Backbone.js

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

Tabla de contenidos

1. Hola MVC en Javascript, Hola Backbone 4
    1.1. MVC y las aplicaciones Javascript 4
        MVC en la web (servidor) 4
        MVC en la web (cliente) 5
    1.2. *Frameworks* MVC en Javascript. Backbone 6
        Características de Backbone 6
        Estructura conceptual de una aplicación Backbone 6
    1.3. Un ejemplo básico de aplicación Backbone : el *widget* del tiempo 7
        El modelo 8
        La vista 10
        Eventos 11
    1.4. Ejercicios 12
        Modificación del *widget* del tiempo 12
            Modificación del modelo 12
            Creación de una nueva vista 12
        UAdivino 12
2. Modelos y colecciones 14
    2.1. Modelos. Funcionalidades básicas 14
        Atributos 14
        Métodos y propiedades de un modelo 15
        Inicializador y valores por defecto 15
        Validación de datos 16
    2.2. Persistencia con APIs REST 17
        Create (POST) 17
        READ (GET) 18
        UPDATE (PUT) 19
        DELETE (DELETE) 19
    2.3. Colecciones 19
        Navegar por las colecciones 19
        Ordenación y filtrado 20
        Manipulación básica 21
        Persistencia con APIs REST 21
    2.4. Eventos 21
        Tratar con eventos desde objetos Javascript 22
        Suscribirse/desuscribirse a eventos 22
        Eventos para gestionar operaciones asíncronas 23
        Emitir eventos de manera manual. Eventos propios 23
    2.5. Configuración de la comunicación con el API REST 24
        Configuración del identificador y/o la URL del modelo 24
        Parseo "a medida" de la respuesta del servidor 25
        Envío de información adicional en la petición 25
        APIs no REST. LocalStorage. 26
    2.6. Ejercicios 26
        Star Wars API (0,5) 26
        Comunicación con un API REST completo (0,75) 27

3. Vistas y templates 30
3.1. Vistas 30
La propiedad "el" 30
Rendering 31
Eventos 32
3.2. Vistas y modelos 33
Relación entre vista y modelo 33
Data binding 33
Data binding con eventos 33
Data binding automático 34
3.3. *Templates* (plantillas). El lenguaje de plantillas Mustache 35
Sintaxis básica 36
Plantillas en el lado del cliente 37
Rendering 37
Dónde almacenar la plantilla 38
Uso típico de plantillas en Backbone 38
Una vista que se corresponde con un único modelo 38
Una vista que se corresponde con un listado de modelos 39
3.4. Ejercicios 40
Formulario para dar coches de alta (0,6 puntos) 40
Listado de coches (0,65 puntos) 40
4. Jerarquías de vistas 41
4.1. Listados dinámicos 41
Subvistas en Backbone 42
La vista global 42
Subvistas con Marionette 44
4.2. Composición genérica de vistas 46
Composición de secciones con Marionette 46
Secciones anidadas con Marionette 47
4.3. Ejercicios 48
Vistas y subvistas con Backbone (0,75 puntos) 48
Vistas y subvistas con Marionette (0,5 puntos) 49
5. Interfaces web con ReactJS 50
5.1. ¿Por qué ReactJS? 50
5.2. ¡Hola React!. Introducción a los componentes 50
Sintaxis JSX 51
Crear una clase componente 52
Redibujado eficiente de componentes 53
Composición de componentes 54
Encapsulando una lista de componentes 55
5.3. Interactividad y estado 56
5.4. React y Backbone 57
Un componente con un modelo asociado 57
Un componente con una colección asociada 58
5.5. Ejercicios 59
5.6. Componente para un solo contacto (0,25 puntos) 59
5.7. Componente para mostrar formulario y lista de contactos (0,5 puntos) 60
5.8. Interactividad: creación y eliminación de contactos (0,5 puntos) 60
6. *Routers*. *Testing* 61
6.1. Routers básicos 61
Rutas con partes variables 61
Rutas por defecto 62
Navegación en el código 63

URLs completas ... 63
6.2. *Testing* con Jasmine ... 64
Suites y casos de prueba ... 64
Expectativas y *matchers* ... 65
Configuración de cada prueba ... 65
Ejecutar las pruebas ... 65
6.3. Pruebas en Backbone ... 66
Pruebas de lógica de negocio ... 66
Pruebas sobre HTML ... 66
Uso de "espías" ... 67
Pruebas con AJAX ... 68
6.4. Ejercicios ... 70
Routers (0,5 puntos) ... 70
Pruebas con Jasmine (0,75 puntos) ... 70
Pruebas de "lógica de negocio" ... 70
Pruebas de HTML ... 70
Pruebas de AJAX ... 71
7. Aplicaciones modulares ... 72
7.1. Revisitando el patrón módulo ... 72
Espacios de nombres ... 72
El patrón módulo simplificado ... 72
7.2. Módulos AMD ... 73
Definición y uso básico de módulos ... 73
"Azúcar" sintáctico para las dependencias ... 75
Carga de módulos con RequireJS ... 76
Configuración de RequireJS ... 76
Paths ... 77
Shims ... 77
plugin para almacenamiento de plantillas ... 77
7.3. Ejercicios ... 78
Módulos (1,25 puntos) ... 78
8. Miniproyecto de aplicación con Backbone y Marionette ... 80
8.1. Requerimientos ... 80
8.2. Implementación de los requerimientos "iniciales" ... 80
La "lógica de negocio" ... 80
Estructura de la interfaz ... 81
Vista de un solo comic: `js/views/VistaComic.js` ... 81
Vista de lista de comics ... 82
Vista Global ... 82
Vista de búsqueda (0,5 puntos) ... 83
De nuevo a la vista global (0,25) ... 84
Ver detalles de comic (0,25) ... 84
Cerrar la vista de detalles y volver al listado de comics (0,25 puntos) ... 85
8.3. Requerimientos "adicionales" (1,25 puntos en total) ... 86
9. Apéndice: Herramientas para gestionar el flujo de trabajo en el desarrollo *frontend* ... 89
9.1. Gestión de paquetes con `Bower` ... 90
9.2. Creación de plantillas con `Yeoman` ... 91

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.

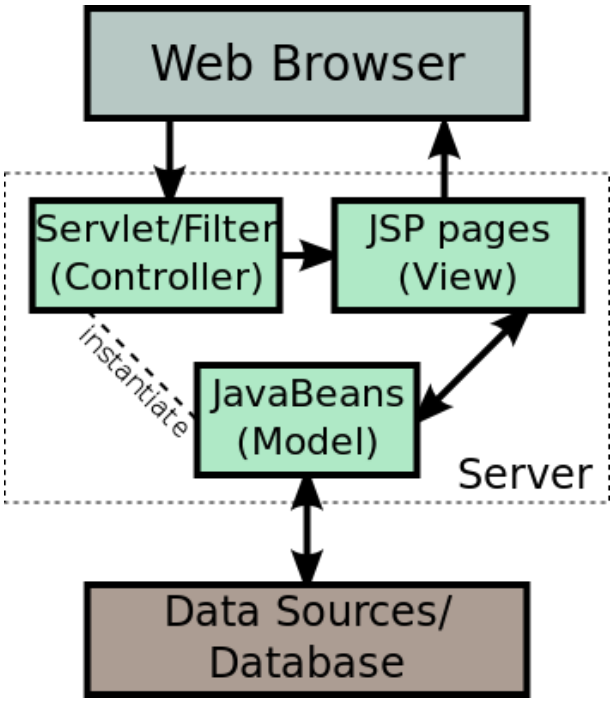


Figura 1. Modelo 2 de aplicaciones JSP. (By Libertyernie2 - Own work. CC BY-SA 3.0 via Wikimedia Commons)

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 asímismo 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 funcionalidaes 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 sustituído 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/

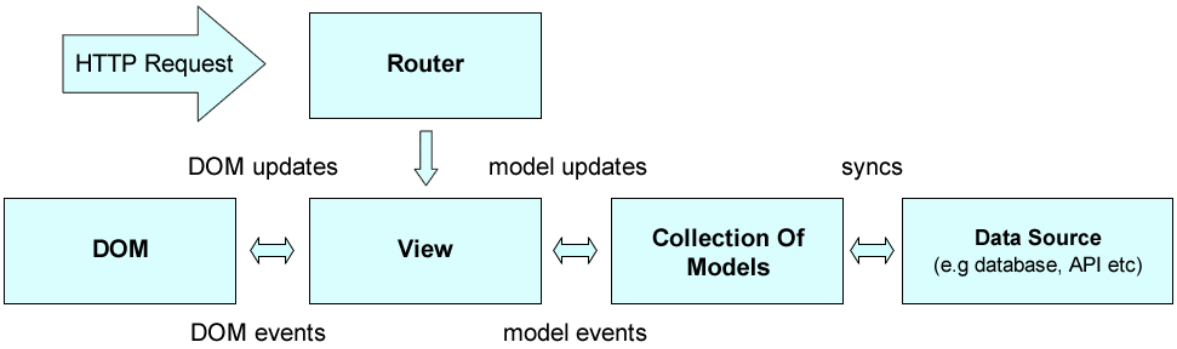


Figura 2. Típico flujo de información en una aplicación Backbone

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:

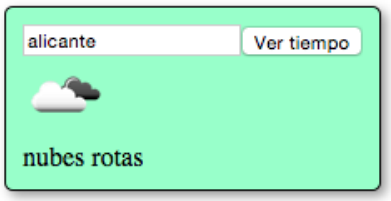


Figura 3. El aspecto del *widget* terminado

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. 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

```
"coord": {
  "lon": -0.48,
```

[12] http://openweathermap.org

```
  "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[13], 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?
units=metric&lang=es&q=";
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();
```

❶ Como vemos, la propiedad es una función, así que luego haremos `miTiempo.actualizarTiempo()` cuando queramos disparar la actualización

[13] http://openweathermap.org/weather-conditions

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();
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() {    ❹
```

```
        this.model.set("localidad", $("#localidad").val());
        this.model.actualizarTiempo();
        this.renderData();
    }
})

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

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 "atara" 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 adquiera 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

> 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

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

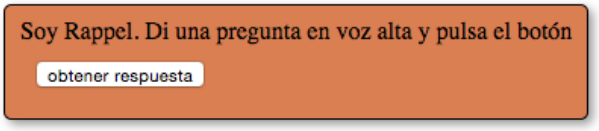
Crear un nuevo tipo de vista `TemperaturaWidget` similar a `TiempoWidget` pero que únicamente mostrará la temperatura actual. Insertarlo en el HTML y comprobar que funciona.

UAdivino

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[14] o directamente el propio código fuente anotado de Backbone, en el apartado "Helpers"[15].

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

[14] http://dailyjs.com/2012/08/09/mvstar-5/
[15] 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"[16] 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 sobreescribir 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');
  }
});
```

READ (GET)

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

[16] https://github.com/jashkenas/backbone/issues/1660

```
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 prerrequisito de Backbone.

Underscore[17] 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.

```
//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.

[17] http://underscorejs.org

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[18]. 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).

[18] http://backbonejs.org/#Events-catalog

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 incorporarle. 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.

```
//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[19] 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"));
  }
};
_.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`

[19] http://caniuse.com/#feat=object-observe

```
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 sobreescribir alguno de los métodos de Backbone para adaptarlo a nuestras necesidades.

También es muy típico el caso en el que debemos autentificarnos 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 sobreescribir 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()`[20], 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[21]. En este caso lo que haría Backbone es guardar el *wrapper* dentro del modelo, que no es lo que queremos. Tendremos pues que sobreescribir `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;
  }
  ...
});
```

Envío de información adicional en la petición

Todos los APIs en los que podamos modificar información van a requerir que nos autentifiquemos 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.

Hay varias formas de enviar la información adicional requerida. Prácticamente todas se basan en 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

[20] http://backbonejs.org/docs/backbone.html#section-75
[21] https://developer.github.com/v3/search/

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 mediante cabeceras especiales, haríamos algo como:

```
$.ajaxPrefilter(function (opts, originalOpts, jqXHR) {
  var headers = originalOpts.headers || {};
  opts.headers = $.extend(headers, {
      "X-Una-Cabecera-Arbitraria": "un_valor_arbitrario",
      "X-Otra-Cabecera-Arbitraria": "otro_valor_arbitrario"
  });
});
```

APIs no REST. LocalStorage.

Para los APIs que no sean del todo REST tendremos que sobreescribir 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*[22], 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*[23] que permiten trabajar con el API REST remoto por defecto y cambiar de manera transparente al LocalStorage cuando se detecta que estamos *offline*.

2.6. Ejercicios

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**.

Star Wars API (0,5)

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

[22] http://documentup.com/jeromegn/backbone.localStorage
[23] https://github.com/nilbus/Backbone.dualStorage

Vamos a probar cómo comunicarnos mediante Backbone con el API de Star Wars[24] 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 por defecto de Backbone a la forma de funcionar del API.

- **Define una clase modelo llamada `Personaje` y una clase colección `Personajes`** formada por instancias de la anterior. Especifica la `url` de la colección al valor que consideres apropiado.
- Si haces una petición para listar personajes[25] verás que el objeto JSON devuelto no es directamente la lista, sino que la lista está dentro de la propiedad `results`. **Reescribe el método `parse()`** de la colección `Personajes` para que rellene la colección adecuadamente. Comprueba que si creas una colección y haces `fetch` se llena de resultados. ¡¡Recuerda que `fetch` es asíncrono!!.

> 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()`[26] para que vaya haciendo `fetch()` mientras queden resultados. No es necesario que lo hagas, trabajaremos solo con 10 resultados.

- Fijate que la propiedad `id` de cada modelo de la colección está `undefined` ya que el API no devuelve un campo `id`. Pero sí identifica de manera única cada elemento con el campo `url`. Por tanto deberías **configurar la propiedad `idAttribute`** para especificar que este será el campo que actúe de identificador.

> Si pudiéramos hacer POST/PUT/DELETE sobre los recursos, el campo `url` no nos valdría como identificador, ya que Backbone generaría la URL completa del recurso concatenando la URL base de la colección, y podría salir algo como `http://swapi.co/api/people/http://swapi.co/api/people/1/`. Habría que sobreescribir el método `url` del modelo para que generara correctamente la URL, aunque no es necesario que lo hagas.

- Haz que **la colección esté ordenada** alfabéticamente por nombre de manera ascendente.
- Añade un método a la colección `buscarPorNombre(cadena)` **que la filtre** devolviendo solo aquellos personajes cuyo nombre contenga la subcadena especificada.

Comunicación con un API REST completo (0,75)

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

El API de Star Wars no nos permite hacer más que peticiones GET, así que vamos a usar otro *backend* con el que podamos desarrollar una aplicación completa. 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.

[24] http://swapi.co/
[25] http://swapi.co/api/people
[26] http://stackoverflow.com/questions/13753540/appending-data-to-same-collection-after-every-pagination-fetch

Usaremos una plataforma llamada Parse[27]. Aunque ofrece otros servicios, el que nos interesa es el de *persistencia remota*, con el que podemos hacer CRUD de objetos en el servidor que no serán más que conjuntos de pares *propiedad-valor*, al igual que en Backbone.

Para poder trabajar con la Parse lo primero es darse de alta[28] como desarrollador. Una vez dados de alta si nos autentificamos accederemos al *dashboard* donde podremos crear aplicaciones. Cada aplicación tiene un almacenamiento persistente independiente y para usarla necesitamos un par de claves, como ahora veremos.

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,…

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

- **Crea una clase modelo** `Coche`.

  # Implementa en ella un **método de validación** para comprobar al menos que la matrícula está formada por 4 dígitos seguidos de 3 letras.
  # Especifica el valor de la propiedad `urlRoot` para que se pueda sincronizar el modelo con el servidor aunque no esté dentro de una colección. Según la documentación de Parse, la URL base para un modelo es `https://api.parse.com/1/classes/nombre_de_la_clase`, donde el nombre de la clase es simplemente una etiqueta con la que aparecerán categorizados los objetos en el *dashboard* de Parse, no tiene por qué corresponderse con el nombre del modelo Backbone.
  # En Parse el identificador de un objeto es el campo `objectId` y no `id` como en Backbone, así que tendrás que añadir al modelo la propiedad

```
idAttribute: 'objectId'
```

- Modifica la función `$.ajax.Prefilter()` para que se envíen las cabeceras que necesita Parse. El id de la aplicación y la clave para usar REST las puedes obtener del apartado "API keys" del *dashboard* de Parse.

```
$.ajaxPrefilter(function (opts, originalOpts, jqXHR) {
  var headers = originalOpts.headers || {};
    opts.headers = $.extend(headers, {
      "X-Parse-Application-Id": "EL-ID-DE-MI-APLICACION",
      "X-Parse-REST-API-Key": "LA-REST-API-KEY-DE-MI-APLICACION"
  });
});
```

Para probar lo implementado, en la consola Javascript crea manualmente coches 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 clase colección* `ListaCoches`, formada por instancias del modelo anterior.

[27] http://parse.com
[28] https://www.parse.com/#signup

En la colección **implementa un método** `obtenerCoches()` que debe solicitarlos todos al servidor

> En las *queries* el API de Parse devuelve la lista de objetos resultantes en un objeto que actúa de *wrapper*, al igual que has visto que sucede con el API de Star Wars y con el API de GitHub. Tendrás que sobreescribir la función `parse()` igual que hiciste en el ejercicio anterior.

3. Vistas y templates

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 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 sobreescribir 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[29] (salvo devolver `this`, hablaremos ahora sobre esto).

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

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

[29] 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)
```

Código[30] 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:

[30] 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`[31], que vamos a ver aquí brevemente. Hay otros como por ejemplo `backbone UI`[32], que además incluye *widgets* o `backbone baguette`[33].

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
```

[31] http://nytimes.github.io/backbone.stickit/
[32] http://perka.github.io/backbone-ui/
[33] http://spacenick.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 `<b>` y `<em>` 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[34] 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[35] 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[36] o Hogan[37].

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*[38]. 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"},
```

[34] http://underscorejs.org/#template
[35] https://mustache.github.io/
[36] http://handlebarsjs.com/
[37] http://twitter.github.io/hogan.js/
[38] 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[39] de Mustache esto se hace con el método `Mustache.render`:

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

[39] 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. Ejercicios

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 (0,6 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 (bastará con un `alert`). En una aplicación real también debería escuchar el evento `error` que indica un error, para simplificar lo vamos a obviar aquí.

Listado de coches (0,65 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.
- Esta vista debe escuchar el evento `add` sobre la colección y si se produce volver añadir al HTML ya existente el del nuevo coche.

también podrías redibujar la lista entera, pero eso no sería demasiado eficiente.

4. Jerarquías de vistas

Hasta ahora todas 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[40].

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:

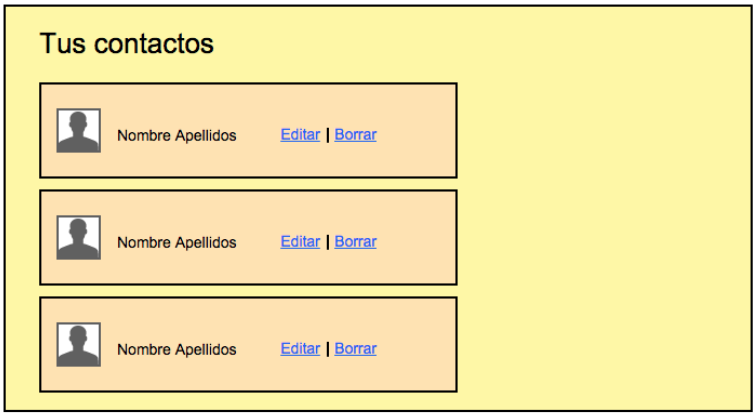


Figura 4. Vista Backbone compuesta a su vez de subvistas

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 encargar de crear un nuevo contacto si le añadiéramos el correspondiente formulario.

Hay algunas razones por la 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 veíamos 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.

[40] 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[41].

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.

```
Backbone.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
```

[41] http://marionettejs.com

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

❶ Por defecto Marionette está configurado para usar *templates* de Underscore. Podemos configurarlo para usar otro motor de plantillas sobreescribiendo el método `Backbone.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[42].

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

[42] 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 permantentes 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 qua ahora la "hija" a su vez es una `LayoutView` compuesta de subregiones.

4.3. Ejercicios

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 (0,75 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 servidor[43] (`destroy()`)
2. Si se elimina correctamente del servidor, elimine la vista[44] de la página (`remove()` de `View`).
3. Quite el modelo[45] 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.

[43] http://backbonejs.org/#Model-destroy
[44] http://backbonejs.org/#View-remove
[45] http://backbonejs.org/#Collection-remove

5. Interfaces web con ReactJS

En este tema vamos a ver una alternativa al enfoque más "clásico" de gestión de vistas de Backbone/Marionette y derivados, y en su lugar consideraremos el uso de un *framework* que ha tenido un gran impulso en los últimos tiempos: `ReactJS`.

5.1. ¿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 capa de presentación**, no es un *framework* MVC. No tiene modelos ni mucho menos controladores. De este modo, en una aplicación Backbone/React seguiríamos usando modelos y controladores Backbone convencionales, pero en lugar de usar vistas "estándar" usaríamos React.

React es un proyecto *open source* creado por Facebook y usado en producción por la misma Facebook en Instagram y por muchos otros sitios web[46] de tráfico elevado (Khan Academy, Codecademy, Atlassian,...).

5.2. ¡Hola React!. Introducción a los componentes

ReactJS está basado en **componentes**, que encapsulan el lenguaje de marcado junto con la lógica de presentación y el manejo de eventos. Como vemos, la descripción anterior se corresponde de manera bastante fiel con el papel que desempeñan las vistas de Backbone, y entre otras cosas es lo que hace factible sustituir las `View` de Backbone por `Component` de ReactJS.

Lo primero sería incluir en nuestra página la librería React, que no tiene ninguna dependencia.

```
<script src="react.js"></script>
```

[46] https://github.com/facebook/react/wiki/Sites-Using-React

Aquí definimos y dibujamos un componente React muy sencillo:

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

Con `React.createElement` creamos un componente que no es más que una etiqueta `<h1>` y le asignamos atributos HTML (un `id`) y un contenido (`¡Hola React!`). Con `React.Render` se inserta el componente React en el DOM en el punto especificado por el segundo parámetro.

Sintaxis JSX

El código anterior no es muy difícil de leer pero tiene un problema: conforme se complica el HTML a generar se hace cada vez más tedioso usar un API de este estilo, es el mismo problema que tiene el API del DOM estándar.

En ReactJS 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 el API anterior. Por eso en la práctica el código que hemos visto no es "nada típico" en React. Esta versión alternativa con JSX es mucho más común:

```
<div id="componente"></div>
<script type="text/jsx">
    var saludo = "Hola";
    React.render(<h1 id="saludo">¡{saludo} React!</h1>,
 document.getElementById('componente'));
</script>
```

Lo primero que hay que destacar es que escribir código JSX no es simplemente usar 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/jsx`. ReactJS incluye una librería adicional para *parsear* el JSX y transformarlo automáticamente a JS convencional (al estilo de la versión "antigua" de nuestro "Hola React").

hay que acordarse del `type=text/jsx` en el *tag* script, siempre que escribamos directamente código JSX y queramos que lo transforme a JS el propio navegador.

Para que el navegador compile el JSX a JS "sobre la marcha" incluimos en el HTML la librería adecuada:

```
<script src="JSXTransformer.js"></script>
```

Es mucho más eficiente hacer la transformación de JSX a JS *offline*, y que la aplicación ejecute directamente el JS generado. Para ello se puede usar

> un compilador JSX en línea de comandos[47], en cuyo caso ya no haría falta el `JSXTransformer.js`.

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 cómoda y concisa.

> Aunque estamos diciendo para simplificar que JSX nos permite mezclar HTML con JS, esto no es estrictamente cierto, lo que se usa es XML. Esto quiere decir que por ejemplo todas las etiquetas *deben* abrirse y cerrarse para que el JSX sea válido.

Crear una clase componente

En los ejemplos anteriores hemos creado un componente React pero no lo "hemos formalizado" en una clase propia. Habitualmente definiremos una clase por componente:

```
<script type="text/jsx">
var Libro = React.createClass({  ❶
    render: function() {         ❷
        return (
          <div className="libro">
              <b>{this.props.children}</b>, por <em>{this.props.autor}</
em>  ❸
          </div>
        );
    }
});

React.render(  ❹
        <Libro autor="George R.R. Martin">Tormenta de espadas</Libro>,
        document.getElementById('example')
);
</script>
```

❶ Creamos la clase que encapsula el componente.

❷ Igual que las vistas de Backbone, los componentes de React tienen un método `render` que genera su HTML. Pero a diferencia de Backbone la convención en React es que `render` devuelve dicho HTML.

❸ El componente lo podemos usar como una "nueva" etiqueta HTML. `this.props` representa los atributos y el contenido de esa etiqueta. Si hay un atributo `autor` será accesible con `this.props.autor`. `this.props.children` representa el contenido de la etiqueta.

❹ Insertamos una instancia concreta del componente en el DOM, haciendo que se dibuje. Nótese que usamos una etiqueta con el mismo nombre que la clase del componente, y pasamos información en forma de atributos o en el contenido de la etiqueta.

[47] http://facebook.github.io/react/docs/getting-started.html#offline-transform

Redibujado eficiente de componentes

Antes hemos visto que `this.props` representa los datos del componente, tal y como los podemos pasar en "formato HTML". Con la llamada del API `setProps(props)` podemos añadir o cambiar los valores. Por ejemplo podríamos hacer algo como:

```
var libro = React.render(
        <Libro autor="George R.R. Martin">Tormenta de espadas</Libro>,
        document.getElementById('example')
);
setTimeout(function() {
        libro.setProps({children:"Festín de cuervos"});
}, 1000);
```

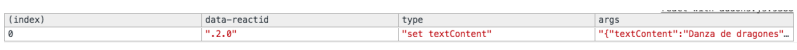
Donde el `setTimeOut` no tiene que ver con React, lo hemos usado para que se pueda ver inicialmente el título original y luego cómo cambia pasado un segundo.

Lo interesante de React es que nosotros simplemente cambiamos los datos y automáticamente se redibujará la interfaz de la manera más eficiente posible. Podemos usar las herramientas de medición de rendimiento[48] de React (`React.addons.Perf`) para comprobar qué está haciendo para actualizar la interfaz:

```
var libro = React.render(
    <Libro autor="George R.R. Martin">Tormenta de espadas</Libro>,
    document.getElementById('example')
);
setTimeout(function() {
    React.addons.Perf.start();
    libro.setProps({children:"Danza de dragones"});
    React.addons.Perf.stop();
    React.addons.Perf.printDOM();
}, 1000);
```

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.



(index)	data-reactid	type	args
0	".2.0"	"set textContent"	"{"textContent":"Danza de dragones"…

Figura 5. Operaciones sobre el DOM para actualizar la interfaz

Igual que hemos cambiado el valor de `props` también podríamos haber llamado al método `forceUpdate()` (no recomendado en la documentación) o bien simplemente haber vuelto a llamar a `React.render`. Si insertamos el componente en el mismo lugar del DOM, React detectará que ya estaba dibujándose (ya estaba

[48] http://facebook.github.io/react/docs/perf.html

> "montado", en el *argot* de React) y lo redibujará de modo eficiente. En los componentes que guardan *estado*, otra alternativa para forzar el redibujado es cambiarlo con `setState`, como veremos luego.

Composición de componentes

Podemos construir componentes de "alto nivel" que agrupen componentes ya creados, por ejemplo supongamos que nos interesa crear un componente React para representar la portada de un libro. Así, una instancia del componente `Libro` contendría una instancia del componente `Portada`:

```
<div id="libro"></div>
<script type="text/jsx">
  var Portada = React.createClass({    ❶
      render: function() {
          return (<img src={this.props.url}/>);
      }
  });

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

  var datosLibro = {
      titulo: "Juego de tronos",
      autor: "George R.R. Martin",
      portada:"http://img1.wikia.nocookie.net/__cb20130204012829/
hieloyfuego/images/9/98/Juego_de_Tronos_nueva.jpg"
  };

  var libro = React.render(
    <Libro autor={datosLibro.autor}
 portada={datosLibro.portada}>{datosLibro.titulo}</Libro>, ❸
    document.getElementById('libro')
  );
</script>
```

❶ Definimos la clase del componente "hijo" `Portada`, que como vemos no es más que una etiqueta `img` cuyo `src` apunta a una propiedad del componente que llamamos `url`. Tendremos que pasársela cuando lo usemos en el componente "padre".

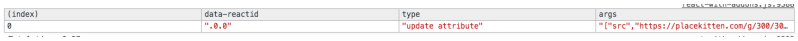
❷ Ahora en el componente `Libro` usamos una `<Portada/>`. Le pasamos la propiedad `url` con el atributo del mismo nombre.

❸ Al crear la instancia de `Libro` para dibujarla en la página, tenemos que pasarle la URL de la imagen para que el `Libro` se la pase a su vez a la `Portada`. Nótese que los componentes hijos heredan las propiedades del "padre", pero en cada nivel las propiedades pueden tener un nombre distinto.

Como los componentes "hijos" heredan las propiedades del padre, podemos cambiar simplemente las "props" del padre para forzar el re-renderizado de *toda* la estructura, aunque como antes solo se redibujarán las partes que cambien. Por ejemplo podríamos hacer desde la consola Javascript:

```
libro.setProps({portada:"https://placekitten.com/g/200/300"}
```

Y mediante las *performance tools* de React podríamos comprobar que únicamente se está modificando el atributo `src` de la etiqueta `img`:



(index)	data-reactid	type	args
0	".0.0"	"update attribute"	"["src","https://placekitten.com/g/300/30…

Figura 6. Operaciones sobre el DOM para actualizar la interfaz

Encapsulando una lista de componentes

Uno de los casos de uso más habituales en componentes que encapsulan otros es cuando tenemos uno que engloba una lista de componentes de otro tipo. Por ejemplo un componente `ListaLibros` que englobaría un conjunto de `Libro`. En este caso el método `render` del padre debe ir creando los componentes "hijos" de manera dinámica, vamos a ver cómo:

```
<script type="text/jsx">
    var ListaLibros = React.createClass({
       render: function() {
           var libros = this.props.data.map(function(libro) {  ❶
              return (
                 <Libro autor={libro.autor}>
                    {libro.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 data={datos}></ListaLibros>,  ❷
            document.getElementById('example')
```

```
    );
```

❶ El método `render` de `ListaLibros` lo que hace es ir iterando por los libros de la lista (que suponemos almacenada en una propiedad `data`) y aplicándoles una función. Para esto usamos el método `map`, que no es ni más ni menos que el "clásico" del mismo nombre de programación funcional. De este modo `libros` acaba siendo un *array* de componentes `Libro`. Al devolver el HTML de `ListaLibros` lo que devolvemos es un `<div>` que encierra esa lista de libros.

❷ Cuando *renderizamos* la instancia de `ListaLibros` le pasamos el atributo `data` con un array de datos en formato JSON.

5.3. Interactividad y estado

Como hemos visto en los ejemplos anteriores, cada componente tiene un conjunto de `props` que hereda del componente "padre" y que le sirve para saber cómo debe renderizarse. Pero los `props` no están pensados para cambiar dinámicamente. Si tenemos algún componente y queremos que contenga algún dato que pueda cambiar dinámicamente usaremos su `state`. Este no es más que un objeto al que podemos añadir las propiedades que queramos. Además del carácter estático/dinámico, respectivamente, otra diferencia entre `props` y `state` es que el `state` debería ser propio del componente y normalmente no accesible desde otros, mientras que como hemos visto, `props` se hereda del "padre".

> La cuestión de si colocar determinado dato en `props` o en `state` es "espinosa" cuando se está empezando a programar en React. Aquí[49] hay una discusión detallada sobre el tema.

Podemos modificar interactivamente el `state` de un componente usando manejadores de evento. Simplemente usaríamos los clásicos `onXXX` (`onClick`, `onSubmit`) en el HTML del componente y los vincularíamos con una función Javascript definida en el componente. Por ejemplo:

```
<div id="ejemplo"></div>

<script type="text/jsx">
    var LikeButton = React.createClass({
      getInitialState: function() {    ❶
        return {meGusta: false};
      },
      handleClick: function(event) {   ❷
        this.setState({meGusta: !this.state.meGusta});
      },
      render: function() {
        var text = this.state.meGusta ? '¡Me gusta!' : 'No me gusta';
        return (
          <button onClick={this.handleClick}> ❸
            {text}
          </button>
        );
      }
    });
```

[49] https://github.com/uberVU/react-guide/blob/master/props-vs-state.md

```
    React.render(
      <LikeButton />,
      document.getElementById('ejemplo')
    );
</script>
```

❶ Podemos asignarle un estado inicial al componente, devolviéndolo como resultado de `getInitialState()`.
❷ El manejador de evento no es más que una función definida en el componente.
❸ Aquí vinculamos el `click` sobre el botón a nuestro manejador de evento.

5.4. 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[50]. Para usar dicho código es necesario incluir un script JS[51] 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/jsx">
    var LibroComp = React.createClass({
        mixins: [Backbone.React.Component.mixin],
        render: function() {
          return (
            <div class="libro">
              <b>{this.state.titulo}</b>, por <em>{this.state.autor}</em>
            </div>
          );
        }
    });
    var libro1 = new LibroModel({titulo:"Crónicas marcianas", autor: "Ray
 Bradbury"});
    React.render(<LibroComp model={libro1}></LibroComp>,
                 document.getElementById('un_libro'));
```

[50] https://github.com/magalhas/backbone-react-component
[51] https://raw.githubusercontent.com/magalhas/backbone-react-component/master/dist/backbone-react-component-min.js

```
</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 class="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
    });
    var 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/jsx">
    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>,  ❹
        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 coleccción 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[52] 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[53]. Además del *mixin* que hemos usado aquí, hay algunas otras implementaciones alternativas[54].

5.5. Ejercicios

Vamos a desarrollar una micro-aplicación para gestionar contactos con Backbone y React. Se podrán listar contactos, añadir nuevos y eliminar los existentes. Cada contacto consistirá simplemente en un nombre y un email.

5.6. Componente para un solo contacto (0,25 puntos)

- Define un modelo Backbone llamado `Contacto` para almacenar el `nombre` y el `email`. No es necesario que tenga valores por defecto, validación ni lógica de negocio.
- Crea un componente React llamado `ContactoComp` para representar un contacto.

```
# Recuerda incluir el mixin correspondiente para comunicar Backbone y React
```

[52] https://github.com/magalhas/backbone-react-component#multiple-models-and-collections
[53] http://www.thomasboyt.com/2013/12/17/using-reactjs-as-a-backbone-view.html
[54] https://github.com/clayallsopp/react.backbone

Haz que aparezca el nombre y al lado el email en negrita, ambos dentro de un `<div>` de la clase HTML `contacto`.

Prueba que el componente funciona creando un modelo con datos cualesquiera y haz un `React.render` del componente asociado con el modelo.

Comprueba desde la consola Javascript que si cambias algún dato del modelo se re-renderiza automáticamente.

5.7. Componente para mostrar formulario y lista de contactos (0,5 puntos)

La agenda de contactos contendrá un formulario para dar de alta nuevos contactos, junto con el listado de los existentes.

Crea una clase colección Backbone `Agenda` para la parte de los datos. Luego crea un componente React llamado `AgendaComp` que encapsule la interfaz (formulario + listado) y que haga uso del `ContactoComp` para dibujar cada contacto individual. Por el momento olvídate de eliminar y crear contactos, solo hay que mostrar los ya existentes (tendrás que crear contactos por código para poder verlos en la página).

no es necesario que el botón creado sea de `type="submit"` ya que los datos no se van a enviar al servidor. Puedes usar un `<input type="button" value="Alta">` o un `<button>Alta</button>`.

5.8. Interactividad: creación y eliminación de contactos (0,5 puntos)

Para eliminar un contacto:

Añádele un botón `Eliminar` al componente `ContactoComp`. Haz que al pulsar sobre él (`onClick`, recuerda que la `C` mayúscula es importante) se elimine el modelo de la colección. Recuerda que el modelo es accesible con `this.getModel()` y la colección con `this.getCollection()`

Para crear un nuevo contacto:

- Añade atributos `ref` a los campos del formulario
- Añade un `onClick` al botón de dar de alta y vincúlalo con un manejador de evento que dé de alta el nuevo contacto. Bastará con que lo crees y lo añadas a la colección, y debería actualizarse en la interfaz automáticamente.

6. *Routers. Testing*

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.

6.1. 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: {
       '' : 'vacia',
       '*default': 'defecto'
   },
   defecto: function(path) {    ❶
        console.log('Ruta por defecto: ' + path);
   },
   vacia: 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.

6.2. *Testing* con Jasmine

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".

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 quereremos 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`).

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[55].

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[56], del repositorio[57] 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

[55] http://jasmine.github.io/2.1/introduction.html#section-Included_Matchers
[56] https://github.com/jasmine/jasmine/releases
[57] https://github.com/jasmine/jasmine

En nuestro caso, el JS incluido en el *spec runner* sería algo como:

```
...
<!-- 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>
...
```

6.3. Pruebas 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`[58] para facilitar esta tarea. Este *plugin* define un gran número de *matchers*[59] con los que podemos chequear de manera sencilla el contenido del HTML usando selectores de jQuery.

[58] https://github.com/velesin/jasmine-jquery
[59] http://jasmine.github.io/2.2/introduction.html#section-Included_Matchers

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.

```
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`[60] 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).

[60] https://github.com/ottocol/tiempo_backbone/blob/master/tiempo.js#L25

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.

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.

6.4. Ejercicios

Routers (0,5 puntos)

Bájate la aplicación de ejemplo contactos-backbone-parse[61] (puedes clonar el repositorio). Se trata de una aplicación al estilo de la de alquiler de coches que estás haciendo, pero de gestión de una agenda de contactos.

El `index.html` incluye al final el *script* `router.js` que carga un *router* muy sencillo. Fíjate que se asocia la URL `hola` con una función que imprime un saludo en la consola. Para comprobarlo, cambia la URL del navegador a `…/index.html#hola` y comprueba que sale el mensaje en la consola.

Fíjate que la agenda tiene implementado un método `filtrar(cadena)` que filtra usuarios por nombre o apellidos. Si desde la consola haces

```
lista_contactos.filtrar("Pep")
```

verás que solo se muestran en pantalla aquellos usuarios cuyo nombre o apellidos contienen `Pep`. `lista_contactos` es una instancia de la vista de Backbone que gestiona la agenda (de la clase `ListaContactosVista`). El filtrado también se puede disparar desde el botón de "filtrar" de la propia vista. De ambos casos se encarga el método `filtrar`.

Usando el *router*, haz que **cuando se acceda a** `…/index.html#filtrar/lo_que_sea` **solamente aparezcan los contactos cuyo nombre o apellidos contiene ese** `lo_que_sea`.

Una vez hecho lo anterior, modifica el código del método `filtrar` de `ListaContactosVista` (en el fichero `agenda_listado.js`) **para que cuando se le llame, se actualice también la URL** (es decir, que por ejemplo si escribimos "Pepe" en el campo de filtrado y pulsamos "filtrar" la URL debería cambiar a #filtrar/Pepe).

> Fijate que vista y router se referencian mutuamente a través de variables globales, lo que no genera un código demasiado elegante. Por simplicidad lo dejaremos así.

Pruebas con Jasmine (0,75 puntos)

Usando Jasmine escribe y ejecuta las siguientes pruebas sobre la aplicación de contactos. Escribe todas las pruebas dentro de la misma *suite*

Pruebas de "lógica de negocio"

Comprueba que un contacto sin nombre y/o apellidos no es válido (`isValid()` devuelve `false`)

Pruebas de HTML

Prueba la `VistaContacto`, que muestra un único contacto. Comprueba que tras crear una instancia asociada a un contacto y llamar a `render`, el `el` contiene el nombre del contacto

[61] https://bitbucket.org/ottocol/contactos-backbone-parse

y los botones de borrar y editar (puedes ver la plantilla de esta vista en las líneas 41-16 de index.html).

Pruebas de AJAX

Usando el *fake* AJAX de Jasmine prueba la `ListaContactosVista`. Comprueba que al crear una instancia automáticamente se hace una petición GET a la URL `https://api.parse.com/1/classes/Contacto` para bajarse los contactos del servidor. Haz que el servidor "fake" devuelve un par de contactos con datos generados manualmente y comprueba que la colección asociada a la vista contiene los datos. No es necesario que los datos "fake" contengan todos los atributos ni tampoco que compruebes que todos se guardan correctamente, basta con un par de atributos.

> Fíjate que `ListaContactosVista` espera una respuesta del servidor del estilo:

```
{
  "results": [
    {
      "objectId": "A22v5zRAgd",
      "nombre" :"Pepe",
      resto de atributos del primer contacto...
    },
    {
      "objectId": "Ed1nuqPvcm",
      "nombre": "Luisa",
      resto de atributos del segundo contacto...
    }
  ]
}
```

7. Aplicaciones modulares

En las aplicaciones Backbone que hemos implementado hasta ahora tanto las clases como las instancias de modelos, colecciones y vistas residen en el espacio de nombres global, es decir, están dentro del objeto predefinido `window`. Conforme la aplicación crece de tamaño aumentan los posibles problemas de colisiones de nombres y de visibilidad de elementos que no tendrían por qué ser visibles en todo el código.

En esta sesión vamos a ver dos soluciones para este problema, la primera bastante sencilla de implementar y que se basa en el patrón modulo que ya visteis en la asignatura de Javascript. La segunda es más formal, usaremos un estándar *de facto* para definición de módulos Javascript en el cliente llamado AMD.

7.1. Revisitando el patrón módulo

Vamos a ver aquí una forma de modularizar la aplicación bastante sencilla pero que no obstante es también bastante usada en la práctica, y que funciona razonablemente bien.

Espacios de nombres

Como ya hemos dicho, no es muy conveniente definir todos los elementos de nuestra aplicación en el espacio global de nombres, para evitar posibles colisiones. Una de las soluciones más sencillas a este problema consiste en definir objetos cuya única misión será servir de "contenedor" a los elementos de nuestra aplicación:

```
var app = app || {};
var app.modelos = app.modelos || {};
app.modelos.Libro = Backbone.Model.extend({});
var app.vistas = app.vistas || {};
...
```

La sintaxis de `objeto || {}` nos permite incluir varias veces la definición sin redefinir el objeto si es que este ya existe, así no tenemos que preocuparnos de si el objeto ya estaba definido en otra parte del código.

El patrón módulo simplificado

Recordad que el **patrón módulo** consistía simplemente en encapsular el código dentro de una función, para que sea privado, y devolver en la función la parte que queremos que sea visible en el exterior. Como ahora la parte que queremos que sea visible la definimos dentro del espacio de nombres ya no es necesario el `return`. Por ejemplo:

```
var app = app || {};
var app.modelos = app.modelos || {};

(function () {
  'use strict';

  app.modelos.Libro = Backbone.Model.extend({
    ...
  });
```

```
    ...
})();
```

7.2. Módulos AMD

AMD (Asynchronous Module Definition) es una especificación de un formato para la definición y uso de módulos en Javascript. Para cada módulo podemos especificar de qué otros módulos depende y definir las funcionalidades que ofrece y que serán visibles para otros módulos. Como ahora mismo veremos, el formato de un módulo AMD es muy similar al **patrón módulo** que ya visteis en la asignatura de Javascript.

Como su propio nombre indica, AMD es asíncrono. A diferencia de las etiquetas `<script>`, que "bloquean" el navegador hasta que no se ha descargado y analizado el Javascript correspondiente, solamente se cargará el código cuando sea necesario, es decir, cuando para algún JS especifiquemos que requiere de determinado/s módulo/s.

Definición y uso básico de módulos

Para **definir un módulo** se usa un formato como el siguiente:

```
define([dependencia1, dependencia2,....]  ❶
  function(par_dependencia1, par_dependencia2, ...) { ❷
      // código privado del módulo
      ...
      // parte visible desde el exterior, o que el módulo "exporta"
      return ...  ❸
  }
);
```

❶ Damos una lista de los módulos de los que depende el módulo actual. Normalmente como identificador de un módulo se usa simplemente el nombre del fichero que lo contiene (sin la extensión `.js`), aunque podríamos definir un identificador "a medida".

❷ Todo el módulo está definido dentro de una función, igual que hacíamos en el **patrón módulo**. Pero al contrario que en este patrón, nuestras funciones sí tendrán parámetros. Estos representan las dependencias del módulo, lo veremos más claro en un momento con un ejemplo.

❸ Al igual que en el patrón módulo, lo que queremos que sea visible al exterior lo devolvemos con `return`. El resto del código será privado.

Como decíamos, vamos a verlo más claro con un ejemplo sencillo: supongamos que tenemos una miniaplicación Backbone con un modelo (`Libro`), una colección (`Biblioteca`) y una única vista (`ListaLibros`). Vamos a dividir la aplicación en dos módulos: uno con el modelo y la colección y el otro con la vista.

Veamos cómo definiríamos el módulo con las clases `Libro` y `Biblioteca`:

```
//Archivo 'js/modelos.js'
define(['lib/backbone'], ❶
 function(BB) {  ❷
  var LibroMod = BB.Model.extend({});
  var BibliotecaCol = BB.Collection.extend({
```

```
        model: LibroMod
    });
    var miVariablePrivada = "mensaje secreto";  ❸

    return {  ❹
        Libro: LibroMod,
        Biblioteca: BibliotecaCol
    }
});
```

❶ Desde hace unas cuantas versiones, Backbone es compatible con AMD, lo que quiere decir que a su vez está definido como un módulo. Así que nuestro módulo depende a su vez del módulo Backbone, que supuestamente tenemos en el archivo `lib/backbone.js`. Como ya hemos dicho, el nombre por defecto de un módulo es el del archivo que lo contiene, sin la extensión.

❷ Los parámetros de la función que encapsula el módulo actual representan los valores exportados por los módulos dependientes. En el caso de Backbone se exporta el objeto "Backbone", que aquí hemos llamado `BB`, por lo que usamos este nombre en nuestro código.

❸ El código dentro de la función que encapsula el módulo en principio es privado de este, salvo que lo devolvamos como resultado de la función. En este caso la variable `miVariablePrivada` no va a ser accesible desde fuera.

❹ Finalmente queremos que sean visibles al exterior las dos clases que hemos definido. Creamos un objeto en el que devolvemos las dos clases asignándoles un nombre arbitrario (podría ser el mismo con que las hemos definido, en este caso usamos otro).

Como ya hemos comentado, que **Backbone sea compatible con AMD** quiere decir que está definido también como un módulo, especificando sus dependencias, que como sabemos son *underscore* y *jquery*. Si consultamos el código fuente de Backbone veremos que comienza con algo como:

```
...
define(['underscore', 'jquery', 'exports'], function(_, $, exports) {
...
```

Podemos ignorar la tercera de las dependencias, `exports`, ya que no significa que dependamos de un módulo llamado así, sino que en AMD tiene un significado especial, se refiere a interoperabilidad con otro estándar de definición de módulos llamado CommonJS.

El código anterior quiere decir que tendremos que tener `underscore.js` y `jquery.js` en nuestra aplicación para que funcione. Luego veremos físicamente en qué directorio deberían estar.

El módulo con la vista lo podríamos definir así:

```
//archivo 'js/vista.js'
define(['modelos', 'backbone'],
  function(modelos, BB) { ❶
    var VistaLibros = BB.View.extend({
      initialize: function() {
        this.collection = new modelos.Biblioteca(); ❷
      },
```

```
      //Los detalles del resto de métodos de la vista no son importantes
 para nuestros propósitos
      render: function() {
        ...
      },
      addLibro: function() {
        ...
      },
      ...
    });

    return VistaLibros; ❸
  }
);
```

❶ El módulo `modelos.js` lo representamos con el parámetro del mismo nombre y a Backbone con `BB`.
❷ Recordemos que el módulo de `modelos.js` devolvía un objeto con un par de propiedades: `Libro` y `Biblioteca`. Aquí estamos referenciando la segunda.
❸ El módulo actual devuelve la clase vista que hemos definido, aquí no hace falta "montar" un nuevo objeto para devolver resultados ya que solo queremos devolver un elemento.

Finalmente, nos quedaría escribir una especie de "programa principal" que haga uso de los módulos que hemos definido. Para ese código "de primer nivel" que hace uso del resto se suele usar el **require** de AMD, que indica que el código actual necesita para su ejecución de una serie de módulos pero no define un módulo propiamente dicho:

```
//Archivo 'js/main.js'
require(['vista'],
    function(Vista) {
        var unaVista = new Vista();
        //Aquí haríamos muchas cosas muy interesantes con la vista
        ...
    }
);
```

Como vemos es muy similar al `define` con la única diferencia de que la función que encapsula el código no devuelve nada, ya que no estamos definiendo un módulo. Solo escribimos código que queremos ejecutar.

No obstante, para que todo lo anterior podamos probarlo en la realidad, todavía nos falta una pieza clave: el *script loader* o cargador de *scripts*, que veremos en un rato.

"Azúcar" sintáctico para las dependencias

Cuando un módulo tiene una lista de dependencias muy larga, puede resultar un poco incómodo el mecanismo estándar de definir un parámetro por cada dependencia. En ese caso se puede usar una sintaxis alternativa, curiosamente denominada *sugar*. La idea consiste en que dentro del código del módulo usamos una instrucción `require` a la que le pasamos el nombre de la dependencia, y almacenamos su valor de retorno en una variable local que a partir de ese momento actuará como la dependencia en sí. Un ejemplo para verlo más claro: el módulo `vista` lo definiríamos con esta sintaxis alternativa como:

```
//archivo 'js/vista.js'
```

```
define(['modelos', 'backbone'],
  function() { ❶
    BB = require('backbone');    ❷
    mods = require('modelos');
    var VistaLibros = BB.View.extend({
      initialize: function() {
        this.collection = new mods.Biblioteca();
      },
      //El resto sería igual
      ...
    });
  }
);
```

❶ La función que encapsula el módulo ya no tiene parámetros
❷ Por cada dependencia ponemos un `require` y asignamos su valor de retorno a una variable, que usamos para referenciar la dependencia a partir de ahora, en lugar de usar el parámetro de antes.

Esta sintaxis es muy similar a la que usa otro sistema de módulos denominado CommonJS.

Carga de módulos con RequireJS

AMD únicamente es una especificación. Para llevarla a la práctica es necesario que haya un "cargador" de *scripts*, es decir, un software que se encargue de "llevar la pista" de las dependencias entre módulos, de decidir cuándo es necesario cargar un determinado módulo y de ejecutar el código de los módulos y de los `require`. El cargador más usado actualmente (y casi el único, en la práctica) es **RequireJS**[62].

Físicamente RequireJS es un único archivo Javascript que debemos incluir en nuestro proyecto al estilo "clásico". Es decir, con un `<script src="require.js"></script>`. Y de hecho *en una aplicación modular RequireJS va a ser el único código que incluyamos con la etiqueta* `<script>`, ya que el resto de los *scripts* los definiremos como módulos y por tanto será el propio RequireJS el que los cargue automáticamente.

A la etiqueta `<script>` con la que carguemos RequireJS le podemos pasar un atributo `data-main` con el nombre del archivo que contiene lo que antes llamábamos "programa principal" (un `require` de AMD).

```
<script src="js/lib/require.js" data-main="js/main.js"></script>
```

Configuración de RequireJS

Aunque no es estrictamente necesario configurar RequireJS, es útil para poder personalizar algunos aspectos. La configuración se hace dentro del "programa principal", en una instrucción `require.config`, a la que se le pasa un objeto en notación literal con las propiedades de configuración. Sin ánimo de ser exhaustivos, vamos a ver las propiedades de configuración más típicas

[62] http://requirejs.org/

Paths

Cuando definimos módulos, las rutas de los archivos se entienden definidas a partir del directorio que contiene a este "programa principal" (en el ejemplo que venimos usando, `js`). De modo que cuando antes definíamos un módulo `backbone`, RequireJS va a intentar cargar el archivo `js/backbone.js`.

Como Backbone está definido como módulo AMD y especifica dependencias de `underscore` y `jquery` eso quiere decir que RequireJS asumirá que estos últimos también están en el mismo directorio que el "programa principal". Para romper este esquema tan rígido y poder hacer que el nombre "simbólico" de las dependencias se pueda corresponder con un *path* físico distinto, podemos usar la propiedad de configuración `path`:

```
require.config({
   paths: {
       'backbone': ' lib/backbone'
       'jquery': 'lib/deps/jquery',
       'underscore': 'lib/deps/underscore'
   }
});
```

Con estos *paths* estamos indicando en primer lugar el nombre que usaremos en las dependencias y en segundo el nombre "real" del archivo a partir del directorio principal. A este nombre "real" RequireJS le sigue añadiendo automáticamente el sufijo '.js'.

Shims

En un mundo ideal todas las librerías que usáramos tendrían soporte AMD "nativo". Evidentemente esto no va a ser así siempre. En los casos en que una librería no lo incluya tendremos que definir lo que se denomina un *shim*, para integrar la librería en el sistema AMD. Las librerías que no usan módulos lo que suelen hacer eses definir una variable en el espacio de nombres global (por ejemplo jQuery define `$`, underscore define `_`, …). En la sección `shim` de la configuración de RequireJS especificaremos con `exports` el nombre de esta variable. Y si la librería a su vez depende de otras especificaremos las dependencias con la propiedad `deps`. Por ejemplo, para usar versiones antiguas de Backbone y Underscore sin soporte AMD haríamos algo como:

```
require.config({
    shim: {
        'lib/underscore': {
          exports: '_'
        },
        'lib/backbone': {
            deps: ['lib/underscore', 'jquery'],
            exports: 'Backbone'
        }
    }
});
```

plugin para almacenamiento de plantillas

Ya hemos visto que un truco muy usado para almacenar plantillas es en forma de *tags* `<script>` dentro del HTML. Pero cuando crece el número de plantillas esta deja de ser una

solución práctica, ya que estamos "ensuciando" demasiado el HTML con código adicional. En estos casos resulta más "limpio" almacenarlas en archivos aparte, probablemente en un directorio `templates` y un archivo por plantilla, y recuperarlas mediante peticiones AJAX.

RequireJS tiene un *plugin* llamado "text", especialmente diseñado para gestionar dependencias de archivos que no son de código sino de texto, y que es muy usado para gestionar las dependencias de las plantillas.

Para usar el *plugin* lo primero es bajárselo y guardarlo junto con el resto de librerías de nuestro proyecto. Lo añadiremos también a la sección `paths` de nuestro `require.config`.

Para especificar que una dependencia de un módulo es en realidad una plantilla, simplemente pondríamos como prefijo de su nombre la cadena `text!`, así no se interpretará como un módulo JS, por ejemplo:

```
define(['modelos', 'backbone', 'mustache',
        'text!templates/vistalibros.mustache'],
  function(modelos, Backbone, Mustache, tmpl) {
    var VistaLibros = Backbone.View.extend({
      initialize: function() {
        this.collection = new modelos.Biblioteca();
      },
      template: tmpl,
      render: function() {
        this.
$el.html(Mustache.render(this.template, this.collection.toJSON()));
        return this;
      }
    })

    return VistaLibros;
  }
);
```

7.3. Ejercicios

Módulos (1,25 puntos)

Convierte la aplicación de gestión de contactos en una aplicación modular con AMD y RequireJS. Haz la entrega en una carpeta `contactos-backbone-parse-amd`.

> Convierte la aplicación tal cual está en el repositorio[63], no la versión que modificaste en la sesión anterior. Así evitarás problemas con los módulos si tienes variables globales. Y también te ahorrarás convertir los tests, sería demasiado largo. Aquí tienes un tutorial[64] sobre cómo estructurar *tests* de Jasmine en una aplicación modular.

Deberías seguir una estructura de directorios como esta, que es más o menos habitual en aplicaciones modulares Backbone:

```
index.html
```

[63] https://bitbucket.org/ottocol/contactos-backbone-parse
[64] http://kilon.org/blog/2012/08/testing-backbone-requirejs-applications-with-jasmine/

```
    ...js/
        main.js
        .../models
                Contacto.js
        .../collections
                ListaContactos.js
        .../views
                ContactoVista.js
                ListaContactosVista.js
        .../templates
                ContactoVista.html
                ListaContactosVista.html
        ../lib
            (Aquí irían todas las librerías JS)
...css/
```

Ten en cuenta que además de definir los módulos AMD en sí tendrás que:

- Separar cada modelo, colección o vista en su propio archivo .js
- Además de las librerías JS que están en la carpeta `lib` del repositorio original de `contactos-backbone-parse` necesitarás requireJS[65] y su *plugin* text[66]
- Usar el plugin `text` de RequireJS para "limpiar" el código de `index.html` de plantillas (los `<script>` que no son Javascript sino Mustache) y mover estas a ficheros independientes.

[65] http://requirejs.org/docs/release/2.1.16/minified/require.js
[66] https://raw.github.com/requirejs/text/latest/text.js

8. 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.

8.1. Requerimientos

Queremos desarrollar una pequeña aplicación de gestión de comics en la que podamos buscar comics de Marvel usando su API REST[67]. Si estamos autentificados 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 podrá hacer *login* y *logout*, usando Parse como *backend*.
- Si se ha hecho login, 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 Parse 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, probablemente más que suficientes). No obstante, si deseas usar tu propia API key, puedes darte de alta como desarrollador[68] en la web de Marvel para obtenerla, accediendo luego a tu panel de control de desarrollador.

8.2. Implementación de los requerimientos "iniciales"

Aquí tienes la plantilla de proyecto[69] que puedes usar como base para tu código.

La "lógica de negocio"

La plantilla usa espacios de nombres y módulos al estilo "patrón módulo", pero no módulos AMD. Tienes ya implementado un modelo (`Comic`) y una colección (`Comics`). Esta última

[67] http://developer.marvel.com/
[68] https://developer.marvel.com/pleasesignin
[69] https://bitbucket.org/ottocol/marvel-miscomics/downloads

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 autentificados, 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="VistaComicTmpl"`, 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[70] 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 `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

[70] http://developer.marvel.com/documentation/images

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=VistaBuscarComicsTmpl`, 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).
- Como la búsqueda será 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);
  }
}
```

8.3. Requerimientos "adicionales" (1,25 puntos 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 Parse.

el código de la plantilla usa las mismas claves de Parse que la aplicación de "contactos-backbone-parse". Deberías cambiarlas por las tuyas, en las líneas 14-15 del `js/main.js`.

Para probar la aplicación con Parse lo primero es añadir algún usuario de prueba:

1. Autentificarse en parse.com e ir al *dashboard* de la aplicación
2. En la barra superior, pulsar sobre "Core" (el que aparece con un icono de un átomo)
3. En la barra de la izquierda, pulsar sobre "Add Class"
4. En el cuadro de diálogo, seleccionar en el desplegable la clase predefinida `User`. Aparecerá a la izquierda el número de usuarios dados de alta (por ahora 0). Pinchando en ese icono "User" aparecerá la tabla de usuarios y podremos dar de alta con `+ Row`. Solo es necesario escribir username y password (el password aparecerá oculto una vez tecleado)

Pistas de implementación para los requerimientos que faltan:

Para el "login/logout": (0,25 puntos)

- Tendrás que definir una vista `VistaFormLogin` o similar que muestre un formulario de login.
- Para hacer login en Parse con el API REST hay que hacer una petición GET a `https://api.parse.com/1/login`, mandando 'username' y 'password' como parámetros HTTP. En Backbone el modelo `Usuario` sería:

```
Marvel.Models = Marvel.Models || {};

(function () {
    'use strict';

    Marvel.Models.Usuario = Backbone.Model.extend({
        url: function() {
            return 'https://api.parse.com/1/login?username='+
              encodeURIComponent(this.get('username'))+'&password='+
              encodeURIComponent(this.get('password'));
        },
        idAttribute: 'objectId'
    });

})();
```

Con el modelo anterior, puedes hacer login creando una instancia, dándole valores a los atributos `username` y `password` y luego haciendo `fetch()`. 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".

- En Parse el *logout* no existe como tal, simplemente dejarías de usar la variable con el usuario, o la marcarías con algún valor especial (por ejemplo usuario.set("logged", "false")).

Para la gestión de "mis comics" (1 punto): La idea sería que usaras en Parse una clase `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 Parse 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"
- La posibilidad de eliminar un comic de la lista de favoritos

Los favoritos deberían ser exclusivos del usuario actual, pero en Parse por defecto cuando se crea un objeto es público y compartido entre todos los usuarios. Para darle permisos restringidos hay que asignarle una ACL (Access Control List). Por ejemplo:

```
var fav = new Marvel.Models.Favorito({comicId:1});
//Una ACL es un objeto con una lista de permisos. Por ahora está vacía
var acl = {};
//Suponemos que u es un Usuario
//hacemos que solo el usuario con un determinado id pueda leer y escribir
 un objeto
acl[u.id] = {read:true, write:true};
//asignamos la ACL al objeto. Debe ser una propiedad llamada "ACL"
fav.set("ACL",acl);
//Guardamos el objeto en el servidor
fav.save();
```

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

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[71] de Addy Osmani o echarle un vistazo a esta *playlist*[72] 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[73] 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[74], Bower[75] y Grunt[76] (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[77] por muchos desarrolladores, por ser más segura.

[71] https://speakerdeck.com/addyosmani/front-end-tooling-workflows
[72] https://www.youtube.com/playlist?list=PLMh6HwjXBltUZ4IixGItJ_pSG6NN3mlQr/
[73] http://nodejs.org/
[74] http://yeoman.io/
[75] http://bower.io/
[76] http://gruntjs.com/
[77] 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[78] (con repositorio[79] 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.

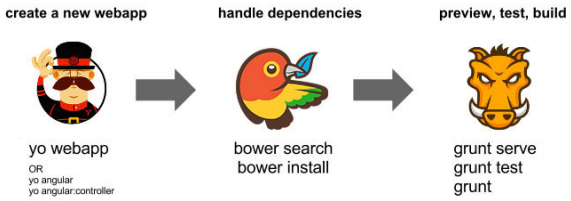


Figura 7. Flujo de trabajo con `Yeoman`, `Bower` y `Grunt` (página original)[80]

- **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
```

78 https://raw.githubusercontent.com/glenpike/npm-g_nosudo/master/npm-g-no-sudo.sh
79 https://github.com/glenpike/npm-g_nosudo
80 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[81] 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"[82]. 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]
```

[81] http://yeoman.io/generators/
[82] https://github.com/yeoman/generator-backbone