para comprobar que funciona, desde la consola de Javascript lanza una búsqueda de comics como has hecho antes, asegúrate de que ya ha respondido el servidor (la colección no está vacía) y luego crea una vista pasándole una posición de la colección y renderízala en la página. Algo como:

```
lista = new Marvel.Collections.Comics()
lista.buscar("Hulk")
...espera unos segundos, asegúrate de que "lista" contiene datos
v = new Marvel.Views.VistaComic({model:lista.at(0)});
v.render().$el.appendTo('body');
```

Deberían aparecer en pantalla los datos del comic.

Vista de lista de comics

Esta vista es del tipo `CollectionView`, lo que quiere decir que no tiene HTML propio, su HTML es solo una concatenación de los HTML de las vistas "hijas" (del tipo `VistaComic`). Por tanto no tiene *template*.



Si quisieras que esta vista tuviera HTML propio y por tanto *template* (por ejemplo para mostrar un título "Lista de resultados") tendrías que cambiar el tipo por `CompositeView`.

No obstante para asegurarte de que todo es correcto puedes hacer una prueba similar a la que has hecho para `VistaComic`, pero ahora para mostrar un listado completo

```
lista = new Marvel.Collections.Comics()
lista.buscar("Hulk")
...espera unos segundos, asegúrate de que "lista" contiene datos
v = new Marvel.Views.VistaComics({collection:lista});
v.render().$el.appendTo('body');
```

Vale, ya estamos seguros de que al menos las piezas básicas con los datos de los comics funcionan. Vamos con las piezas de "alto nivel".

Vista Global

Tienes el esqueleto en `js/views/VistaGlobal.js`. Es una vista de tipo `LayoutView`, o sea compuesta. El esqueleto únicamente especifica las secciones que controla (con el objeto

⁵³ <http://developer.marvel.com/documentation/images>

`regions`, que establece la correspondencia entre nombres simbólicos de secciones y nodos del HTML).

Fijate que en el archivo `js/main.js`, cuando se carga el documento (evento `$(document).ready`) se crea una instancia de la vista global y se muestra una vista con el formulario de búsqueda en la sección `formBusqueda`. Pero por ahora el formulario no hace nada. Vamos a solucionar esto enseguida.

Vista de búsqueda (0,5 puntos)

Esta vista debe ser la encargada de mostrar el formulario, disparar la búsqueda y obtener los resultados, que le pasará a la vista global. Por ahora solo muestra el formulario.

Está representada en el código por:

- Una clase `Marvel.Views.VistaBuscarComics`, cuyo esqueleto básico ya tienes implementado en `js/views/VistaBuscarComics.js`
- Una *template* que tienes en `index.html` en forma de `script` con un `id=VistaBuscarComicsTpl`, con un formulario básico (puedes modificarlo si lo deseas).

Para que esta vista haga su trabajo completo, tienes que:

- Definir el array `events` para que cuando se pulse sobre el botón con `id=botonBuscar` se llame a una función `buscar` de la vista, que debes definir.
- En esta función `buscar` debes llamar al método `buscar` de la colección (el que has probado antes en la consola).
- En el `initialize` debes crear la colección de comics asociada, por el momento vacía (se llenará de datos cuando se haga el `fetch`)

```
initialize: function() {
  this.collection = new Marvel.Collections.Comics();
  //Todavía faltan cosas
  ...
}
```

- Como la búsqueda es asíncrona, para saber cuándo se han recibido resultados usaremos el evento `sync` de la colección que indica que se ha recibido del servidor. En el `initialize` debes hacer que la vista escuche el evento `sync` sobre la colección y que cuando se produzca llame a una función de la vista `busquedaCompletada`.



recuerda que cuando un evento llama a un *callback*, `this` no apunta a la vista sino a `window`. Para solucionarlo puedes hacerlo de dos formas

```
//Forma 1: usando el bindAll de la librería underscore
_.bindAll(this, 'busquedaCompletada');
this.listenTo(this.collection, 'sync', this.busquedaCompletada);
//Forma 2 (USA SOLO UNA DE ELLAS!!): usando el "bind" estándar de
  Javascript
this.listenTo(this.collection, 'sync', this.busquedaCompletada.bind(this));
```

- Finalmente define la función `busquedaCompletada` para que lance un evento "a medida" (es decir, no estándar de Backbone) y que escuchará la vista global, luego veremos cómo.

```
busquedaCompletada: function() {
  //El nombre `completed:search` es totalmente inventado, podrías poner
  lo que quieras.
  //this.collection se le pasará como parámetro a quien esté escuchando
  este evento
  this.triggerMethod('completed:search', this.collection);
},
```



Sí, es un poco retorcido esto de capturar el evento `sync` para lanzar un evento `completed:search`. En Marionette una vista global puede escuchar eventos de las vistas hijas, pero no directamente de una colección gestionada por una vista hija. También podrías pasarle a la vista global una referencia a la colección para que pudiera escuchar directamente el evento `sync`. Pero es un poco embrollado que las vistas se pongan a escuchar eventos sobre objetos que en principio no son suyos.

De nuevo a la vista global (0,25)

Ahora para que la **vista global** escuche el evento `completed:search` y en respuesta muestre una vista con la lista de comics encontrados puedes añadirle lo siguiente:

```
childEvents: {
  //los eventos de las subvistas automáticamente reciben como 1er
  parámetro la subvista
  //y luego los que hayamos incluido cuando generamos el evento con el
  triggerMethod
  'completed:search' : function(child, col) {
    //Creamos una vista para mostrar una lista de comics, le pasamos la
    colección
    //y la mostramos en la sección "listado" de la vista global
    this.showChildView('listado', new Marvel.Views.VistaComics({
      collection: col
    })))
  }
}
```

Ver detalles de comic (0,25)

Nuestro objetivo ahora es que cuando pulsemos en "ver detalles" de un comic se sustituya la lista de comics por los datos detallados del comic elegido. Luego al "cerrar detalles" aparecerá de nuevo la lista.

- Lo primero, tendrás que modificar la *template* de la vista que solo muestra un comic `VistaComic` para añadirle un enlace o botón "ver detalles".
- Después usa la propiedad `events` de la vista para asociar el click sobre el enlace o botón con una función `verDetalles`



En la función `verDetalles`, lo primero asegúrate de que anulas el comportamiento por defecto del enlace o botón, ya que podría recargar la página y se comportaría de forma "extraña":

```
//Los manejadores de evento Javascript reciben automáticamente el evento
producido
function verDetalles(evento) {
  //Anular el manejador por defecto del navegador, que podría recargar la
  página
  evento.preventDefault();
  ...
}
```

- Además de lo anterior, en la función `verDetalles` debes generar un evento "a medida" que llegará a la vista global y le indicará que hay que mostrar los detalles de un comic concreto. Pasa algo parecido a la búsqueda. En este caso una vista "madre" no puede escuchar directamente los eventos de interfaz de usuario de las hijas, así que las hijas tienen que capturarlos ellas mismas y lanzar un nuevo evento para la "madre". Finalmente `verDetalles` quedará:

```
//Los manejadores de evento Javascript reciben automáticamente el evento
producido
function verDetalles(evento) {
  //Anular el manejador por defecto del navegador, que podría recargar la
  página
  evento.preventDefault();
  //me invento un evento (¡y rima!). Pasamos el modelo, para que la vista
  "madre" sepa
  //de quién hay que mostrar los detalles
  this.triggerMethod('show:details', this.model);
}}
```

Ahora tendremos que modificar el código de la vista global, que es la que tiene que recibir el evento `show:details`. En respuesta a este evento, sustituiremos la vista con la lista de comics por una vista para mostrar los detalles (clase `VistaDetallesComic`). Pero como cuando cerremos los detalles queremos volver a la lista, le diremos a Marionette que no destruya la vista de lista, y la guardaremos en el objeto vista global.

```
childEvents: {
  'completed:search' : function(child, col) {
    //...esto ya lo teníamos de antes
  },
  'show:details': function(child, model) {
    //guardamos la vista con el listado actual
    this.vistaLista = this.getRegion('listado').currentView;
    //Creamos una vista de tipo "detalle" asociada al modelo
    var nv = new Marvel.Views.VistaDetallesComic({model: model});
    //mostramos la nueva vista, diciéndole a Marionette que no libere la
    memoria
    //de la anterior, ya que luego la colocaremos otra vez en su sitio
    this.getRegion('listado').show(nv, {preventDestroy:true});
  }
}
```

```
}

```

Cuidado, si quieres probar esto, lo primero que debes hacer es rellenar la *template* asociada a la `VistaDetallesComic`, para que aparezca información en pantalla. Coloca los datos que quieras, y una imagen a mayor tamaño que la que aparece en el listado.

Cerrar la vista de detalles y volver al listado de comics (0,25 puntos)

En la *template* asociada a la clase `VistaDetallesComic` tiene que haber un botón o enlace "Cerrar". Ahora es similar al proceso seguido para mostrar los detalles. En la clase `VistaDetallesComic`

- Mediante el `events` debes asociar el click sobre "Cerrar" con una función de la vista `cerrarDetalles`
- En la función `cerrarDetalles` debes generar un evento "a medida" para que lo capture la vista "madre". Por similitud con el anterior, puedes llamarlo "hide:details" (o como quieras).



recuerda llamar a `preventDefault()` para evitar posibles recargas de la página.

Finalmente, modifica el código de la vista global para que reciba el evento `hide:details` (o como lo hayas llamado) y vuelva a poner "en su sitio" la lista de comics.

```
childEvents: {
  ...
  ...
  'hide:details': function() {
    this.getRegion('listado').show(this.vistaLista);
  }
}
```

5.3. Requerimientos "adicionales" (1 punto en total)

Con la guía anterior espero que hayas adquirido una idea básica de cómo coordinar varias vistas desde una vista "madre", que es la parte más complicada. Los requerimientos que faltan necesitan de la gestión de vistas y además de la interacción con el *backend* de Kinvey.

Pistas de implementación para los requerimientos que faltan:

Para el "login": (0,3 puntos)

Tendrás que definir una vista `VistaFormLogin` o similar que muestre un formulario de login. Tómate la operación de login simplemente como una "comprobación de que el login y el password son correctos". En realidad no usamos una sesión de usuario ya que recuerda que en cada petición a Kinvey mandamos de nuevo login y password, al estar usando HTTP Basic.

Para [hacer login en Kinvey](http://devcenter.kinvey.com/rest/guides/users#login)⁵⁴ con el API REST hay que hacer una petición POST a `https://baas.kinvey.com/user/MI_APP_ID/login`, mandando un objeto JSON con

⁵⁴ <http://devcenter.kinvey.com/rest/guides/users#login>

las propiedades 'username' y 'password'. La petición debe estar autenticada usando HTTP BASIC con las credenciales de la aplicación, es decir, como login el APP_ID y como password el APP_SECRET, que aparecen en la parte superior del *dashboard* web de Kinvey.



puede haber cierta confusión entre las credenciales (login+password) del usuario y las credenciales de la aplicación. Para la operación de login en Kinvey debemos enviar un objeto JSON con las credenciales del usuario. Pero es que además TODAS las peticiones a Kinvey deben estar autenticadas. ¿Pero con qué credenciales, si precisamente estamos intentando hacer login ahora mismo?. En Kinvey esto se resuelve usando las credenciales de la aplicación (APP ID+APP SECRET), que viene a ser una especie de "usuario global" de la app.

Crea un modelo Backbone llamado `Usuario`, con las propiedades `username` y `password`, y define en él un método `login` que llame internamente al `save()` de Backbone. Cuando el login se efectúe con éxito (evento `sync` sobre el modelo), la vista de la barra superior debería cambiar para mostrar tu login y un botón/enlace para ver "mis comics".

Aunque en Kinvey existe una operación de *logout*, en nuestro caso no tiene sentido, porque no estamos usando sesiones. Simplemente cuando se haga *logout* tu aplicación debería dejar de mostrar los datos del usuario actual y volver a mostrar el formulario de login.

Para la gestión de "mis comics" (0,7 puntos): La idea sería que usaras un modelo `Favorito` en la que almacenaras los datos que quieras guardar del comic (como mínimo su `id`, para poder recuperar el resto de datos con el API de Marvel, o bien copiar los campos y repetirlos en Kinvey para no tener que buscar de nuevo en Marvel). Tendrás que implementar:

- La parte de interfaz que te permite marcar un comic como favorito
- La sustitución de los resultados de búsqueda por la lista de tus comics cuando pulsas en "mis comics"

Como es lógico también deberían poderse eliminar los favoritos, pero ya son bastantes requerimientos, lo dejaremos fuera.

Puedes implementar estas funcionalidades de la forma que mejor te parezca, mientras funcionen correctamente. ¡Suerte!.

6. Interfaces web con ReactJS

6.1. ¿Por qué ReactJS?

Una de las partes más tediosas de una SPA es actualizar la interfaz gráfica de manera dinámica. Conforme cambia el modelo y las colecciones debemos estar constantemente redibujando la interfaz para reflejarlo. Si además tenemos en cuenta que Backbone no ofrece *data binding* de manera nativa es fácil ver que va a ser una labor que consuma bastante tiempo de desarrollo si se quiere hacer de forma eficiente, redibujando solo la parte que cambia.

Recordemos que el `render` de una vista en Backbone, por convenio genera **todo** el HTML de la vista. Si queremos redibujar solo parte de la interfaz tenemos que escribir otros métodos de *rendering* adecuados para la tarea en particular, como hacíamos en el primer ejemplo del *widget* del tiempo con el método `renderData`. El problema es que en una interfaz compleja acabaríamos con multitud de métodos `renderXXX` o solo con unos pocos pero que tendrían una lógica de control complicada.

La solución que propone ReactJS a este problema puede sorprender inicialmente por su aparente "ingenuidad": si es tan tedioso comprobar qué ha cambiado y redibujar solamente eso, ¿por qué no **redibujar siempre toda la interfaz**?. Así estaríamos seguros de que está correctamente actualizada. "Ya, pero eso debe ser muy ineficiente", habrás pensado inmediatamente. Pues resulta que no, porque una de las ideas clave de ReactJS es que *aunque el desarrollador se limita simplemente a forzar el redibujado completo, ReactJS calcula automáticamente qué es lo que cambia en la vista del estado actual al siguiente, y solo redibuja las partes necesarias*.

Una cosa que hay que tener clara es que **ReactJS es únicamente un framework para la vista**, no es un *framework* MVC completo. No tiene modelos ni tampoco controladores. En aplicaciones complejas adicionalmente a React se recomienda usar una arquitectura denominada **flux**, que veremos en las siguientes sesiones.

Un aspecto interesante de React es que está orientado al desarrollo de **componentes de interfaz**, que pueden ser fácilmente reutilizables. Además dichos componentes se pueden organizar de modo jerárquico para montar la interfaz completa de la aplicación uniendo componentes simples.

React es un proyecto *open source* creado originalmente por Facebook y usado en producción por la misma Facebook en su sitio web y también en sitios "hermanos" como Instagram. Además se usa en [muchos otros sitios web](#)⁵⁵ de tráfico elevado (Khan Academy, Codecademy, Atlassian,...).

6.2. ¡Hola React!. Nuestros primeros componentes

Lo primero es instalar la librería. Lo más sencillo es simplemente bajarla desde [su sitio web](#)⁵⁶. Como es habitual en este tipo de librerías viene en versión **minificada** y de desarrollo. Usaremos aquí esta última ya que muestra más avisos en la consola del navegador. Necesitaremos como mínimo los archivos `react.js` y `react-dom.js`.



La forma clásica de referenciar librerías Javascript usando `<script>` no es precisamente modular. Actualmente se tiende a usar herramientas

⁵⁵ <https://github.com/facebook/react/wiki/Sites-Using-React>

⁵⁶ <https://facebook.github.io/react/>

como [Webpack](#)⁵⁷ o [Browserify](#)⁵⁸ para referenciar librerías externas de forma más "elegante". Al ser React un *framework* bastante moderno, muchos tutoriales y libros de React tienden a usar estas herramientas, cosa que no haremos aquí para poder concentrarnos en aprender React, sin perdernos en tecnologías adicionales. Por el mismo motivo tampoco usaremos funcionalidades de EcmaScript versión 6 (ES6) en los ejemplos.

Los componentes de React son muy similares a las vistas de Backbone. Veamos primero cómo instanciaríamos un componente de los ya predefinidos en React. La librería incluye como componentes todas las etiquetas de HTML. Así por ejemplo si quisiéramos crear un encabezado `h1` que fuera un componente de React haríamos:

```
<script src="react.js"></script> ❶
<script src="react-dom.js"></script>
<div id="miApp"></div> ❷
<script>
  var miComponente = React.DOM.h1({id:"saludo"}, '¡Hola React!'); ❸
  ReactDOM.render(miComponente, document.getElementById('miApp')); ❹
</script>
```

- ❶ Incluimos la librería. Como ya hemos dicho como mínimo necesitamos estos dos archivos.
- ❷ Necesitamos un "sitio" en el HTML para colgar el componente React, al igual que ocurre con las vistas en Backbone. Lo más habitual es usar un `div` vacío.
- ❸ Creamos el componente. En este caso es simplemente uno de los predefinidos en React. En React tenemos toda una familia de métodos `React.DOM.*` que se corresponden con las etiquetas de HTML. Especificamos sus propiedades como un literal JSON y su contenido (al ser un `h1` será su texto).
- ❹ Finalmente pintamos el componente en el lugar destinado para ello.



es posible que en algún tutorial o ejemplo de React veas en lugar de `React.DOM.h1` código al estilo `React.createElement('h1', ...)`. La primera forma no es más que un método de conveniencia para evitar el método general, que es un poco más tedioso de usar.



En versiones anteriores de React se usaba `React.render` en lugar de `ReactDOM.render`. Por eso es posible que encuentres en Internet o en fuentes impresas muchos ejemplos que lo hacen así. No obstante en la versión actual la primera forma genera un *warning* en la consola del navegador, especificando que el método apropiado es el que hemos usado aquí.

En el mundo real es de esperar que nuestro componente sea algo más que una única etiqueta simple de HTML, normalmente contendrá otras en diferentes niveles, formando un fragmento de HTML. El API de `React.DOM.*` nos permite anidar etiquetas. Cuando una etiqueta admita dentro un conjunto de otras, en el API podremos pasar un array con los correspondientes componentes. Por ejemplo:

```
<script src="lib/react.js"></script>
<script src="lib/react-dom.js"></script>
<div id="miApp"></div>
```

⁵⁷ <https://webpack.github.io/>
⁵⁸ <http://browserify.org/>

```

<script>
  var miLista = React.DOM.ul(null,
    React.DOM.li(null, 'Pan'),
    React.DOM.li(null, 'Patatas')
  );
  ReactDOM.render(miLista, document.getElementById('miApp'));
</script>

```

Nótese que la intención del ejemplo anterior es simplemente ilustrar que unos componentes pueden contener otros. Por desgracia tal y como está el ejemplo no es muy útil, ya que los datos de la lista están fijados en el código. Sería interesante que los items se pudieran especificar desde fuera del componente (por ejemplo obtenidos del servidor) y además que la lista fuera modificable (por ejemplo ir "tachando" las cosas a medida que las vamos comprando). En los siguientes apartados veremos cómo se pueden conseguir estas funcionalidades a través de las *propiedades* y del *estado* del componente, respectivamente.

Componentes personalizados. Clases.

Usar como componente de nuestra aplicación directamente uno de los predefinidos en React puede ser divertido (ejem) pero tiene un alcance limitado, ya que la posible personalización se limita a especificar los atributos de la etiqueta HTML. Es mucho más útil poder controlar también su comportamiento, y para eso necesitaremos crear nuestra propia *clase* de componente.

```

<script src="lib/react.js"></script>
<script src="lib/react-dom.js"></script>
<div id="miApp"></div>
<script>
  var MiPrimerComponente = React.createClass({ ❶
    render: function() { ❷
      var mensaje = "Soy un componente personalizado";
      mensaje += " (" + mensaje.length + " caracteres )";
      return React.DOM.h1(null, mensaje);
    }
  });
  var instancia = React.createElement(MiPrimerComponente, null); ❸
  ReactDOM.render(instancia, document.getElementById('miApp'));
</script>

```

- ❶ Para crear un componente propio usamos `React.createClass`. Esta función acepta como parámetro un literal JSON con las propiedades del componente.
- ❷ Como mínimo el componente debería tener una función `render` que puede ejecutar el código que queramos y finalmente debe devolver un componente React como resultado.
- ❸ Para crear una instancia de un componente propio usamos este método. El segundo parámetro vamos a dejarlo por el momento a `null` y posteriormente veremos su uso.

Como vemos la función `render` del componente es similar en espíritu a la del mismo nombre de Backbone. Con la diferencia, eso sí, de que con React no manipulamos directamente el DOM de la página, sino que trabajamos con una capa de abstracción por encima.

6.3. Componentes sin estado. Propiedades

En el ejemplo anterior el mensaje que muestra el componente está fijado en el propio código, lo que claramente no es muy flexible. React permite que los componentes tengan

propiedades, que podemos pasarle al componente cuando lo instanciamos. Las propiedades se especifican en forma de literal Javascript, y desde el componente serán accesibles a través de `this.props`. Veamos cómo reescribiríamos el ejemplo anterior para hacer uso de propiedades:

```
<script src="lib/react.js"></script>
<script src="lib/react-dom.js"></script>
<div id="miApp"></div>
<script>
  var MiComponente = React.createClass({
    render: function() {
      var mensaje = this.props.texto; ❶
      mensaje += " (" + mensaje.length + " caracteres)";
      return React.DOM.h1(null, mensaje);
    }
  });
  var instancia = React.createElement(MiComponente, {texto:'Hola
  React'}); ❷
  ReactDOM.render(instancia, document.getElementById('miApp'));
</script>
```

- ❷ Pasamos las propiedades en forma de literal JS. Es decir, las propiedades no son más que un objeto Javascript. En este caso dicho objeto tiene una propiedad llamada `texto`.
- ❶ Desde el componente accedemos a las propiedades con `this.props`.

Según la filosofía de React **las propiedades de un componente deberían ser solo de lectura**. Por tanto no deberíamos modificar en el código del componente el valor de `this.props`. Como el valor de `props` no es en principio modificable, a este tipo de componentes se les denomina estáticos o *sin estado* (*stateless*).



Nada nos impide modificar `this.props`, bien directamente o bien a través del método `setProps`. En React modificar las propiedades de un componente durante su ciclo de vida se considera una mala práctica. Para almacenar datos que puedan cambiar durante la vida del componente usaremos el *estado*, como veremos en el siguiente apartado.

Veamos cómo trabajaríamos con un `props` algo más complicado. Volvamos al ejemplo de la lista de la compra y vamos a solucionar el primero de sus problemas: que los *items* de la lista están fijos en el código.

```
<script src="lib/react.js"></script>
<script src="lib/react-dom.js"></script>
<div id="miApp"></div>
<script>
  var ListaCompra = React.createClass({
    render: function() {
      return React.DOM.ul(null,
        this.props.items.map(function(item) { ❶
          var texto = item.nombre + ' (' + item.cantidad + ')';
          return React.DOM.li(null, texto);
        })
      );
    }
  });
```

```

    var lista = [{nombre:'huevos', cantidad:12}, {nombre:'pan',
cantidad:1}]; ❷
    var instancia = React.createElement(ListaCompra, {items: lista}); ❸
    ReactDOM.render(instancia, document.getElementById('miApp'));
</script>

```

- ❷ Por simplicidad definimos los datos de la lista directamente como una variable en nuestro código, pero idealmente vendrían del servidor como respuesta a una petición al API.
- ❸ Pasamos la lista como la propiedad `items` del componente, por lo que dentro de él será accesible como `this.props.items`
- ❶ Generamos el array de `React.DOM.li` a partir de la propiedad `items` usando el método de Javascript estándar llamado `map`.

El método `map`⁵⁹ no tiene nada que ver con React, es javascript "puro". Lo que hace es ejecutar una función para cada uno de los componentes del array e ir componiendo un nuevo array con los resultados que va devolviendo dicha función. En nuestro caso partimos del array `items` y queremos obtener un array de `React.DOM.li`.



Si has usado alguna vez programación funcional probablemente recuerdes que `map` es típica de este paradigma. Podríamos haber empleado un bucle `for` convencional para generar el array de `React.DOM.li`, pero en casi todos los ejemplos de React que encontrarás por ahí se suele usar `map`. Una vez asimilado su uso produce un código bastante más elegante y compacto.



Te estarás dando cuenta de que conforme se va complicando el HTML del componente, crearlo usando el API de `React.DOM.*` se va haciendo cada vez más tedioso. Luego veremos una sintaxis alternativa denominada JSX que nos permite escribir directamente el HTML del componente, resultando en un código mucho más intuitivo.

Si ejecutas el ejemplo anterior y abres la consola del desarrollador de tu navegador, verás un *warning* que dice "Each child in an array or iterator should have a unique "key" prop". Este mensaje nos está indicando que las etiquetas que forman parte de una colección de elementos deben tener un atributo `key` con un valor único que las identifique. El valor debe ser único dentro de la colección, no es necesario que lo sea dentro de la app de React. Afortunadamente, la función `map` nos pasa como segundo parámetro la posición actual dentro del array, que nos puede servir como valor para `key`. Así que arreglaremos el código dejándolo como sigue:

```

var ListaCompra = React.createClass({
  render: function() {
    return React.DOM.ul(null,
      this.props.items.map(function(item, pos) {
        var texto = item.nombre + ' (' + item.cantidad + ')';
        return React.DOM.li({key:pos}, texto);
      })
    );
  }
});

```

⁵⁹ https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map

6.4. Componentes con estado

Para los componentes que vayan a ser interactivos necesitaremos probablemente mantener un *estado*. La diferencia entre estado y propiedades es que el primero puede cambiar durante el ciclo de vida del componente. Recordemos que las propiedades no deben cambiar.

Por ejemplo supongamos que en la lista de la compra queremos ir "tachando" las cosas a medida que las vamos comprando. Para cada *item* de la lista necesitaremos una variable booleana. Por tanto el estado del componente "lista de la compra" podría ser simplemente un array de valores booleanos.

El estado de un componente se guarda en una propiedad llamada *state*, que al estilo de *props* es simplemente un objeto Javascript con las propiedades que deseemos. Nuestro objetivo va a ser que visualmente los *items* ya comprados aparezcan "tachados". Para eso lo primero que definiremos en el HTML es el estilo CSS apropiado:

```
<head>
  <style>
    .tachado {text-decoration: line-through}
  </style>
  <!-- aquí vendría el resto de tags que tengamos en el head -->
</head>
```

Ahora modificaremos el `render` para que asigne la clase `tachado` a los items ya comprados, algo como:

```
render: function() {
  return React.DOM.ul(null,
    this.props.items.map(function(item, indice) { ❶
      var texto = item.nombre + ' (' + item.cantidad + ')';
      if (this.state.comprados[indice]) ❷
        return React.DOM.li({className: 'tachado'}, texto);
      else {
        return React.DOM.li(null, texto);
      }
    }).bind(this) ❸
  );
}
```

- ❶ Como el estado es un array que almacena los datos por posición del item en la lista de la compra, necesitamos saber el índice del item actual. Por suerte el `map` pasa automáticamente a su *callback* dicho índice como segundo parámetro.
- ❷ La propiedad `comprados` dentro de `this.state` es el array con los valores booleanos que indican si un item ha sido ya comprado o no. En caso de que haya sido comprado, devolvemos un `li` de la clase `tachado`. En caso contrario devolvemos un `li` "simple". Nótese que como estamos asignando el atributo `class` de HTML, que es una palabra reservada en Javascript, nos vemos obligados a cambiarla por `className`, que es la convención habitual.
- ❸ Nos hemos visto obligados a añadir el `bind(this)` al *callback* ya que si no lo hacíamos el contexto de ejecución del mismo (su valor de `this`) acababa siendo `window`. Nosotros queremos que `this` sea el componente porque ahora necesitamos acceder a `this.state`.

Nos queda por atar un par de "cabos sueltos": ¿Cómo se modifica el estado y cómo se fija su valor inicial?.

Fijar el valor inicial del estado es bastante sencillo: un componente con estado debería definir una función llamada `getInitialState` que devolviera el estado inicial. En nuestro caso algo como:

```
getInitialState: function() {  
  return {comprados: new Array(this.props.items.length)}  
}
```

Es decir, creamos un array de tantos elementos como tengamos en la lista de la compra (propiedad `items` del componente). No necesitamos inicializar el array ya que contendrá `undefined` y recordemos que en términos booleanos esto es lo mismo que `false`.

Con respecto a modificar el estado, hay dos cosas de las que hablar: cómo se hace en el API de React y cómo se implemente a nivel de interfaz de usuario. Con respecto a esto último, probablemente en una aplicación real cada item sería "pulsable" o tendría una casilla de verificación para cambiar su estado. No obstante la parte de la interfaz la vamos a dejar de lado por el momento para no complicarnos con el manejo de eventos en React. Por ahora baste con saber que en el API de React para cambiar el estado se usa el método `setState`, pasándole un objeto con las propiedades que queramos modificar

```
this.setState({comprados:[true, false]})
```



no hay que cambiar nunca el estado directamente accediendo a `this.state`, siempre hay que hacerlo a través de la llamada a `setState`, ahora mismo veremos por qué.

Como no tenemos por el momento un mecanismo para que el usuario final pueda marcar un item como comprado tendremos que hacer un poco de "trampa" y manipular el componente desde la consola Javascript. El componente React es en realidad el resultado que devuelve la llamada a `ReactDOM.render`:

```
var componente = ReactDOM.render(instancia,  
  document.getElementById('miApp'));
```

Ahora si abrimos la consola Javascript del navegador podemos hacer algo como:

```
componente.setState({comprado:[true, false]})
```

Si probamos esto experimentaremos por fin la parte más *cool* de React: la llamada a `setState` desencadena automáticamente un *repintado* del componente. Nuestra función `render` se vuelve a llamar de nuevo, como podríamos comprobar si insertáramos en ella un `console.log`. Y por tanto la lista aparece con el primer componente tachado.

Podríamos pensar que ejecutar el `render` entero cada vez que cambie el estado es extremadamente ineficiente, pero en una función de este tipo el cuello de botella va a estar en la manipulación del DOM y no en el resto de código (salvo que el código de `render` hiciera cálculos costosos, lo que normalmente no es el caso, y de darse sería una mala práctica). Nótese que con React no manipulamos directamente el DOM sino que usamos objetos propios

del *framework* (`React.DOM.*`), lo que en React se denomina el *DOM virtual*. Lo que va a hacer React es a partir de este DOM virtual calcular automáticamente la *transformación mínima que se debería hacer en el DOM "real"* para obtener el resultado deseado. Es decir, si hemos cambiado el estado únicamente del primer ítem no hace falta modificar todo el `ul`, solo el primer `li`. Y esto es lo que va a calcular React automáticamente para nosotros en cada cambio de estado del componente.

Podemos usar las [herramientas de medición de rendimiento](#)⁶⁰ de React (`React.addons.Perf`) para comprobar qué está haciendo para actualizar la interfaz:

```
React.addons.Perf.start();
componente.setState({comprado:[true, false]});
React.addons.Perf.stop();
React.addons.Perf.printDOM();
```

Para poder usar las herramientas de medición de rendimiento hay que incluir el `script react-with-addons.js` en lugar del `react.js` original.

Con las llamadas a `start()` y `stop()` comienza y termina el bloque de código en que queremos las medidas. Una vez ejecutado el `stop` podemos ver distintas tablas de rendimiento, por ejemplo `printDOM()` imprime las modificaciones que React ha tenido que hacer en el DOM para actualizar la interfaz (figura 1). Nótese que solo ha modificado el elemento necesario, no el resto. Además React ha hecho un "buen trabajo" ya que ha detectado que el único cambio necesario era añadir la clase "tachado" a la etiqueta.

(index)	data-reactid	type	args
0	".0.0"	"update attribute"	"[\"className\", \"tachado\"]"

Total time: 14.41 ms

Figura 5. Operaciones sobre el DOM para actualizar la interfaz de la "lista de la compra"

Es fácil de adivinar que la "prohibición" de modificar directamente la variable `state` es porque si cambiamos directamente la variable React no está "informado" de que el estado ha cambiado, mientras que al llamar a `setState` se lo estamos indicando.



`setState` no tiene por qué desencadenar un cambio inmediato del estado. Si en nuestro código llamamos a `setState` e inmediatamente después accedemos a `this.state` es posible que nos encontremos con que este no ha cambiado todavía. Por cuestiones de eficiencia React determina cuándo cambiar de manera efectiva el estado, en *batches*.

6.5. Interactividad. Eventos

Recapitulemos primero sobre cómo se tratan los eventos de la interfaz de usuario en aplicaciones no React. En la mayoría de los casos (aunque no siempre) los cambios en la interfaz se desencadenan como consecuencia de acciones del usuario, por ejemplo un *clic* sobre un botón. En Javascript ya estamos acostumbrados a modelar dichas acciones como eventos y a tratarlas con manejadores de eventos. La forma clásica de definir estos manejadores en la web es usando la familia de atributos HTML `on_XXX`, por ejemplo:

```
<p onClick="alert('has hecho clic')">Hola</p>
```

⁶⁰ <http://facebook.github.io/react/docs/perf.html>

En React vamos a definir los manejadores de eventos de forma similar (aunque como veremos es similar solo en la sintaxis, pero no así en el funcionamiento interno). Al igual que podemos definir cualquier atributo HTML de un componente de `React.DOM` en sus propiedades, podemos hacer lo propio con los manejadores de evento asociados:

```
<div id="miApp"></div>
<script>
  var MiComponente = React.createClass({
    saludar: function() { ❶
      alert('Has hecho clic');
    },
    render: function() {
      return React.DOM.h1({onClick:this.saludar}, 'Hola'); ❷
    }
  });
  var instancia = React.createElement(MiComponente, null)
  ReactDOM.render(instancia, document.getElementById('miApp'));
</script>
```

- ❶ Definimos el manejador de evento como una función en nuestro componente
- ❷ Asociamos el evento con el manejador dando el valor para el correspondiente atributo `onXxx`. Nótese que React usa la sintaxis "camel case" para este atributo.

Visto así parece que React usa simplemente la sintaxis clásica para definir manejadores de evento, adaptada a las necesidades del *framework*, pero en realidad hay mucho más.

Por un lado, **el API de React usa eventos sintéticos**, es decir el desarrollador no está tratando directamente con los eventos nativos del navegador, sino con una fina capa de abstracción que uniformiza el tratamiento de los eventos independientemente de la compatibilidad del navegador (un poco al estilo jQuery). Así, todos los manejadores de evento reciben automáticamente un objeto con las propiedades del evento, al estilo W3C, y dichas propiedades serán [las mismas en todos los navegadores](#)⁶¹.

Por otro lado, da la impresión de que estamos asociando directamente el manejador de evento a la etiqueta HTML correspondiente cuando en realidad no es así. **React usa automáticamente delegación de eventos**, lo que quiere decir que vincula los manejadores de evento al nivel superior del DOM. Este es un método de gestión de eventos mucho más eficiente que vincular los manejadores a los nodos del DOM directamente implicados (pensemos por ejemplo qué pasaría con un evento clic al que tiene que responder cada fila de una tabla). Para ver con más detalle cómo funciona la delegación de eventos se puede consultar [este artículo](#)⁶², referenciado en la documentación de React.

Vamos a aplicar esto a nuestro ejemplo de la lista de la compra. Queremos que cada vez que se pulse sobre un item este cambie de estado (de no comprado a comprado, y viceversa)

```
<script src="lib/react-with-addons.js"></script>
<script src="lib/react-dom.js"></script>
<div id="miApp"></div>
<script>
  var ListaCompra = React.createClass({
    getInitialState: function() {
```

⁶¹ <https://facebook.github.io/react/docs/events.html>

⁶² <https://davidwalsh.name/event-delegate>

```

    return {comprados: new Array(this.props.items.length)}
  },
  toggle: function(indice) { ❶
    var compradosNew = this.state.comprados.slice(0); ❷
    if (this.state.comprados[indice]) ❸
      compradosNew[indice] = false;
    else {
      compradosNew[indice] = true;
    }
    this.setState({comprados: compradosNew}); ❹
  },
  render: function() {
    return React.DOM.ul(null,
      this.props.items.map(function(item, indice) {
        var texto = item.nombre + ' (' + item.cantidad + ')';
        var atribs = {onClick:this.toggle.bind(this,indice),
key:indice}
        if (this.state.comprados[indice]) {
          atribs.className = 'tachado';
          return React.DOM.li(atribs, texto);
        }
        else {
          return React.DOM.li(atribs, texto);
        }
      }).bind(this)
    );
  }
});
var lista = [{nombre:'huevos', cantidad:12}, {nombre:'pan',
cantidad:1}];
var instancia = React.createElement(ListaCompra, {items: lista});
ReactDOM.render(instancia, document.getElementById('miApp'));
</script>

```

- ❶ Nuestro manejador de evento se ocupa de cambiar el estado correspondiente al ítem en que hemos hecho *clic*. Recibe como parámetro el índice de dicho ítem, en un momento veremos cómo se lo hemos pasado.
- ❷ Clonamos el array con la información sobre productos comprados. Un truco muy habitual en Javascript es usar el método `slice`, que nos devuelve una copia de una porción de un array (o del array entero si pasamos 0 como índice).
- ❸ Cambiamos el estado del ítem en que hemos hecho clic de `true` a `false` o viceversa.
- ❹ Cambiamos el estado pasándole el nuevo array a `setState`. Esta llamada desencadenará el repintado del componente.

6.6. JSX

Habrás visto que el código del método `render` se complica mucho conforme se va complicando el HTML a generar. Se hace cada vez más tedioso usar el API Javascript para generar HTML. Es el mismo problema que tiene el API del DOM estándar cuando necesitamos generar HTML dinámicamente con él.

En React existe una sintaxis alternativa, llamada **JSX**, que nos permite mezclar fragmentos de HTML (XML, en realidad) con código JS de manera mucho más natural y concisa que con el API anterior. Por eso el código que hemos visto hasta el momento no es "nada típico" en los ejemplos, libros y tutoriales de React que puedes ver por ahí.

Así, recordemos nuestro primer ejemplo de "Hola React"

```
<div id="miApp"></div>
<script>
  var miComponente = React.DOM.h1({id:"saludo"}, '¡Hola React!');
  ReactDOM.render(miComponente, document.getElementById('miApp'));
</script>
```

Lo podríamos reescribir usando JSX del siguiente modo:

```
<script src="react.js"></script> ❶
<script src="react-dom.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/
browser.min.js"></script>
<div id="miApp"></div>
<script type="text/babel">
  var saludo = "Hola";
  var miComponente = <h1 id="saludo">¡{saludo} React!</h1>;
  ReactDOM.render(miComponente, document.getElementById('miApp'));
</script>
```

Lo más notable del ejemplo es que estamos escribiendo directamente el HTML que queremos generar, en lugar de usar un API Javascript.

Además hemos aprovechado para insertar JS en medio del HTML para que se vea que es posible hacerlo, sin más que rodearlo de llaves `{...}`. Por lo demás el código es funcionalmente equivalente al ejemplo que no usa JSX, solo que con una sintaxis mucho más concisa y cómoda de usar.

Por otro lado podemos ver que referenciamos un `<script>` externo adicional y que además hemos añadido un atributo `type="text/babel"` a nuestro código. Explicaremos esto en el siguiente apartado.

De JSX a JS

Escribir código JSX no es simplemente concatenar cadenas de HTML dentro del JS. Nótese que *el HTML está tal cual dentro del código, sin delimitadores*. Por eso este `script` no encaja con la sintaxis JS estándar y de ahí que en el `type` del `script` se haya puesto el valor especial `text/babel`. Hay que incluir una librería adicional para *parsear* el JSX y transformarlo automáticamente a JS convencional, resultando en algo del estilo de la versión "antigua" de nuestro "Hola React". Las etiquetas tipo HTML se convierten en llamadas a `React.createElement`. El proceso de traducción de JSX a JS se denomina "transpilación". Aquí tenemos un ejemplo de cómo se traduciría

```
// Original (JSX):
var app = <div className="pie" />;
// Resultado (JS):
var app = React.createElement('div', {className:"pie"});
```



Hay que acordarse del `type="text/babel"` en el `tag script`, siempre que escribamos código JSX y queramos que lo transforme a JS el propio navegador.

Para que el navegador transpile el JSX a JS "sobre la marcha" simplemente incluimos en el HTML la librería adecuada. En versiones antiguas de React esto se hacía en una librería propia denominada JSXTransformer. A partir de la versión 0.14 se hace con una librería de terceros denominada [Babel](#)⁶³.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
```

Babel es un proyecto *open source* ampliamente conocido y usado en la web, ya que no solo hace *transpilación* de JSX a JS, sino también de ES6 a ES5, permitiéndonos usar la nueva sintaxis JS en los navegadores actuales, aunque no ofrezcan soporte nativo.

En aplicaciones en producción, por eficiencia no se recomienda hacer la *transpilación* desde el propio navegador, sino realizarla *offline* como parte del proceso de construcción de la aplicación, e incluir en el HTML el JS ya transformado. Se recomienda consultar la documentación de React para ver más detalles sobre el proceso.

Aquí tenemos otro ejemplo un poco más complicado para ilustrar cómo mezclar JSX y JS, el componente `ListaCompra` (sin estado y sin eventos, para simplificar el código). Nótese que se pueden introducir variables y fragmentos de código JS entre llaves `{...}`.

```
var ListaCompra = React.createClass({
  render: function() {
    var items = this.props.items.map(function(item, indice){
      return <li key={indice}> {item.nombre} ({item.cantidad}) </li>
    });
    return <ul>{items}</ul>
  }
});

var lista = [{nombre:'huevos', cantidad:12}, {nombre:'pan', cantidad:1}];
var instancia = <ListaCompra items={lista}/>;
var componente = ReactDOM.render(instancia,
  document.getElementById('miApp'));
```

Nótese que cuando creamos un componente React, en JSX estamos "creando" la etiqueta equivalente, que podemos usar en nuestro código. En nuestro caso hemos creado una etiqueta `<ListaCompra>`. Para pasarle `props` al componente usamos atributos con el mismo nombre.

"Gotchas" de JSX

Vamos a ver algunas pequeñas pegadas o problemas que se nos pueden plantear en el uso de JSX.

La primera cuestión importante es que **JSX usa sintaxis XML, no HTML**. Eso significa que todas las etiquetas deben abrirse y cerrarse. En HTML el cierre de algunas es opcional, por ejemplo el de `<p>`. Los atributos HTML deben usar sintaxis *camel case*.

```
//(...en una función render)
```

⁶³ <http://babeljs.io>

```
//¡¡no va a funcionar!!  
return <a onClick={this.saludar} href='#>Clicame</a>;  
//pero esto sí  
return <a onClick={this.saludar} href='#>Clicame</a>;
```

Otra cosa a tener en cuenta es que el `render` **solo debe devolver una única etiqueta de "nivel superior"**. Es decir, esto no es legal en un `render`.

```
return (  
  <span>Hola</span>  
  <span>mundo</span>  
)
```

Pero esto sí

```
return (  
  <div>  
    <span>Hola</span>  
    <span>mundo</span>  
  </div>  
)
```

El operador "spread"

Es sencillo tomar los valores de los atributos HTML de variables Javascript. Como hemos visto, solo hay que usar la sintaxis `{}`. Por ejemplo:

```
render: function() {  
  var url = 'http://expertojava.ua.es';  
  var destino = '_blank';  
  return <a href={url} target={destino}>Experto Java</a>;  
}
```

Lo que sucede es que si el número de atributos es muy grande puede llegar a ser un poco tedioso ir poniéndolos todos. Para simplificar esto, podemos usar una funcionalidad que JSX toma prestada de ES6 denominada el *spread operator* o el operador `...`. Creamos un objeto con las propiedades correspondientes a los atributos y estos se asignarán automáticamente, por nombre.

```
render: function() {  
  var enlace = {  
    href: 'http://expertojava.ua.es',  
    target: '_blank'  
  };  
  return <a {...enlace}>Experto Java</a>;  
}
```

Nótese que hemos tenido que cambiar los nombres de las variables para que coincidan con los atributos de la etiqueta `<a>`, pero es un pequeño precio a pagar por la simplificación de la sintaxis.

6.7. Manejo de formularios

Los formularios tienen algunas peculiaridades en React con respecto a lo que es habitual en el DOM nativo. Vamos a verlas.

Los componentes en los que hay diferencias son los interactivos, como `<input>`, `<textarea>` o `<option>`. Estos componentes soportan varias `props` a las que afectan las interacciones del usuario:

- `value`: al igual que en HTML para los `<input>`, y en React se usa también para `<textarea>` por consistencia (recordemos que en HTML es el contenido de la etiqueta).
- `checked`: para los `<input>` de tipo `checkbox` o `radio`.
- `selected`: para los `<option>`.

Componentes controlados

Los componentes para los que especificamos estas `props` se dice que son **controlados**. La interacción del usuario no tendrá un efecto directo sobre ellos. Es decir, si tenemos un componente cuyo `render` es así:

```
render: function() {  
  return <input type="text" value="hola"/>;  
}
```

Estamos especificando que el valor del campo sea *siempre* "hola", por lo que React no "hará caso" a lo que el usuario quiera modificar en el campo. Nótese que en HTML el atributo `value` significa el valor inicial, pero en React significa el valor, en cualquier momento. Si queremos especificar el valor inicial podemos usar la `prop` llamada `defaultValue`.

Si queremos que la interacción del usuario tenga efecto sobre un componente controlado, tenemos que hacerlo explícitamente usando los eventos, por ejemplo, para hacer que se pueda escribir en un campo de texto controlado lo que haremos será que el `value` dependa del estado actual e ir cambiando el estado a medida que el usuario va tecleando (evento `onChange`). Este evento se dispara con cada pulsación de tecla, a diferencia del `onChange` de HTML:

```
var MiForm = React.createClass({  
  getInitialState: function() {  
    return {value: this.props.texto};  
  },  
  handleChange: function(event) {  
    this.setState({value: event.target.value});  
  },  
  render: function() {  
    return <input type="text" value={this.state.value}  
      onChange={this.handleChange}/>;  
  }  
});  
var instancia = <MiForm texto="hola"/>  
var componente = ReactDOM.render(instancia,  
  document.getElementById('miApp'));
```

Esto puede parecer muy engorroso, pero nos permite controlar perfectamente el estado del campo, por ejemplo si quisiéramos limitar el texto a 140 caracteres, cambiaríamos la función `handleChange` anterior por esta:

```
handleChange: function(event) {  
  this.setState({value: event.target.value.substr(0, 140)});  
}
```

refs

Una necesidad muy típica cuando se trabaja con formularios no controlados es obtener el valor actual que tiene un campo. Para esto podemos usar una "ref", que salvando las distancias viene a ser como los `id` del DOM tradicional, pero limitados al componente. Así si en un `render` hacemos:

```
<input ref="miCampo"/>
```

En una función de nuestro componente podríamos hacer esto para referenciar al campo

```
var valor = this.refs.miCampo.value;
```

Esta forma en la que un `ref` se especifica como una cadena es *legacy* en la versión actual de React, y tiende a sustituirse por la posibilidad de asignar una función *callback* que se ejecutará en el momento de montar el componente. Este *callback* puede por ejemplo guardar la referencia para usarla posteriormente:

```
render: function() {  
  return <input ref="{function(nodo) {this.campo = nodo}}"/>;  
}  
otraFuncion: function() {  
  console.log(this.campo.value);  
}
```

Nótese que los `refs` no están ni mucho menos limitados a los formularios, son usables en cualquier tipo de componente, pero el empleo en formularios es un caso de uso muy típico

6.8. Ejercicios de React (I)

Almacena los ejercicios en una carpeta `sesion06`. Para todos ellos puedes usar como base la carpeta `plantilla-react` incluida en el repositorio de plantillas. Simplemente puedes copiar/pegar y cambiar de nombre.

El retorno del Widget del tiempo (0,4)

Guarda este ejercicio en una carpeta `sesion06/tiempo`.

Reescribe usando React el widget del tiempo que hicimos en la sesión 1. Guárdalo en una carpeta llamada `tiempo_react_v1`. No uses JSX en este ejercicio, ya que uno de los objetivos es practicar con el API `React.DOM.*`.

```
var TiempoWidget = React.createClass({
  getInitialState: function() {
    ...Devolverá un estado con una descripcion y una icono_url vacíos
  }
  ('')
  verTiempo: function() {
    ...Aquí irá la llamada AJAX al servidor
  }
  render: function() {
    ...Devolverá el widget "envuelto" en un <div>, usando los métodos
    React.DOM.*
  }
});
```

Guías para la implementación:

- Debes eliminar del código todas las referencias a Backbone y usar sólo React. No obstante puedes reutilizar código de la sesión 1, siempre que no use el API de Backbone.
- incluye jQuery en la página si quieres seguir usando el `$.getJSON` para hacer la llamada AJAX. Puedes copiar el `.js` de la plantilla de Backbone o referenciarla de una CDN

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.2.0/
jquery.min.js"></script>
```

Componente tabla de datos

Almacena este ejercicio en una carpeta `/sesion06/tabla`

Crea un componente que muestre una serie de datos como una tabla HTML. Los datos se obtendrán de dos arrays:

```
var cabeceras = ['titulo', 'distribuidora', 'recaudación ($)'];
var datos = [['Jurassic World', 'Universal', 1668984926],
['Furious 7', 'Universal', 1515047671],
['Los Vengadores: la era de Ultrón', 'Buenavista', 1405035767],
['Star Wars: el despertar de la Fuerza', 'Buenavista', 1228349526],
['Los Minions', 'Universal', 1157275017],
['Del revés', 'Buenavista', 856130132],
['Spectre', 'Sony', 850604955],
['Misión Imposible: nación secreta', 'Paramount', 682330139],
```

```
[ 'Los Juegos del Hambre: Sinsajo parte 2', 'Lionsgate', 619444461 ] ,
[ 'Marte (The Martian)', 'Fox', 594161725 ] ]];
```

Puedes usar JSX, simplificará el código.



Recuerda usar el `type=text/babel` en el `<script>`.

Haz dos versiones del componente

En la **primera versión**, `tabla_v1.html`, **se mostrarán los datos de forma estática** (0,4 puntos). Decide si es más conveniente usar `props` o `state` para guardar los datos.

Puedes usar dos `map` anidados para generar los datos, o bucles convencionales

En la **segunda versión**, `tabla_v2.html`, **haz que además de lo anterior se puedan ordenar los datos** (0,45 puntos) haciendo *clic* sobre las celdas de la cabecera. Para simplificar basta con hacer que aparezcan ordenados solo de menor a mayor, no es necesario cambiar el sentido de la ordenación al hacer *clic* de nuevo en la misma columna.

Para ordenar puedes usar la siguiente función, que debes hacer que se dispare con el *clic* en una celda de la cabecera:

```
ordenar: function(evento) {
  //evento.target es la celda en que se ha pulsado, y cellIndex el
  número de la celda
  var numCol = evento.target.cellIndex;
  //con slice(0) clonamos el array
  var ordenados = this.state.datos.slice(0);
  //con sort ordenamos. Hay que pasarle un callback que, dadas dos filas
  A y B devuelva:
  //<0 si A debe ir antes que B
  //>0 si A debe ir después que B
  ordenados.sort(function(filaA, filaB){
    if (typeof filaA[numCol]==='string')
      return filaA[numCol].localeCompare(filaB[numCol]);
    else {
      return filaA[numCol]-filaB[numCol];
    }
  });
  //ATENCIÓN. Si tu variable de estado no se llama 'datos' tendrás que
  cambiar esto
  this.setState({datos:ordenados});
}
```

7. Interfaces web con ReactJS (II)

En esta segunda sesión vamos a ver cómo construir jerarquías de componentes. No solo desde el punto de vista de qué código necesitamos para ello, sino también cuáles son las "buenas prácticas" al hacerlo, sobre todo la cuestión de cuándo usar estado y cuándo usar `props`. También veremos cuál es el ciclo de vida de un componente y dónde introducir nuestro código para que se dispare en el momento adecuado de este ciclo. Terminaremos dando unas guías de cómo usar Backbone para almacenar los modelos de una aplicación React, lo que puede ser útil si tenemos una aplicación Backbone y queremos aprovechar las ventajas que nos da React sobre las vistas nativas de Backbone.

7.1. Composición de componentes

Una de las características más interesantes de React es que los componentes se pueden anidar o componer (ejem, por otro lado...¿qué clase de componentes serían si no?). Como [se resume perfectamente](#)⁶⁴ en la propia documentación de React:

"Al desarrollar componentes modulares que reutilizan otros componentes con interfaces bien definidas, se consiguen los mismos beneficios que al usar funciones o clases. Específicamente se pueden separar las diferentes responsabilidades (*concerns*) de la aplicación como mejor parezca simplemente desarrollando nuevos componentes. Al desarrollar una biblioteca de componentes personalizados para la aplicación se está expresando la interfaz de usuario en el modo que mejor se ajusta al dominio"

Por ejemplo veamos la siguiente jerarquía de componentes, tomada de un [tutorial](#)⁶⁵ muy interesante que forma parte de la documentación de React.

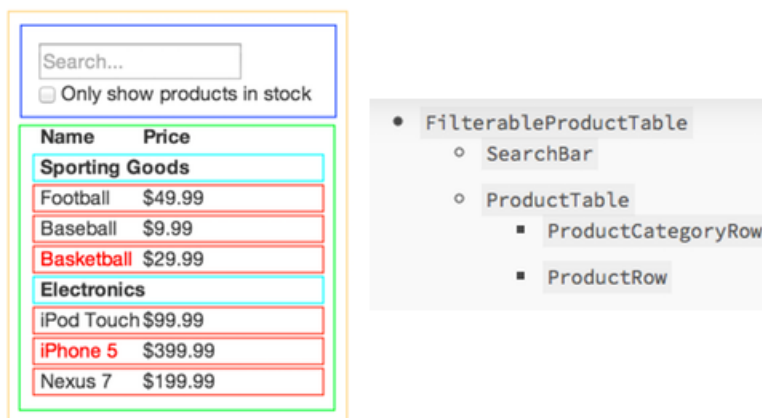


Figura 6. Ejemplo de jerarquía de componentes

Yendo a un ejemplo un poco más sencillo veamos cómo podríamos "componentizar" nuestra lista de la compra, por el momento dejando fuera el estado y los eventos, para simplificar. Nuestro componente `ListaCompra` ahora contendrá una colección de componentes de tipo `Item`:

```
var Item = React.createClass({
  render: function() {
    return <li> {this.props.nombre} ({this.props.cantidad}) </li>
  }
});
```

⁶⁴ <https://facebook.github.io/react/docs/multiple-components.html#motivation-separation-of-concerns>

⁶⁵ <https://facebook.github.io/react/docs/thinking-in-react.html>

```
});

var ListaCompra = React.createClass({
  render: function() {
    var items = this.props.items.map(function(item, indice) {
      return <Item key={indice} nombre={item.nombre}
        cantidad={item.cantidad}/>
    });
    return <ul>{items}</ul>;
  }
});
var lista = [{nombre: 'huevos', cantidad: 12}, {nombre: 'pan', cantidad: 1}];
var instancia = <ListaCompra items={lista}/>;
ReactDOM.render(instancia, document.getElementById('miApp'));
```

Como en JSX cada componente se escribe como una etiqueta lo único que hacemos de especial es generar la etiqueta `<Item>` en lugar de directamente HTML, dentro del `render` de `ListaCompra`.

En la terminología de React se dice que `ListaCompra` es el componente "dueño" (*owner*) del componente `Item`. Un componente "dueño" fija el valor de los `props` de sus componentes hijos. Es decir, en React existe un *flujo de datos unidireccional* que va del dueño a sus hijos en forma de `props`.

Dónde colocar el estado

Vamos a añadir estado a nuestra nueva versión de la lista de la compra. Como ahora tenemos dos componentes, ¿En cuál deberíamos mantener el estado, en `ListaCompra` como hacíamos antes, o en cada `Item` por separado?. Para responder a esta pregunta debemos revisar la filosofía de diseño de React. Uno de estos principios de diseño es *reducir al máximo el número de componentes que deben mantener estado*. La idea es que esto hará los componentes más fáciles de depurar y testear, ya que para un componente sin estado (solo con `props`) solo necesitamos comprobar si dado un determinado valor para estas *props* el resultado es el deseado.

Se puede formular una serie de reglas prácticas que nos permitan identificar dónde almacenar el estado. Para cada variable perteneciente al estado debemos:

- Identificar todos los componentes que *renderizan* algo basándose en esa variable
- Encontrar un componente dueño común a todos ellos (es decir, un único componente por encima en la jerarquía de todos los que necesitan ese estado).
- El estado debería residir en ese dueño común o en otro componente todavía más alto en la jerarquía.
- Si no podemos encontrar un componente en el que pueda residir el estado, crear uno nuevo simplemente para almacenarlo, y añadirlo en la jerarquía en algún lugar por encima de ese dueño común.

Las reglas anteriores están tomadas del excelente tutorial que ya hemos nombrado aquí llamado *Thinking in React*⁶⁶, de Pete Hunt.

En nuestro caso parece claro según las reglas anteriores que el estado debería residir en `ListaCompra` y no en cada `Item` por separado. Cada item de la lista debe "saber" si debe

⁶⁶ <https://facebook.github.io/react/docs/thinking-in-react.html>

mostrarse tachado o no, pero desde su "punto de vista" esta información es un `prop`, no es estado. Por tanto el código quedará como sigue:

```

<script type="text/babel">
var Item = React.createClass({
  render: function() {
    if (this.props.comprado) {
      return <li className="tachado">{this.props.nombre}
({this.props.cantidad}</li>
    }
    else {
      return <li>{this.props.nombre} ({this.props.cantidad}</li>
    }
  }
});
var ListaCompra = React.createClass({
  getInitialState: function() {
    return {comprados: new Array(this.props.items.length)};
  },
  render: function() {
    var items = this.props.items.map(function(item, indice) {
      return <Item key={indice} nombre={item.nombre}
cantidad={item.cantidad} comprado={this.state.comprados[indice]}>
    }.bind(this));
    return <ul>{items}</ul>;
  }
});
var lista = [{nombre: 'huevos', cantidad:12}, {nombre: 'pan', cantidad:1}];
var instancia = <ListaCompra items={lista}/>;
window.miComponente = ReactDOM.render(instancia,
  document.getElementById('miApp'));
</script>

```

Para poder "jugar" con la variable `miComponente` desde la consola del desarrollador la hemos metido dentro del objeto global predefinido `window`. Antes no era necesario esto porque escribíamos el código JS directamente, pero tengamos en cuenta que ahora el código está *transpilado* por Babel y las variables definidas en él **no son accesibles desde fuera**⁶⁷.

Para probar el código anterior haríamos cosas como:

```

window.miComponente.setState({comprados: [true, false]})

```

Al cambiar el estado del componente *owner* se llamará automáticamente a su `render`, lo que implica que también se hace el *render* de los hijos con los nuevos valores para las `props`.

Comunicación de abajo a arriba

Aunque el flujo de datos primario en React es de arriba a abajo, en muchos casos será necesario comunicarnos en sentido inverso. Esto es típico de los casos en los que hay interactividad. Por ejemplo en la lista de la compra queremos que al hacer *clic* sobre un ítem cambie su estado (comprado/no-comprado), pero este no está contenido en el propio ítem sino en el "padre". Esto nos obliga a comunicarnos con el padre para notificar el cambio de estado.

⁶⁷ <http://stackoverflow.com/questions/33109430/script-tag-text-babel-variable-scope>

La forma más típica de hacer esto en React es que el padre le pase al hijo un *callback* al que éste debe llamar para indicar el cambio de estado. En nuestro ejemplo sería algo como:

```

var Item = React.createClass({
  toggle: function() {
    this.props.handleClick(this.props.id); ❶
  },
  render: function() {
    var atribs = {
      onClick: this.toggle ❷
    };
    if (this.props.comprado) {
      atribs.className = 'tachado';
    }
    return <li {...atribs}>{this.props.nombre} ({this.props.cantidad})</
li> ❸
  }
});
var ListaCompra = React.createClass({
  getInitialState: function() {
    return {comprados: new Array(this.props.items.length)};
  },
  toggleState: function(id) { ❹
    var compradosNew = this.state.comprados.slice(0);
    compradosNew[id] = !this.state.comprados[id];
    this.setState({comprados: compradosNew});
  },
  render: function() {
    var items = this.props.items.map(function(item, indice) {
      return <Item id={indice} key={indice}
        nombre={item.nombre}
        cantidad={item.cantidad}
        comprado={this.state.comprados[indice]}
        handleClick={this.toggleState}/> ❺
    }).bind(this);
    return <ul>{items}</ul>
  }
});
var lista = [{nombre: 'huevos', cantidad: 12}, {nombre: 'pan', cantidad: 1}];
var instancia = <ListaCompra items={lista}/>;
ReactDOM.render(instancia, document.getElementById('miApp'));

```

- ❶ esta función `toggle` es la que se dispara cuando hacemos *clic* en un item, luego veremos dónde se vincula el evento con ella. Dentro llamamos a `handleClick`, que no es más que una `prop` que nos pasa el padre (como el resto de `props`), pero que en lugar de ser un dato, es una función. A esta función tenemos que llamar para que el padre actualice el estado. Le pasamos el índice del item.
- ❷ Creamos un objeto con los atributos HTML que va a tener nuestro item. El primero es el manejador de evento `onClick`. Aquí vinculamos el evento con la función `toggle`
- ❸ Usamos el *spread operator* para asignar los atributos al componente `Item`
- ❹ En el "padre", este es el *callback* que se llama desde los hijos para indicar que se ha hecho *clic*. Recibimos el índice del hijo como parámetro y actualizamos el estado
- ❺ Finalmente, aquí es donde le hemos pasado el *callback* al hijo como una `prop` más.

Todo esto puede parecer un poco complicado frente a lo "sencillo" que sería mantener el estado en los hijos para evitar la comunicación con el padre. Pero como ya hemos dicho, el objetivo es reducir al máximo el número de componentes con estado. En el extremo de esta filosofía, en que tenemos un único componente con estado, o incluso centralizado en un elemento separado de la jerarquía de componentes, el flujo de datos sería como el de la siguiente figura:

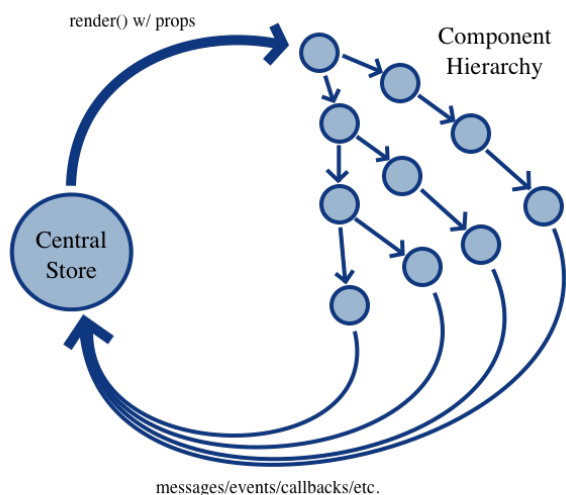


Figura 7. Flujo de datos con estado centralizado (tomada de [React Tips and Best Practices](#)⁶⁸)

Tenemos un flujo de datos unidireccional, y los componentes son sin estado y por tanto más fáciles de testear: "solo" hay que verificar que generan el HTML correcto dado un valor de `props` y que se llama a los `callbacks` adecuados cuando se producen los eventos correspondientes. Pero no hay interacciones complejas entre ellos que dependan del estado actual de unos y otros. Como veremos en la siguiente sesión esta es una de las ideas centrales de una arquitectura denominada **Flux**.

7.2. Ciclo de vida de un componente

Los componentes tienen un ciclo de vida: un componente aparecerá por primera vez en la página, o desaparecerá de ella, o se re-renderizará por haber cambiado su `state` o sus `props`. En ciertos de estos momentos podría interesarnos ejecutar código propio. Por ejemplo podemos aprovechar el momento en que se inserte por primera vez el componente en la página para solicitarle al API REST del servidor los datos que debe mostrar. O el momento en que se cambia el `state` para chequear si es un cambio que no necesita de re-renderizado. React nos ofrece un conjunto de `hooks` o métodos del ciclo de vida, que podemos implementar para ejecutar código propio en el momento adecuado.

A grandes rasgos podemos dividir estos métodos en varios tipos:

- **Montado** de componentes: cuando un componente se inserta por primera vez en la aplicación, o cuando desaparece de ella. Esta parte del ciclo de vida se muestra en la siguiente figura. En rojo se ponen los métodos que nosotros podemos implementar y en gris los procesos "internos" de React en los que no podemos intervenir directamente.

⁶⁸ <http://aeflash.com/2015-02/react-tips-and-best-practices.html>

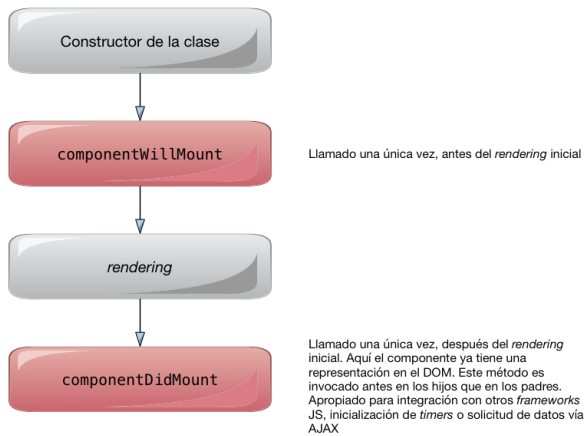


Figura 8. Ciclo de vida al montar un componente

- **Desmontado** de componentes: cuando se elimina de la aplicación. Esto es más típico de los componentes hijos (por ejemplo pensemos en un ítem que se elimina de una lista)



Figura 9. Ciclo de vida al desmontar un componente

- **Cambio en props**

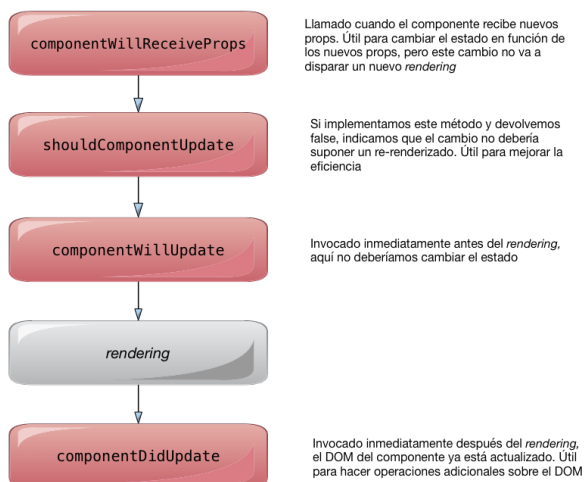


Figura 10. Ciclo de vida al cambiar props

- **Cambio en state** es prácticamente igual al cambio de props, salvo que como es lógico no tiene el primero de los métodos, el `componentWillReceiveProps`.

Ejemplo: carga de datos con AJAX

Un ejemplo de uso muy típico del ciclo de vida es cuando un componente tiene que cargar datos del servidor con AJAX. El momento adecuado para hacer la petición es cuando se dispare el `componentDidMount`. Podemos aprovechar el `componentWillUnmount` para cancelar la petición AJAX si es que todavía está pendiente. Veamos un ejemplo de un componente que muestra datos básicos del perfil de un usuario de GitHub a partir de su login.


```

var GitHubUserProfile = React.createClass({ ❶
  render: function() {
    return (<div>
      <h3>{this.props.nombre}</h3>
      <img src={this.props.url_avatar}/>
    </div>)
  }
});
var GitHubUser = React.createClass({ ❷
  getInitialState: function() {
    return {
      nombre: '',
      url_avatar: ''
    };
  },
  componentDidMount: function() { ❸
    var peticion = new XMLHttpRequest();
    peticion.onreadystatechange = function() {
      if (peticion.readyState == 4) {
        var datos = JSON.parse(peticion.responseText);
        this.setState({nombre: datos.name, url_avatar:datos.avatar_url});
      }
    }.bind(this); ❹
    peticion.open('GET', 'https://api.github.com/users/'+this.props.login,
true);
    peticion.send();
    this.peticion = peticion;
  },
  componentWillUnmount: function() { ❺
    this.peticion.abort();
  },
  render: function() {
    return <GitHubUserProfile nombre={this.state.nombre}
      url_avatar={this.state.url_avatar}/>
  }
});

var instancia = <GitHubUser login='octocat'/>;
ReactDOM.render(instancia, document.getElementById('miApp'));

```

- ❶ Este es el componente que muestra los datos del perfil propiamente dicho, pero no se encarga de nada más. Por tanto otro componente tendrá que hacer la petición al API de GitHub. Nótese que siguiendo las buenas prácticas de React es un componente sin estado.
- ❷ Este componente se encarga de hacer la petición correspondiente al API de GitHub y pasarle los datos a una instancia del componente anterior. Este sí es un componente con estado, ya que no tenemos los datos del perfil hasta que recibimos la respuesta del servidor.
- ❸ En el `componentDidMount` hacemos la petición AJAX y cuando el servidor nos devuelve la respuesta en JSON guardamos los datos que nos interesan en el estado, forzando por tanto un *render*.
- ❹ Nótese el `bind` que nos hemos visto forzados a hacer en el *callback* de la petición AJAX para que `this` sea el componente y poder acceder así a su estado con `this.state`.

- 5 En el `componentWillUnmount` abortamos la petición AJAX por si estuviera pendiente, ya que si se elimina el componente ya no tiene sentido.



Podemos forzar el desmontado del componente para probar el `componentWillUnmount` ejecutando desde la consola del navegador: `ReactDOM.unmountComponentAtNode(document.getElementById('miApp'))` (suponiendo que `miApp` es el `id` del nodo del DOM donde hemos hecho el `ReactDOM.render`).

Mejora del rendimiento con `shouldComponentUpdate`

Aunque uno de los mantras de React es que "Javascript es más rápido de lo que crees", no siempre es adecuado disparar el `render` ciegamente cada vez que cambia el estado o las props o cada vez que se re-renderiza el componente "padre" y se fuerza por tanto al *re-rendering* de los hijos. En ciertas circunstancias nos podemos ahorrar el *render*.

El método `shouldComponentUpdate` del ciclo de vida nos da una oportunidad de decidir si se debería o no disparar el *render*. Recibimos como argumentos el nuevo valor para `props` y `state`, en este orden, y si devolvemos `false` indicamos que el `render` no se debe ejecutar.

7.3. React y Backbone

Vamos a ver aquí cómo conectar Backbone, con el que implementaremos la parte del modelo, con React, que nos va a dar una implementación mucho más avanzada de las vistas que las nativas de Backbone. En principio React no está preparado especialmente para trabajar junto con Backbone. Afortunadamente, React implementa una forma de *mixins*, que nos permiten compartir código Javascript entre múltiples componentes, sin tener que repetirlo. Eso nos facilita definir componentes React que incorporen las funcionalidades necesarias para trabajar de forma sencilla con modelos y colecciones de Backbone.

Hay varias implementaciones hechas por terceros de *mixins* para combinar Backbone y React. De ellas vamos a usar aquí una llamada `backbone-react-component`⁶⁹. Para usar dicho código es necesario incluir un `script JS`⁷⁰ en nuestra página.

Como dice su documentación, el *mixin* sirve de "pegamento" entre componentes React y modelos y/o colecciones de Backbone. De esta forma si tenemos por ejemplo un componente asociado a una colección y esta cambia, el *mixin* disparará el re-renderizado.

Un componente con un modelo asociado

Este es el caso más sencillo, tenemos un componente y queremos asociarle un modelo de Backbone. Al definir el componente, en el método `render` los atributos del modelo estarán accesibles a través de propiedades de `state` del mismo nombre. Por ejemplo:

```
<script type="text/babel">
  var LibroComp = React.createClass({
    mixins: [Backbone.React.Component.mixin],
    render: function() {
      return (
```

⁶⁹ <https://github.com/magalhas/backbone-react-component>

⁷⁰ <https://raw.githubusercontent.com/magalhas/backbone-react-component/master/dist/backbone-react-component-min.js>

```

        <div className="libro">
            <b>{this.state.titulo}</b>, por <em>{this.state.autor}</em>
        </div>
    );
}
});
var libro1 = new LibroModel({titulo:"Crónicas marcianas", autor: "Ray
Bradbury"});
ReactDOM.render(<LibroComp model={libro1}></LibroComp>,
    document.getElementById('un_libro'));
</script>

```

Si cambiamos el modelo, el *mixin* disparará un re-rendering automáticamente.

Además de solo a los atributos podemos acceder al modelo completo con el método `getModel()`. Así, podríamos haber implementado el `render` como:

```

...
render: function() {
    var m = this.getModel();
    return (
        <div className="libro">
            <b>{m.get('titulo')}</b>, por <em>{m.get('autor')}</em>
        </div>
    );
}
...

```

Un componente con una colección asociada

Vamos a ver el mismo ejemplo de antes de la colección de libros, pero ahora usando un modelo de Backbone para almacenar los datos de un libro y una colección para almacenar la lista de libros.

```

<script type="text/javascript">
    var LibroModel = Backbone.Model.extend({}); ❶
    var Biblioteca = Backbone.Collection.extend({
        model: LibroModel
    });
    miBiblio = new Biblioteca([
        new LibroModel({titulo: "Juego de tronos", autor: "George R.R.
Martin"}),
        new LibroModel({titulo: "El mundo del río", autor: "Philip J.
Farmer"})
    ]);
</script>
<script type="text/babel">
    var ListaLibros = React.createClass({
        mixins: [Backbone.React.Component.mixin], ❷
        render: function() {
            var libros = this.getCollection().map(function(libro) { ❸
                return (
                    <Libro autor={libro.get("autor")}>

```

```

        {libro.get("titulo")}
      </Libro>
    );
  });

  return (
    <div className="listaLibros">
      {libros}
    </div>
  );
}
});

var Libro = React.createClass({
  render: function() {
    return (
      <div className="libro">
        <b>{this.props.children}</b>, por
        <em>{this.props.autor}</em>
      </div>
    );
  }
});

React.render(
  <ListaLibros collection={miBiblio}></ListaLibros>, ❷
  document.getElementById('example')
);
</script>

```

- ❶ Definimos un modelo `Libro` y una colección `Biblioteca` usando Backbone. Este código no tiene nada de ReactJS.
- ❷ Como dice la documentación de `backbone-react-component` hay que incluir este *mixin* en el componente raíz de la jerarquía.
- ❸ El componente React tiene una colección asociada (luego veremos cómo asociarla), que es accesible mediante `getCollection()`. Como cada elemento de la colección es un modelo de Backbone usamos los `getter`s` correspondientes para acceder a los datos.
- ❹ Aquí es donde asociamos la colección de Backbone al componente de React. El *mixin* está preparado para que la propiedad que referencia a la colección se llame `collection`. Si quisiéramos asociar un modelo usaríamos una propiedad llamada `model`. En la documentación de `backbone-react-component` podemos ver [cómo asociar más de un modelo y/o colección](#)⁷¹ a un componente React.

El *mixin* que hemos usado se ocupará de que cuando cambie algún modelo de la colección el componente se redibuje automáticamente. No obstante, también podríamos gestionar manualmente la comunicación, como se hace por ejemplo en [este artículo](#)⁷². Además del *mixin* que hemos usado aquí, hay algunas otras [implementaciones alternativas](#)⁷³.

⁷¹ <https://github.com/magalhas/backbone-react-component#multiple-models-and-collections>

⁷² <http://www.thomasboyt.com/2013/12/17/using-reactjs-as-a-backbone-view.html>

⁷³ <https://github.com/clayallsopp/react.backbone>

7.4. Ejercicios de React (II)

En ambos ejercicios puedes usar JSX o bien el API JS, como prefieras.

Composición de componentes y ciclo de vida (0,5 puntos)

Crea un componente React denominado `Crono`, que debe mostrar un cronómetro indicando minutos y segundos desde que se cargó. El componente contendrá dos componentes hijos `Elemento`, uno para visualizar los minutos y otro para los segundos.

- El estado debería estar en el componente de nivel superior. Necesitamos guardar los minutos y segundos transcurridos
- En la función del ciclo de vida `componentDidMount`:
 - # Para actualizar el estado usa un temporizador JS. Recuerda que puedes crearlo con la función `setInterval`⁷⁴. No lo llames cada segundo justo, porque un *timer* puede ejecutarse con cierto retraso, ejecútalo cada menos tiempo, por ejemplo cada 500ms o menos.
 - # Puedes calcular el tiempo transcurrido desde que se cargó el componente guardando la fecha inicial en el estado y luego restando la fecha actual menos la inicial. Te dará el tiempo transcurrido en milisegundos, que luego puedes convertir a minutos y segundos.

```
[source, javascript]
----
//momento actual
var ahora = new Date();
//tiempo transcurrido en ms. Suponemos inicializado "momentoInicial"
y guardado en state
var tiempo = (ahora-this.state.momentoInicial)/1000;
this.setState({minutos:Math.round(tiempo/60),
  segundos:Math.round(tiempo%60)});
----
```

Comunicación en la jerarquía de componentes (0,75 puntos)

Crea una nueva versión del **widget** del tiempo con React que hiciste en la sesión anterior, pero ahora usa una jerarquía de 3 componentes:

- El componente `TiempoWidget`, que contendrá el código que comunica con el API del servidor y los dos componentes hijos que se ocupan de dibujar la interfaz.
- Un componente hijo que contendrá únicamente los campos de formulario. Tendrá que comunicarse con el padre llamando a un *callback*, cuando el usuario pulse sobre el botón `Ver tiempo`.
- Un componente hijo que se limitará a mostrar el tiempo en modo texto y con el icono.

Sigue las prácticas recomendadas en React en cuanto a dónde almacenar el estado: éste debería residir únicamente en el componente de nivel superior, y los hijos si necesitan mostrar datos deberían recibirlos como `props`.

⁷⁴ <https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval>

8. Introducción a la arquitectura Flux para aplicaciones React

8.1. ¿ Por qué Flux?

Flux es una alternativa a MVC propuesta por los ingenieros de Facebook y especialmente pensada para aplicaciones de tamaño mediano y grande (aunque nada impide usarla en aplicaciones pequeñas, ya que no es una arquitectura compleja).

Flux fue presentada en la conferencia de desarrolladores de Facebook, la F8, en 2014, en una charla titulada "[Hacker Way: Rethinking Web App Development at Facebook](#)"⁷⁵. La información de este apartado está adaptada de dicha charla. Además del video podéis consultar también unas [notas sobre la presentación en PDF](#)⁷⁶.

MVC es un patrón arquitectónico muy conocido y ampliamente probado, pero según los proponentes de Flux no es adecuado para aplicaciones de tamaño mediano o grande. En aplicaciones pequeñas la arquitectura MVC es clara y sencilla:

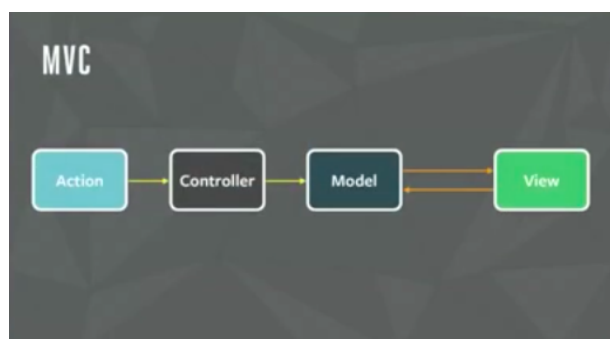


Figura 11. MVC en aplicaciones pequeñas

Sin embargo en aplicaciones de mayor tamaño, donde hay un número elevado de modelos y vistas, el diagrama se complica de manera considerable:

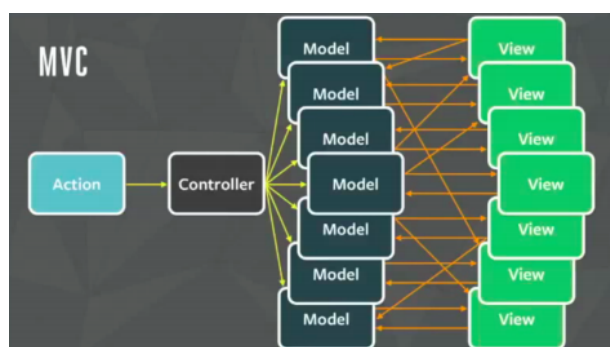


Figura 12. MVC en aplicaciones medianas o grandes

El problema fundamental aquí es la complicada relación que existe entre todos los modelos y las vistas, formando un grafo complejo y con ciclos. Cuando hay este tipo de ciclos, un *bug* en un componente puede causar problemas en muchos otros, que además serán difíciles de detectar al ser el flujo de datos tan complicado. Contra esto, se propone una arquitectura basada en un **flujo unidireccional de datos**. Este flujo unidireccional va a hacer mucho más fácil la detección de *bugs* y a mejorar la comprensión que tienen los desarrolladores del sistema.

⁷⁵ <https://www.youtube.com/watch?v=nYkdrAPrdcw>

⁷⁶ <https://drive.google.com/file/d/0B6FDE5kMwYTItdkLWEo1X05rQzA/view>

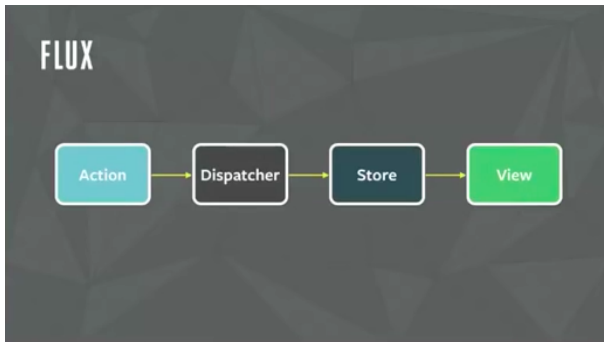


Figura 13. Flujo unidireccional de datos en Flux



La discusión anterior está contada desde el punto de vista de los ingenieros que propusieron Flux. Es posible que una aplicación MVC bien diseñada no se parezca tanto al diagrama puesto como ejemplo de "mala práctica". Parte del problema reside en que no hay un acuerdo 100% en lo que significa MVC. De cualquier modo, el argumento principal de Flux es primar el flujo unidireccional de datos con respecto a cualquier otra arquitectura que no lo tenga. En ese sentido no tiene tanta importancia que la "mala práctica" sea MVC o no, se trata solo de evitarla.

8.2. Flux a grandes rasgos

Una de las ideas principales de Flux es separar en lo posible el estado de la aplicación de los componentes de la interfaz. Ya hemos discutido en sesiones anteriores por qué nos interesa tener componentes sin estado. Idealmente tendríamos por un lado el estado de la aplicación, y cuando este estado se actualizara, de alguna forma los componentes serían notificados del cambio para que se *re-renderizaran*. Recordemos este diagrama:

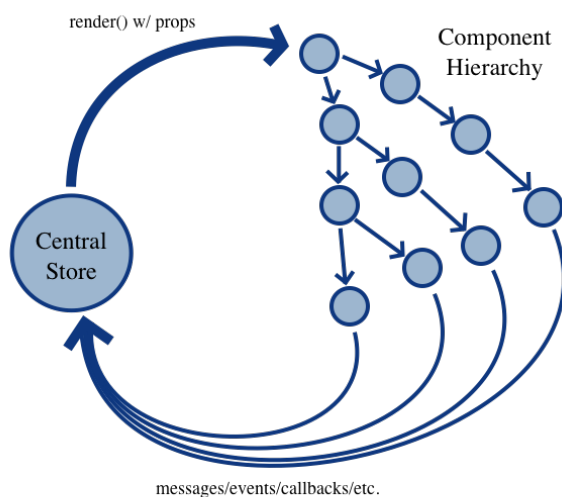


Figura 14. Flujo de datos con estado centralizado (tomada de [React Tips and Best Practices](http://aeflash.com/2015-02/react-tips-and-best-practices.html)⁷⁷)

⁷⁷ <http://aeflash.com/2015-02/react-tips-and-best-practices.html>

Stores

Al elemento que almacena el estado en una aplicación Flux se le denomina **Store**. Aunque estemos hablando en singular, puede haber varios. Las vistas (componentes de React) se suscriben a los *stores* que les interesa para recibir las notificaciones de que se han actualizado los datos.

En un paradigma MVC tradicional, los *stores* se corresponderían más o menos con los modelos. Sin embargo hay una diferencia importante: se puede acceder a los datos que contienen los *stores* (tienen *getters*) pero no se los puede cambiar directamente (no tienen *setters*). La única forma que hay de cambiar los datos de un *store* es de modo indirecto a través de **acciones**.

Actions

Las **acciones** son "cosas que suceden en la aplicación". Por ejemplo en Facebook sería que el usuario pone un *post* en el muro, o sube una foto, o le da *like* a otra,... es decir, más o menos los casos de uso, aunque pueden tener una granularidad más fina.

Cada acción tiene un *tipo* (un valor único que la identifica) y opcionalmente un *payload* con más información. Las acciones se despachan a los *stores*, y estos son los encargados de procesarlas. Cada *store* debe saber cómo actualizar los datos que contiene en respuesta a una determinada acción.

Dispatcher

El **dispatcher** es la última pieza que nos falta. Es el que se encarga de *despachar* las acciones a los *stores*. A diferencia de un sistema típico publicar/suscribir, el *dispatcher* despacha "ciegamente" todas las acciones a todos los *stores* que se han suscrito a él. Es responsabilidad del *store* procesar la acción efectivamente o no.

El *dispatcher* suele ser la pieza más genérica e intercambiable entre todas las aplicaciones Flux, así que es la que dan ya implementada la mayor parte de las librerías que implementan esta arquitectura, como componente "listo para usar".

Flujo de datos

Estos son los únicos componentes de Flux. Recapitulando, las acciones van al *dispatcher* que se encarga de enviarlas a los *stores*. Estos a su vez notifican a los componentes interesados del cambio en el estado para que se repinten. Nótese que esto genera un flujo unidireccional de datos: Acciones#Dispatcher#Stores#Componentes React, que es uno de los *mantras* de Flux. No obstante, evidentemente tiene que haber un sitio del que salgan las acciones. Típicamente las disparará el usuario al interactuar con los componentes, así que hay una retroalimentación para "cerrar el bucle" de la aplicación, como se ve en el siguiente diagrama

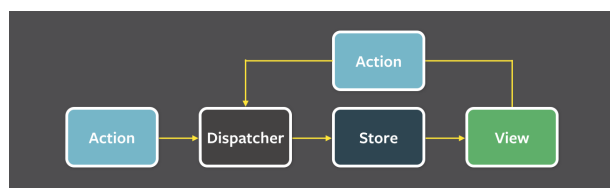


Figura 15. Flujo de datos en Flux

8.3. Un ejemplo sencillo

Vamos a ver cómo funciona Flux en una aplicación muy sencilla. Al ser un ejemplo *de juguete* evidentemente el uso aquí de Flux o de cualquier otra arquitectura no está justificado en realidad, salvo por motivos pedagógicos.

La idea es hacer una pequeña aplicación que en teoría mantenga el saldo de nuestra cuenta bancaria. El componente React debería mostrar en todo momento el saldo actual y permitir que se hagan ingresos y reintegros de la cuenta (nuestro único objetivo es que se vaya actualizando y mostrando el saldo, no hacer ingresos ni reintegros reales con ningún API).

Librerías para implementar Flux

Lo primero que debemos tener claro es que Flux es una arquitectura, al igual que MVC, y no una librería ni un *framework* concreto. Facebook tiene [su propia implementación de Flux](#)⁷⁸, que es *open source*. Al ser la "oficial" de Facebook es la más difundida. Básicamente lo que hace esta implementación es proporcionarnos un *dispatcher* listo para usar y poco más, al menos en las primeras versiones.

No obstante, a tono con la ebullición general de todo lo que tenga que ver con React, últimamente ha surgido multitud de implementaciones alternativas a la original, algo interesante teniendo en cuenta que se trata de una arquitectura que no llega a los dos años de vida: Redux, Reflux, DeLorean, Alt, ... Una vez introducidos en Flux, sería interesante echarle un vistazo a alguna de las comparativas de implementaciones, por ejemplo [esta](#)⁷⁹ o [esta](#)⁸⁰. Dado que es la más difundida usaremos la implementación de Facebook.

Casi todas las implementaciones de Flux recomiendan el uso de `npm` para su instalación junto con un *bundler* como `browserify` o `webpack` que nos permita usar en el navegador los paquetes instalados con React. Para no complicarnos la vida inicialmente con estas herramientas y centrarnos en lo que es Flux en sí, vamos a usar Flux "a la antigua usanza", es decir, incluyéndolo con una etiqueta `<script>`. No obstante, para el trabajo "más serio" con Flux una vez se ha aprendido, se recomienda encarecidamente usar estas herramientas. Su uso se describe brevemente en uno de los apéndices de los apuntes.

En los ejemplos, incluiremos Flux desde una CDN, para no tener que bajarlo localmente:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/flux/2.1.1/Flux.js"></script>
```

El *Dispatcher*

Como ya hemos comentado, el *dispatcher* es la única pieza "lista para usar" que nos da la implementación de Facebook. Aunque en el diagrama de la arquitectura no es la primera del flujo de datos, empezaremos por ella, ya que cuando creemos las acciones debemos poder enviarlas aquí y por tanto necesitamos tener ya instanciado el *dispatcher*. Para instanciar un *dispatcher*, simplemente haremos:

```
var MiDispatcher = new Flux.Dispatcher();
```

⁷⁸ <https://github.com/facebook/flux>

⁷⁹ <https://medium.com/social-tables-tech/we-compared-13-top-flux-implementations-you-won-t-believe-who-came-out-on-top-1063db32fe73#wq0efptrq>

⁸⁰ <http://pixelhunter.me/post/110248593059/flux-solutions-compared-by-example>

Las acciones

Tenemos que identificar las "cosas que pueden pasar" en nuestra aplicación. En nuestro caso es sencillo, solo hay tres: nuestra cuenta puede abrirse para empezar con 0 euros, podemos ingresar dinero y podemos sacar dinero.

Una acción no es más que un objeto Javascript típicamente con dos campos, el tipo y el *payload*. El primero es simplemente una constante que identifica la acción de manera única. El segundo contiene más información sobre la acción, en nuestro ejemplo la cantidad ingresada o sacada. Primero vamos a definir los tipos de las acciones como constantes.

```
var Ctes = Ctes || {};  
Ctes.Acciones = {  
  CUENTA_CREADA: 1,  
  ABONO: 2,  
  CARGO: 3,  
}
```

Ahora, una acción no sería más que un objeto como:

```
{  
  tipo: Ctes.Acciones.ABONO,  
  cantidad: 100  
}
```

Aunque es un elemento que no figura en la arquitectura Flux genérica, en las implementaciones se suelen usar *creadores de acciones*, que son simplemente métodos para crear acciones y que se suelen "empaquetar" en un objeto Javascript. No es que sean estrictamente necesarios, pero nos facilitan crear una acción desde el punto del código que la necesitemos. El creador de acción debe instanciar el objeto con el *tipo* y el *payload* y enviarla al *dispatcher* para que este la despache a los *stores*.

```
var Acciones = {  
  crearCuenta: function() {  
    MiDispatcher.dispatch({  
      tipo: Ctes.Acciones.CUENTA_CREADA,  
      cantidad: 0  
    });  
  },  
  depositarEnCuenta: function(cant) {  
    MiDispatcher.dispatch({  
      tipo: Ctes.Acciones.ABONO,  
      cantidad: cant  
    });  
  },  
  sacarDeCuenta: function(cant) {  
    MiDispatcher.dispatch({  
      tipo: Ctes.Acciones.CARGO,  
      cantidad: cant  
    });  
  }  
}
```

El código anterior lo hemos implementado simplemente porque ahora desde cualquier punto es muy sencillo crear una acción y enviarla al *dispatcher*, por ejemplo:

```
Acciones.crearCuenta();
Acciones.depositarEnCuenta(100);
```

Los stores

Nuestra aplicación es tan sencilla que solo crearemos uno, pero en una aplicación más real normalmente habría varios. Los *stores* almacenan el estado y también la lógica para actualizar ese estado. Veamos una primera versión.

```
var CuentaStore = {
  balance: 0, ❶
  getBalance: function() { ❷
    return this.balance;
  },
  manejarAccion: function(accion) { ❸
    switch(accion.tipo) {
      case Ctes.Acciones.CUENTA_CREADA:
        this.balance = 0;
        break;
      case Ctes.Acciones.ABONO:
        this.balance += accion.cantidad;
        break;
      case Ctes.Acciones.CARGO:
        this.balance -= accion.cantidad;
        break;
    }
  }
}
```

- ❶ Aquí almacenamos el estado, en nuestro caso nos basta con el balance actual de la cuenta. Este estado deberíamos intentar hacerlo privado, aunque aquí lo hemos dejado como una propiedad convencional para simplificar.
- ❷ Aquí tenemos un *getter* para obtener el valor actual del estado.
- ❸ Aquí agrupamos la lógica que actualiza el estado en función de las posibles acciones. Nos falta conectar el *store* por un lado con el *dispatcher* y por otro con los componentes React. Para hacer lo primero tenemos que registrar el *dispatcher* con la función del *store* que gestiona las acciones. Hacemos:

```
MiDispatcher.register(CuentaStore.manejarAccion.bind(CuentaStore));
```

Una vez que se haya ejecutado la lógica de gestión de la acción, el *store* habrá actualizado su estado y debe notificar a las vistas interesadas que el estado ha cambiado. Así las vistas podrán obtener el nuevo estado y repintarse. Vamos a hacer esta notificación mediante eventos, al modo en que se comunican modelos y vistas en Backbone.

Los navegadores ofrecen soporte para gestionar los eventos típicos del DOM (*click*, *mouseover*, *load*, ...) pero no podemos crear eventos personalizados. Necesitamos alguna

librería adicional que nos permita generar y escuchar eventos propios. Facebook ofrece una librería propia de tipo *open source* llamada `fbemitter`⁸¹. Nosotros usaremos otra distinta denominada `EventEmitter`⁸², simplemente porque la de Facebook está pensada para instalarla con `npm`, que no estamos usando aquí por el momento. El API de ambas es prácticamente igual. Podemos incluir esta última librería simplemente con un *tag* `<script>`:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/EventEmitter/4.3.0/EventEmitter.min.js"></script>
```

La librería nos proporciona una "clase" `EventEmitter` con capacidad para generar eventos arbitrarios y registrar *listeners* para esos eventos. Para añadir esas capacidades a un objeto de nuestra aplicación (el *store*) podemos usar el `Object.assign` de Javascript. Cambiamos la primera línea donde definíamos el `CuentaStore` por la siguiente:

```
var CuentaStore = Object.assign({}, EventEmitter.prototype, {  
  //propiedades del objeto  
});
```

Lo que estamos haciendo es crear el `CuentaStore` a partir de un objeto vacío, `{}`, y sobre él estamos copiando las propiedades de `EventEmitter.prototype`, y después las propiedades que definíamos antes (`balance`, `getBalance`, ...). Este prototipo contiene los métodos que necesitamos para emitir y escuchar eventos (podéis consultar el funcionamiento de `Object.assign`⁸³ si tenéis dudas).

Lo que tenemos que hacer es emitir un evento cuando cambie el estado del *store* (en nuestro caso el `balance`, que es la única variable de estado). Primero definiremos una constante para representar dicho evento:

```
Ctes.Eventos = {  
  CAMBIO_CUENTA: 'cambio_cuenta'  
}
```

usamos `cambio_cuenta` como nombre "real" del evento ya que es típico que los eventos tengan un nombre y no un valor numérico, aunque también funcionaría con este último.

Ya solo nos falta emitir el evento tras el cambio de estado. Lo haremos al final del `manejarAccion` del *store*:

```
manejarAccion: function(accion) {  
  switch(accion.tipo) {  
    case Ctes.Acciones.CUENTA_CREADA:  
      this.balance = 0;  
      break;  
    case Ctes.Acciones.ABONO:  
      this.balance += accion.cantidad;  
      break;  
    case Ctes.Acciones.CARGO:
```

⁸¹ <https://www.npmjs.com/package/fbemitter>

⁸² <https://github.com/Olical/EventEmitter>

⁸³ https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/assign

```

    this.balance -= accion.cantidad;
    break;
  }
  this.emitEvent(Ctes.Eventos.CAMBIO_CUENTA);
}

```

A este evento lo estarán escuchando una o más vistas, es decir, uno o más componentes de React. Vamos a ver ya el último paso de Flux: cómo detectan los componentes React el cambio de estado y se repintan automáticamente.

Los componentes React

Hemos transformado el problema de detectar el cambio de estado en el de escuchar el evento `Ctes.Eventos.CAMBIO_CUENTA`. Usando la librería `EventEmitter` esto es muy sencillo, simplemente se trata de registrar una función que actúe de *listener* del evento.

Recordemos que en React el *re-rendering* es automático cada vez que se cambia el estado de un componente con `setState`, y este cambio va a ser el que vamos a hacer cuando recibamos el evento de cambio de estado del *store*. Es decir, la idea es que los cambios de estado del *store* disparen cambios de estado en los componentes.

Recordemos también que una buena práctica en React es que el máximo posible de componentes sean sin estado y solo usen `props`. Pero entonces, si solo tenemos `props` y no estado, ¿cómo vamos a disparar el *re-rendering*? La solución es que por cada árbol de componentes (por cada componente que contenga a otros) tendremos un componente adicional de "nivel superior" que en lugar de tener parte propia de interfaz en realidad actúa como "controlador". Almacena el estado, escucha el evento de cambio y actualiza su estado con la información del *store*, disparando así el *re-rendering* de los componentes de niveles inferiores.

En nuestro ejemplo vamos a implementar un componente React llamado `CajeroController` que hará este papel:

```

var CajeroController = React.createClass({
  componentDidMount: function() { ❶

    CuentaStore.addListener(Ctes.Eventos.CAMBIO_CUENTA, this.cambiaStoreListener)
  },
  componentWillUnmount: function() { ❷

    CuentaStore.removeListener(Ctes.Eventos.CAMBIO_CUENTA, this.cambiaStoreListener)
  },
  cambiaStoreListener: function() { ❸
    this.setState({balance: CuentaStore.getBalance()});
  },
  getInitialState: function() { ❹
    Acciones.crearCuenta();
    return {balance: CuentaStore.getBalance()};
  },
  render: function() { ❺
    return <Cajero balance={this.state.balance}/>;
  }
});

```

```
var instancia = <CajeroController />;
window.componente = ReactDOM.render(instancia,
  document.getElementById( 'miApp' ));
```

- 1 Cuando el componente se monta en el DOM empezamos a escuchar el evento que genera el *store*.
- 2 Al desmontar el componente aprovechamos para eliminar el *listener* del evento.
- 3 El *listener* lo que hace simplemente es actualizar el estado del componente con la información que viene del *store*. En este sencillo ejemplo, componente y *store* almacenan la única variable de estado que hay, pero en un ejemplo más complejo el *store* podría tener otras variables de estado que no nos interesaran aquí. Recuérdese que el `setState` además de cambiar el estado dispara el *re-rendering* del componente, y de los componentes hijos. Con esto hemos conseguido que cada vez que se actualice el *store* se actualice también el componente.
- 4 Aquí vemos un ejemplo de "flecha en la otra dirección" en Flux. Al inicializar el estado del componente, creamos la cuenta. Nótese que no podemos (o no debemos) manipular el *store* directamente. Lo que hacemos es disparar una acción que a su vez será la que actualice el estado del *store*.
- 5 Como vemos el componente que actúa de controlador no tiene HTML por sí mismo, simplemente tiene un componente hijo al que le pasa su estado pero en forma de `props`. Así conseguimos eliminar el estado del resto de componentes y simplificar su comportamiento.

Vamos a ver el componente `Cajero` que es el que muestra la interfaz de usuario propiamente dicha:

```
var Cajero = React.createClass({
  depositar: function() {
    Acciones.depositarEnCuenta(Number(this.refs.cantidad.value));
  },
  reintegrar: function() {
    Acciones.sacarDeCuenta(Number(this.refs.cantidad.value));
  },
  render: function() {
    return (
      <div>
        <h1>{this.props.balance}</h1>
        <input type="text" ref="cantidad"/>
        <button onClick={this.depositar}>Depositar</button>
        <button onClick={this.reintegrar}>Reintegrar</button>
      </div>
    )
  }
});
```

El componente lo primero que hace es mostrar el balance de la cuenta, que él ha recibido del componente "controlador" como un `prop` (para evitar tener que guardar su propio estado). Luego tenemos un cuadro de texto en el que tecleamos una cantidad y unos botones para "depositar" esta cantidad en la cuenta o hacer el reintegro de la misma. Esos botones están vinculados con métodos que lo que hacen es disparar las correspondientes acciones de `depositarEnCuenta` y `reintegrarEnCuenta`, que a su vez serán despachadas al *store*, que a su vez actualizará su estado y avisará al componente "controlador", cerrando así un nuevo ciclo de Flux.

Coordinación entre *stores*

Hasta ahora en el ejemplo solo tenemos un *store*, pero vamos a ver qué sucede si introducimos uno adicional. Por poner un caso simple, supongamos que el banco hace una promoción en la que da distintos regalos a los clientes según el saldo que tengan en la cuenta. Crearemos un *store* para almacenar los datos del regalo que debería recibir el cliente con el saldo actual. Para simplificar almacenaremos solo el nombre del regalo. Cada vez que se cambie el saldo de la cuenta este se actualizará. Nótese que para que esto funcione, *primero* se debe actualizar el saldo y *después* el regalo. Es decir, debe haber una coordinación entre *stores*.

Al procesar una acción en un *store* podemos especificar que primero tienen que haberla procesado otros *stores*. Para ello se usa el método `waitFor` del *dispatcher*, al que se le pasa un array con los "identificadores" o *dispatch token* de los *stores* a los que hay que esperar. Este método se "bloqueará" hasta que hayan procesado la acción. El *dispatch token* de un *store* se obtiene como resultado de la llamada al método `register` del *dispatcher* (hasta el momento habíamos ignorado este valor de retorno). Es decir, al registrar un *store* con el *dispatcher*, haremos algo como:

```
CuentaStore.dispatchToken =
  MiDispatcher.register(CuentaStore.manejarAccion.bind(CuentaStore));
```

Vamos a ver el código del nuevo *store*, al que llamaremos `PromocionStore`:

```
var PromocionStore = Object.assign({}, EventEmitter.prototype, {
  promocion: '',
  getPromocion: function() {
    return this.promocion;
  },
  manejarAccion: function(accion) {
    MiDispatcher.waitFor([CuentaStore.dispatchToken]);
    if (accion.tipo==Ctes.Acciones.ABONO ||
    accion.tipo==Ctes.Acciones.CARGO) {
      var saldo = CuentaStore.getBalance();
      if (saldo>0 && saldo<1000)
        this.promocion = 'Taza de Hello Kitty';
      else if (saldo>1000)
        this.promocion = 'Vajilla completa';
    }
    else if (accion.tipo==Ctes.Acciones.CUENTA_CREADA) {
      this.promocion = '';
    }
    this.emitEvent(Ctes.Eventos.CAMBIO_PROMOCION);
  }
});
```

Lo primero que hacemos al manejar la acción es esperar a `CuentaStore`. Una vez que se ha actualizado su estado, ya podemos usar el balance de la cuenta para determinar el regalo al que tiene derecho el cliente. Una vez terminado el procesamiento de la acción, emitimos un nuevo evento: `Ctes.Eventos.CAMBIO_PROMOCION` (que debemos haber definido en nuestro código, aunque aquí no mostramos la definición).

Tendremos que modificar el componente React `CajeroController` para que escuche el nuevo evento y actualice su estado en respuesta a él:

```
var CajeroController = React.createClass({
  componentDidMount: function() {

    CuentaStore.addListener(Ctes.Eventos.CAMBIO_CUENTA, this.cambiaStoreListener)

    PromocionStore.addListener(Ctes.Eventos.CAMBIO_PROMOCION, this.cambiaStoreListener)
  },
  componentWillUnmount: function() {

    CuentaStore.removeListener(Ctes.Eventos.CAMBIO_CUENTA, this.cambiaStoreListener)

    PromocionStore.removeListener(Ctes.Eventos.CAMBIO_PROMOCION, this.cambiaStoreListener)
  },
  cambiaStoreListener: function() {
    this.setState({balance: CuentaStore.getBalance(), regalo:
PromocionStore.getPromocion()});
  },
  getInitialState: function() {
    Acciones.crearCuenta();
    return {balance: CuentaStore.getBalance(), regalo:
PromocionStore.getPromocion()};
  },
  render: function() {
    return <Cajero balance={this.state.balance}
regalo={this.state.regalo}/>;
  }
});
```

Al montar el componente tenemos que añadir un nuevo *listener* para el nuevo evento. Podríamos haber usado un *listener* para cada evento, pero para simplificar hemos usado el mismo que ya teníamos, solo que ahora ampliamos el estado con una nueva variable para almacenar el regalo que se le puede hacer al cliente. El componente de interfaz `Cajero` admite una nueva `prop` con esta información.

8.4. Ejercicio de Flux (1,25 puntos en total)

Rediseña (¡cómo no!) el *widget* del tiempo para usar la arquitectura Flux.

Acciones (0,4 puntos)

Lo primero que debes hacer es identificar las acciones de tu aplicación. A primera vista la única acción que habría sería la de "Actualizar el tiempo", disparada cuando el usuario pulsa sobre el botón de "Ver tiempo". No obstante esta acción comunica de forma asíncrona con el servidor, por lo que necesitaremos dos acciones más para representar la respuesta de éste (respuesta OK con los datos del tiempo a mostrar, y respuesta de error).

- Define las constantes que creas apropiadas para representar las tres acciones.
- Implementa el *creador de acciones*. Al crear la acción de "actualizar el tiempo" es cuando se hará la petición al API del servidor, y desde el *callback* de la petición es desde donde se llamará a las otras dos acciones.

Para comprobar el caso en que el servidor devuelve un error lo más sencillo es poner la localidad vacía, ya que si se escribe una cadena arbitraria el API buscará un nombre similar en lugar de devolver un error.

Stores (0,4 puntos)

Implementa el `TiempoStore`, que como propiedades debe tener como mínimo la "descripción" del tiempo (sol, nubes,...) y la url del icono que lo representa. Cuando se actualice el tiempo con éxito o se actualice con error debes generar los eventos apropiados.

Componente React (0,45 puntos)

Modifica el componente React de la sesión anterior para que tome los datos de `TiempoStore`, escuchando los eventos correspondientes, y para que al pulsar en el botón de `Ver tiempo` se llame ahora a la acción correspondiente.

9. Apéndice: Herramientas para gestionar el flujo de trabajo en el desarrollo *frontend*

Durante mucho tiempo la forma habitual de usar librerías Javascript en una aplicación ha sido ir a la web de la librería, bajarse el `.zip` con la última versión e incluir la librería y las dependencias con etiquetas `<script src="">`. Sin embargo esta ya no es la manera más común de trabajar en el lado del servidor desde hace algún tiempo. Las librerías no se suelen bajar manualmente de la web sino que se usan herramientas como Maven para gestionar automáticamente las dependencias y generar una plantilla para no tener que partir de cero. Con el aumento de la complejidad de las aplicaciones en el lado del cliente también ha surgido un conjunto de herramientas para gestionar más o menos las mismas cosas que podemos gestionar con Maven.

Aunque las herramientas del lado del cliente todavía no están tan maduras como las del lado del servidor, han surgido algunas que se han ido imponiendo como estándares "de facto". Vamos a instalar aquí tres de ellas, que iremos usando a lo largo de los ejercicios de la asignatura.



La variedad y complejidad de las herramientas de desarrollo para *frontend* ha "explotado" en los últimos tiempos, para dar soporte a los cada vez más complejos flujos de trabajo del proceso de desarrollo en el cliente. Como información adicional sobre otras (muchas) herramientas existentes podéis consultar [estas transparencias](#)⁸⁴ de Addy Osmani o echarle un vistazo a [esta playlist](#)⁸⁵ de YouTube con interesantes charlas sobre el tema.

Muchas herramientas de *frontend* están implementadas en Javascript (¿Qué mejor que una herramienta en Javascript para trabajar con aplicaciones Javascript?). Y la mayoría de las implementadas en este lenguaje usan [Node.js](#)⁸⁶ como soporte, básicamente porque es un intérprete JS que puede realizar operaciones que son necesarias para una herramienta de desarrollo pero que no se pueden hacer desde el navegador, como escribir en el sistema de archivos local.

En la máquina virtual del curso ya está instalado `Node.js` junto con su gestor de paquetes, `npm`. Usaremos este último para instalar las tres herramientas de desarrollo que vamos a necesitar: [Yeoman](#)⁸⁷, [Bower](#)⁸⁸ y [Grunt](#)⁸⁹ (las dos últimas son dependencias de la primera).



En principio las herramientas habría que instalarlas en modo superusuario. Son paquetes de `npm` que se instalan en modo global `-g` para que estén disponibles desde cualquier directorio, y por defecto esto instalaría archivos en directorios del sistema. Una alternativa es cambiar el `prefix` de `npm` para que instale siempre los paquetes en el directorio del usuario. [Esta alternativa es la recomendada](#)⁹⁰ por muchos desarrolladores, por ser más segura.

⁸⁴ <https://speakerdeck.com/addyosmani/front-end-tooling-workflows>

⁸⁵ https://www.youtube.com/playlist?list=PLMh6HwjXBltUZ4IixGltJ_pSG6NN3mlQr/

⁸⁶ <http://nodejs.org/>

⁸⁷ <http://yeoman.io/>

⁸⁸ <http://bower.io/>

⁸⁹ <http://gruntjs.com/>

⁹⁰ <https://github.com/sindresorhus/guides/blob/master/npm-global-without-sudo.md>



para poder instalar globalmente paquetes de `npm` de forma sencilla sin privilegios de superusuario puedes ejecutar primero [este script](#)⁹¹ (con [repositorio](#)⁹² en Github). Irónicamente, lo primero que hace el *script* es pedir permisos de superusuario. Luego cambiará el prefijo de la instalación de paquetes npm y nos pedirá permiso para modificar el `.bashrc` para que `npm` tenga en cuenta el nuevo prefijo a partir de ahora.

Para instalar yeoman, abrir una terminal y teclear:

```
npm install -g yo bower grunt-cli
```

Tras un rato en el que se instalarán unos cuantos paquetes de Node, si todo ha ido bien podremos empezar a trabajar con las herramientas. En la documentación de Yeoman hay una imagen bastante ilustrativa de la relación entre las tres y el papel que desempeña cada una.

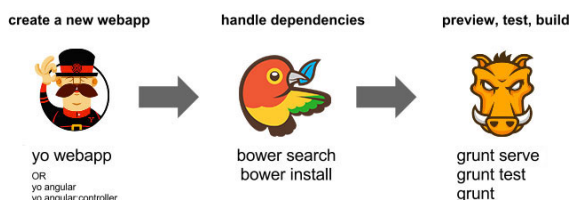


Figura 16. Flujo de trabajo con Yeoman, Bower y Grunt (página original)⁹³

- **Grunt** es una herramienta para automatizar tareas repetitivas. La podríamos asimilar al `make` de C (o al "antiguo" `ant` de Java).
- **Bower** es un gestor de dependencias entre paquetes. Con él podemos bajarnos una determinada versión de una librería Javascript y automáticamente todas sus dependencias.
- **Yeoman** es un generador de plantillas para no tener que partir de cero cada vez que comencemos una nueva aplicación web. Haría más o menos el mismo papel que hacen los arquetipos en Maven. Depende de `Bower` y `Grunt` (o mas genéricamente, depende de un gestor de paquetes y de un sistema de automatización, es configurable para trabajar con otros, por ejemplo `npm` y `gulp` respectivamente).

Por el momento vamos a dejar a Grunt un poco apartado y vamos a ver cómo trabajar con las otras dos herramientas a nivel básico.

9.1. Gestión de paquetes con Bower

Bower facilita la tarea de bajarse librerías junto con sus dependencias. Al igual que con Maven, hay un registro centralizado de "artefactos" que especifica las relaciones de dependencia. Podemos buscar librerías desde línea de comandos con `search`. Por ejemplo, podemos teclear en la terminal

```
bower search backbone
```

⁹¹

https://raw.githubusercontent.com/glenpike/npm-g_nosudo/master/npm-g-no-sudo.sh

⁹² https://github.com/glenpike/npm-g_nosudo

⁹³ <http://yeoman.io/learning/index.html>

para ver todos los paquetes que contienen `backbone` en el nombre (que como podemos ver, son muchos). Para bajarse una librería usamos el comando `install`:

```
bower install backbone
```

Este comando nos instalará `backbone` y sus dependencias directas (`underscore`). Lo que hace es bajárselo a un directorio llamado `bower_components`.



Bower se baja las dependencias, pero el cómo las usemos en nuestro proyecto ya es cosa nuestra. Nos tocará incluir los `.js` manualmente con las típicas `<script src="">`. Alternativamente también podemos usar herramientas que pueden hacer esto por nosotros, como Yeoman.

9.2. Creación de plantillas con Yeoman

Con yeoman podemos generar la estructura básica de nuestra aplicación, para no tener que partir de cero. Como ya hemos dicho es algo similar a los arquetipos de Maven.

Para poder crear una plantilla de aplicación que use una determinada tecnología (Backbone, Angular, Bootstrap,...) necesitamos que alguien haya desarrollado un *generador*. El [repositorio de Yeoman](#)⁹⁴ tiene un gran número de ellos, y por supuesto también podríamos definirlo nosotros.

Por ejemplo el generador básico para aplicaciones backbone se llama "[generator-backbone](#)"⁹⁵. Lo instalamos con

```
npm install -g generator-backbone
```

Una vez instalado el generador, podemos generar una plantilla de aplicación Backbone sin más que ejecutar

```
yo backbone [nombre-de-la-aplicación]
```

9.3. npm + *bundler*

Una opción que se está popularizando enormemente en los últimos tiempos es usar directamente `npm` como gestor de paquetes también para el *frontend*. Este sistema de módulos es muy sencillo: cuando vamos a referenciar uno lo hacemos con la instrucción `require`. Esta instrucción nos devuelve un valor que para la mayoría de módulos es un objeto que encapsula el API.

Veamos un ejemplo sencillo de uso: por ejemplo el módulo `detect-browser` nos devuelve el nombre y versión del navegador en que estamos. Lo primero es instalar el paquete con npm

```
npm install --save detect-browser
```

y luego en nuestro código haríamos algo como:

⁹⁴ <http://yeoman.io/generators/>

⁹⁵ <https://github.com/yeoman/generator-backbone>

```
var navegador = require('detect-browser');
console.log(navegador.name);
console.log(navegador.version);
```

El problema es que este código no es usable directamente desde el navegador, ya que por desgracia los navegadores no ofrecen soporte nativo para CommonJS. Y aquí es donde intervienen unas herramientas denominadas genéricamente *bundlers*. El *bundler* toma nuestro código y los módulos referenciados por él con `require` y los une en un único archivo JS (un *bundle*), que contiene todo el Javascript necesario y que sí es usable directamente desde el navegador.

En la actualidad los *bundler* más usados son [browserify](#)⁹⁶ y [webpack](#)⁹⁷. Este último está ganando bastante tracción últimamente, aunque es un poco más complicado de usar, por lo que vamos a ver un ejemplo sencillo pero completo de código React usando `browserify`.

Uso de `browserify`

Lo primero es **instalar el propio** `browserify`, que se distribuye en forma de paquete `npm`. Es una herramienta en línea de comandos, por lo que la instalaremos con el `switch -g`:

```
npm install -g browserify
```

Vamos a usarlo con React. Lo primero es instalar los paquetes necesarios con npm:

```
npm install --save react react-dom
```

y luego escribir el código React en un archivo `ejemplo.js`

```
var React = require('react');
var ReactDOM = require('react-dom');

ReactDOM.render(
  React.DOM.h1(null, '¡Hola React!'),
  document.getElementById('miApp')
);
```

Ahora tenemos que transformar el `ejemplo.js` junto con todo el código de React que referencia en un único archivo `.js` que podemos llamar `bundle.js`. Este archivo podemos generarlo en línea de comandos con `browserify`:

```
browserify ejemplo.js -o bundle.js
```

Y este `bundle.js` ya podemos incluirlo en nuestro HTML al "modo clásico", porque ya es usable desde el navegador:

...

⁹⁶ <http://browserify.org>

⁹⁷ <https://webpack.github.io>

```
<div id="miApp"></div>
<script src="bundle.js"></script>
...
```

Un problema que se nos va a plantear al cargar el `bundle.js` en el navegador en lugar del archivo original es que se complica mucho la detección de errores y la depuración, ya que `browserify` no detecta los errores en el código, se detectan al cargarlos en el navegador, y por tanto los mensajes de error se referirán a las líneas del archivo `bundle.js`. Tampoco podremos usar el depurador integrado en el navegador para depurar el código original.

Para solucionar esto podemos decirle a `browserify` que genere un *source map*, que es un archivo que mapea la relación entre el código original y el resultante, permitiéndonos depurar sobre el código original. Basta con pasarle el *switch* `-d` a `browserify`:

```
browserify -o bundle.js -d ejemplo.js
```

Otro problema es la incomodidad de tener que estar ejecutando manualmente la orden anterior cada vez que modificamos el código. Para solucionarlo existe una serie de herramientas que detectan cambios automáticamente en el fichero original y generan el *bundle* automáticamente. La más conocida es `watchify`. Se instala con `npm` (usando `-g`, ya que instala una herramienta en línea de comandos) y su uso es muy sencillo, simplemente hay que reemplazar `browserify` por `watchify`, y dejar la orden ejecutándose permanentemente:

```
watchify -o bundle.js -d ejemplo.js
```

browserify y babel

En el tema de React hemos visto que para usar JSX tenemos que emplear una librería denominada `babel`, que se encarga de traducir (o *transpilar*) el código JSX a código Javascript nativo. Cuando explicábamos el tema hacíamos la transformación en el propio navegador, pero esto presenta dos problemas: por un lado, la transpilación tiene un coste en tiempo, lo que no es apropiado en aplicaciones en producción. Además tenemos el mismo problema que al hacer *bundling*: se dificulta la depuración y la detección de errores. Por esto se suele recomendar que la transpilación la haga alguna herramienta antes de cargar el código en el navegador, y dejar la transformación en el navegador para pruebas rápidas.

Primero necesitamos instalar un par de paquetes `npm` adicionales:

```
npm install --save babelify babel-preset-react
```

Luego debemos crear un fichero de configuración para Babel llamado `.babelrc` en el directorio del proyecto. El contenido del fichero depende del uso que estemos haciendo de Babel, en nuestro caso lo usamos para React, así que contendría lo siguiente:

```
{ "presets": ["react"] }
```

Finalmente debemos pasarle a `browserify` el *switch* `-t babelify` para que se haga la transpilación:

```
browserify -t babelify -d ejemplo.jsx -o bundle.js
```

10. Apéndice: *testing* con Backbone

Backbone es *agnóstico* en cuanto a qué *framework* de *testing* usar. Nosotros emplearemos Jasmine, que es bastante sencillo de usar ya que está "autocontenido" (incluye el *test runner*, las aserciones e incluso métodos para generar *mocks*). No obstante puedes encontrar muchos libros y tutoriales sobre Backbone que usan *frameworks* de *testing* con otra filosofía más de "juntar las piezas", como por ejemplo [Mocha](#)⁹⁸. Para usar esta última necesitamos además una librería adicional que implemente las aserciones, otra para generar *mocks*, etc.

Jasmine es una herramienta de *testing* que sigue el paradigma BDD (Behavior Driven Development), y como tal usa la terminología habitual en este paradigma, un poco diferente de la habitualmente usada en las pruebas unitarias "clásicas".

10.1. Introducción a Jasmine

Suites y casos de prueba

Al igual que en cualquier herramienta de tipo xUnit, las pruebas se escriben como **casos de prueba** y estos se agrupan en **suites**. No obstante la sintaxis es algo distinta a la tradicional en xUnit.

Para empezar, las pruebas no se suelen llamar *tests* sino *specs* (de "especificaciones"). Así, es habitual colocar el código de prueba en archivos con sufijo `spec.js`, en lugar del que sería más "tradicional" `test.js`.

Las *suites* se definen con `describe`, seguido de una cadena con la descripción de la *suite* y una función que encapsula todo su código. Cada caso de prueba (cada *spec*, por seguir la terminología habitual) se define de manera similar, usando la palabra `it`.

```
describe('Préstamo de libros', function() {
  it('Un libro recién creado no debería estar prestado', function() {
    ...
  });
  it('Al prestar un libro debería dejar de estar disponible', function() {
    ...
  });
  ...
});
```

Como vemos, la idea de esta estructura es que quede clara cuál es la intención de cada *suite* y de cada caso de prueba. Las "etiquetas" de texto de `describe` e `it` sustituyen a los nombres de los métodos de *test* en xUnit, que si queremos que sean descriptivos resultan engorrosos (`testLibroRecienCreadoNoDeberiaEstarPrestado`).

Las *suites* de pruebas pueden contener a su vez otras *suites*.



en algunos casos puede que tengamos una *spec* a medio crear y necesitemos ejecutar las pruebas. En lugar de comentarla para que no dé error, podemos ponerle una `x` delante al `it` (cambiarlo por `xit`). No se ejecutará, y en el informe de ejecución de Jasmine se marcará la prueba como pendiente. Podemos hacer lo propio con una *suite* al completo (`xdescribe`).

⁹⁸ <https://mochajs.org>

Expectativas y *matchers*

En el mundo *xUnit* las comprobaciones sobre el código se suelen hacer con `assert`. En cambio en BDD se suele usar la forma `expect` (que indica que esperamos determinado resultado, o que se cumpla determinada condición). Los partidarios de esta sintaxis defienden que mejora la legibilidad de las pruebas al hacer la sintaxis más similar a la del lenguaje natural.

Las expectativas se construyen con `expect` sobre una expresión, que es el valor real que queremos comprobar. El `expect` se encadena con el valor deseado a través de un *matcher*. El más sencillo es el de igualdad, `toBe`, equivalente a comprobar si el valor real es `==` al deseado.

```
it("Prueba de ser o no ser", function() {
  a = true;
  expect(a).toBe(true);
  expect(a).not.toBe(false);
});
```

Como vemos en el ejemplo, `not` se puede usar antes de cualquier *matcher* para invertir el sentido.

Jasmine tiene un amplio conjunto de *matchers* para comprobar si dos valores primitivos son iguales (el `toBe` que ya hemos visto), si lo son dos objetos (`toEqual`), si una cadena encaja con una expresión regular (`toMatch`), si un valor es `undefined` (`toBeUndefined`), o `null` (`toBeNull`), si un array contiene un valor (`toContain`),... La documentación de Jasmine contiene [numerosos ejemplos](#)⁹⁹.

Configuración de cada prueba

Podemos ejecutar código para preparar las pruebas, bien antes de la suite (`beforeAll`) o bien antes de cada prueba (`beforeEach`). Igualmente podemos ejecutar código de "limpieza" cuando acabe la suite (`afterAll`) o después de cada prueba (`afterEach`).

Ejecutar las pruebas

Podemos bajar un .zip con la versión actual de Jasmine de la página con las [releases](#)¹⁰⁰, del [repositorio](#)¹⁰¹ en Github.

Al descomprimirlo veremos en la raíz un archivo `specRunner.html`. Es una plantilla que nos puede servir de base para ejecutar nuestras propias pruebas. Básicamente en el *runner* tenemos que cargar varias cosas:

- La propia librería Jasmine
- Los *plugins* o librerías auxiliares para *testing* con Jasmine que estemos usando
- Nuestro código fuente
- Las *specs* que queramos ejecutar

En nuestro caso, el JS incluido en el *spec runner* sería algo como:

⁹⁹ http://jasmine.github.io/2.1/introduction.html#section-Included_Matchers
¹⁰⁰ <https://github.com/jasmine/jasmine/releases>
¹⁰¹ <https://github.com/jasmine/jasmine>

```

...
<!-- Jasmine (luego iremos añadiendo plugins) -->
<script src="lib/jasmine-2.2.0/jasmine.js"></script>
<script src="lib/jasmine-2.2.0/jasmine-html.js"></script>

<!-- código fuente a probar, y librerías de las que depende... -->
<script src="../lib/jquery.js"></script>
<script src="../lib/underscore-min.js"></script>
<script src="../lib/backbone-min.js"></script>
<script src="../tiempo.js"></script>

<!-- specs... -->
<script src="spec/modelo_spec.js"></script>
<script src="spec/vista_spec.js"></script>
...

```

10.2. Pruebas con Jasmine en Backbone

En realidad las pruebas en Backbone no se diferencian demasiado de las de otros tipos de código, pero sí es verdad que por los patrones que se suelen usar en aplicaciones Backbone hay ciertos "casos de uso típicos" para las pruebas. Vamos a ver algunos de ellos.

Usaremos como hilo conductor de los ejemplos el *widget* del tiempo que vimos en la primera sesión, aunque ligeramente modificado para complicarlo un poco.

Pruebas de lógica de negocio

Una de las ventajas fundamentales de usar un *framework* MVC como Backbone es que nos hace separar modelo y vista. Entre otras cosas esto nos va a facilitar los *tests* de lógica de negocio, que básicamente tendrán que tratar únicamente con modelos y colecciones.

Las pruebas "puras" de lógica de negocio no tienen nada de particular, simplemente usamos el API de Jasmine para formular expectativas sobre el código:

```

it("Un modelo recién creado no tiene localidad asignada", function () {
    expect(new DatosTiempo().has("localidad")).toBeFalsy();
});

```

Pruebas sobre HTML

Una de las cosas de las que hay que asegurarse en una vista es que genera el HTML correcto. Más que comprobar si el HTML es literalmente igual a una cadena de referencia en general será más sencillo simplemente comprobar si contiene determinados elementos. Podemos usar un *plugin* llamado `jasmine-jquery`¹⁰² para facilitar esta tarea. Este *plugin* define un *gran número de matchers*¹⁰³ con los que podemos chequear de manera sencilla el contenido del HTML usando selectores de jQuery.

Por ejemplo, vamos a **comprobar que el widget genera correctamente el HTML en su estado inicial**. Podemos ver que los *matchers* de `jasmine-jquery` son bastante autoexplicativos.

¹⁰² <https://github.com/velesin/jasmine-jquery>

¹⁰³ http://jasmine.github.io/2.2/introduction.html#section-Included_Matchers

```

it("El HTML generado debe ser correcto", function() {
  vista = new TiempoWidget({model: new DatosTiempo()});
  vista.render();
  expect(vista.$el).toContainElement('#localidad');
  expect(vista.$('#descripcion')).toBeEmpty();
  expect(vista.$('#ver_tiempo')).toHaveValue('Ver tiempo');
  expect(vista.$('#icono')).toHaveAttr("src", "");
});

```

Una ventaja de las vistas de Backbone es que son *autocontenidas*, es decir, que el HTML se genera dentro del `el` y que para comprobar que es correcto no es necesario insertar la vista en el DOM de la página. De este modo no tenemos que tocar el HTML del *runner* de los test para probar la parte de la interfaz.



Otra funcionalidad interesante de jasmine-jquery es la posibilidad de definir *fixtures* de HTML, es decir, fragmentos de HTML que necesitamos que estén presentes en la página actual para que interactúen con nuestro código. Así podríamos probar no solo el funcionamiento interno de la vista sino también el del código que la inserta en el lugar apropiado del DOM. Las *fixtures* se cargan desde ficheros independientes y se limpian automáticamente con cada *spec*, para no ir "ensuciando" la página con el *runner* de los test. Se recomienda consultar la documentación del *plugin* para ver cómo usar esta funcionalidad.

Uso de "espías"

En muchas ocasiones, más que comprobar el valor de una variable o el valor de retorno de una función nos interesará saber si una determinada función ha sido llamada correctamente (el número de veces que debería, con los parámetros adecuados, etc.). Esto es necesario cuando estamos probando un método que se llama desde otra parte de nuestro código.

En *testing* en general se suelen tratar estos casos usando *mocks*. El nombre que reciben en Jasmine es *spies*, por motivos evidentes.

Un caso de uso típico en vistas de Backbone es **comprobar que los eventos del DOM sobre la vista disparan los callbacks adecuados**. En el ejemplo del tiempo, comprobar que al pulsar sobre el botón de "ver tiempo" se llama efectivamente a la función `ver_tiempo_de`:

```

it("Al clicar sobre el botón se debería llamar a
'ver_tiempo_de", function(){
  vista = new TiempoWidget({model: new DatosTiempo()});
  spyOn(vista, 'ver_tiempo_de');
  vista.delegateEvents();
  vista.render();
  var elem = vista.$('#ver_tiempo')
  elem.click();
  expect(vista.ver_tiempo_de).toHaveBeenCalled();
});

```

- **Líneas 2 y 3:** creamos una nueva vista y el espía sobre el método `vista.ver_tiempo_de`
- **Línea 4:** al haber creado el espía hemos cambiado el manejador de evento, hay que decirle a Backbone que lo tenga en cuenta y "refresque" los manejadores

- **Línea 5:** renderizamos la vista para generar el HTML y tener algo en lo que clicar.
- **Líneas 6 y 7:** Accedemos al botón y simulamos el click
- **Línea 8:** comprobamos que se ha llamado al espía.



Es posible que veas muchos libros y tutoriales que usen la librería Sinon.js junto con Jasmine para trabajar con espías. Las versiones anteriores de Jasmine tenían algunas funcionalidades muy limitadas y de ahí la necesidad de librerías auxiliares. La versión actual de Jasmine ofrece funcionalidades en cuanto a *spies* muy similares a las que tiene Sinon.js

Otro caso similar al anterior y también muy típico es **comprobar que cuando se dispara un evento de Backbone se está llamando al callback adecuado**. En realidad es el mismo caso que antes, pero ahora con eventos de Backbone en lugar de eventos del DOM.

Por ejemplo en el *widget* del tiempo queremos comprobar que efectivamente se está llamando a `renderDatos` cuando cambia el atributo `dt` del modelo.

```
it("Al cambiar el atributo 'dt' del modelo se llama a
  'renderData'", function() {
  spyOn(TiempoWidget.prototype, 'renderData');
  vista = new TiempoWidget({model: new DatosTiempo()});
  vista.model.trigger("change:dt");
  expect(vista.renderData).toHaveBeenCalled();
});
```

Recordar que en el `initialize` de `TiempoWidget`¹⁰⁴ vinculábamos el método `renderData` al evento de cambio sobre el atributo `dt` del modelo. Si tras ejecutar el `initialize` creamos un espía sobre `renderData` el evento Backbone seguirá vinculado al `renderData` original. Es por esto que tenemos que crear el espía ANTES de vincular el evento. Nos vemos obligados a trabajar sobre el prototipo de la clase `TiempoWidget` ya que cuando se instancie la clase será demasiado tarde.

Pruebas con AJAX

Aunque es posible probar las funcionalidades AJAX de la aplicación con el servidor real, tendremos dos problemas:

- Coste temporal: la ejecución de la *suite* se hará muy lenta si incluimos muchas pruebas con AJAX.
- Fiabilidad: no sabremos si una prueba falla por nuestro código o bien porque el servidor externo ha fallado ocasionalmente. En algunos casos tampoco sabemos lo que va a devolver el servidor y por tanto no podemos asegurar que nuestro código esté procesando bien la información que recibe (caso del *widget* del tiempo).

Por ello, en la mayoría de los casos es mejor *simular* que estamos trabajando con un servidor externo. Jasmine incluye un *plugin* llamado `jasmine-ajax` que es un *mock* para el XMLHttpRequest.



De nuevo es posible que veas Sinon.js usado para esta finalidad en libros o tutoriales, ya que `jasmine-ajax` es relativamente reciente.

¹⁰⁴ https://github.com/ottocol/tiempo_backbone/blob/master/tiempo.js#L25

Para hacer que cualquier llamada a XMLHttpRequest se haga en realidad al *mock* hay que haber hecho antes la llamada `jasmine.Ajax.install()`, y para que las llamadas AJAX "vuelvan a la normalidad" se hace `jasmine.Ajax.uninstall()`. Típicamente estas llamadas se harán en un `beforeEach/afterEach` respectivamente o un `beforeAll/afterAll`.

En el *widget* del tiempo, queremos comprobar que el código que hace la petición al servicio web y el callback que procesa la respuesta del servidor funcionan correctamente.

```
it("La comunicación con el servicio web funciona correctamente", function
() {
  jasmine.Ajax.install(); ❶
  t.set("localidad", "Alicante");
  t.actualizarTiempo(); ❷
  //comprobamos que la petición es correcta
  expect(jasmine.Ajax.requests.mostRecent().url).toEqual(URL_API
+ '&q=Alicante'); ❸
  expect(jasmine.Ajax.requests.mostRecent().method).toEqual('GET');
  //devolvemos una respuesta fake
  jasmine.Ajax.requests.mostRecent().respondWith({ ❹
    status: 200,
    responseText: JSON.stringify({
      weather: [
        {description: "Prueba", icon: "test"}
      ],
      dt: 0
    })
  });
  //comprobamos que las propiedades se han instanciado OK con la info
del "servidor" ❺
  expect(t.get("dt")).toBe(0);
  expect(t.get("descripcion")).toEqual("Prueba");
  jasmine.Ajax.uninstall(); ❻
});
```

- ❶ Queremos que dentro de este código se use un *mock* de AJAX y no el real
- ❷ Llamamos al método de negocio que dispara la petición AJAX
- ❸ El API del *mock* nos permite obtener información de las peticiones hechas, en este caso de la última. Comprobamos que la URL solicitada es correcta y que se ha hecho una petición GET.
- ❹ Devolvemos una respuesta *fake*, para nuestro código será como si se la hubiera devuelto el servidor
- ❺ Comprobamos que las propiedades del modelo se han fijado a los valores correctos, que venían en la respuesta del servidor.
- ❻ Finalmente, eliminamos el API *mock* por si otra prueba quiere hacer una llamada AJAX real.