

MVC: Backbone.js

El problema: acoplamiento

Las aplicaciones JS tienden a ser un caos

- La lógica del interfaz se mezcla con
- La lógica de control y validación de datos y con
- La lógica de comunicación con el servidor y con
- La lógica de reacción a eventos!

El problema: acoplamiento

El peor tipo de código espagueti!

```
$.ajax({  
  url: 'events/since',  
  data: { timestamp: old_id },  
  dataType: 'json',  
  type: 'GET',  
  global: false,  
  cache: false,  
  success: function(newEvents) {  
    if (newEvents.events && newEvents.events.length != 0) {  
      if (prepend_events) {  
        if (page <= '1') {  
          if ($('#div.new_events').length == 0) {  
            $('#events').prepend('<div class="note new_events"><strong  
class="new_event_count">'+newEvents.events.length+'</strong> New Events Are  
Available Click here To View Them.</div>');  
          }  
        }  
      }  
    }  
  }  
});
```



MVC

En los 70 se propuso una solución:

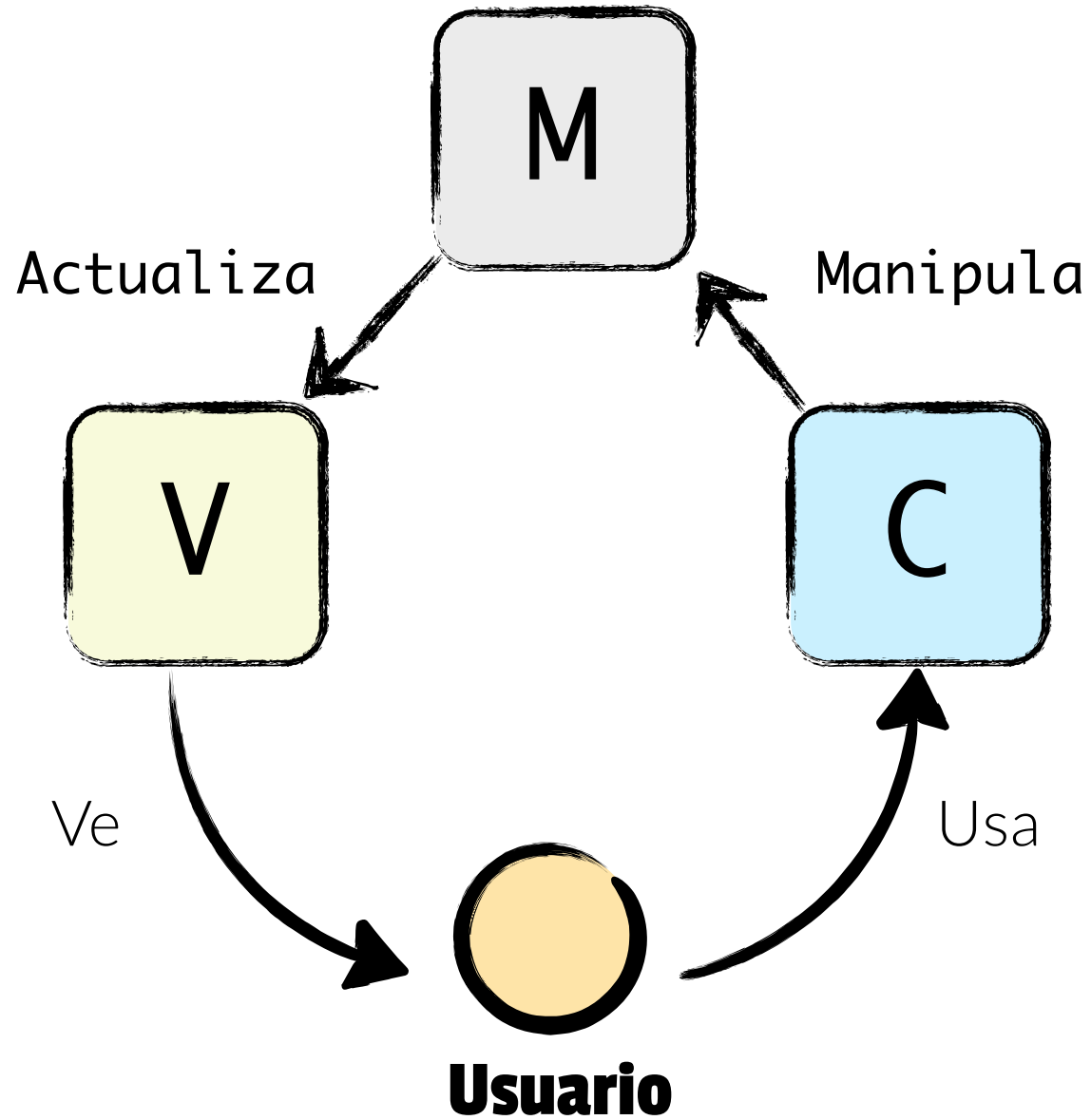
- Separar el código en 3 componentes:
 - Modelo: datos y lógica de negocio
 - Vista: presentación de los datos
 - Controlador: gestión de interacciones
- Limitar su comunicación
- Muy popular desde hace mucho tiempo!
 - Aplicaciones de escritorio
 - Aplicaciones móviles
 - Una interpretación peculiar en los frameworks web

MVC

Se ha extendido hace “poco” por JS

- Backbone.js
- Spine.js
- Batman.js
- Ember.js
- ...

MVC



MVC

Vamos a aplicar el patrón MVC

- Entender su filosofía
- Comprender los mecanismos fundamentales
- Apreciar sus ventajas
- Ver sus limitaciones
- Estudiar diferentes soluciones

MVC

JS con todas nuestras utilidades:

- [tema4/lib/prelude.js](#)
 - **Class** con herencia de propiedades de clase
 - **bind, curry, clone, merge**
 - Namespaces
 - Observable
 - Algunas utilidades extra

Modelo

Modelo

El Modelo se encarga de manejar los datos

- Representa una entidad
- Mecanismos de lectura y modificación
- Operaciones con los datos
- Validación
- Serialización

Modelo

Representar una entidad

```
var Usuario = Backbone.Model.extend({  
    /* ... */  
});
```

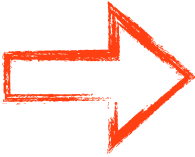
Modelo

Un modelo, en esencia, un conjunto de atributos

```
var u = new Usuario();  
  
/* Crear o modificar un atributo */  
u.set({nombre: "Pepito"});  
  
/* Leer el valor de un atributo */  
var nombre = u.get("nombre");  
console.log(nombre);
```

Modelo

Valores por defecto para los atributos: **defaults**



```
var Usuario = Backbone.Model.extend({  
  defaults: {  
    nombre: "Anónimo",  
    nacionalidad: "Español"  
  }  
});
```

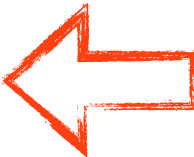
```
var u = new Usuario();  
console.log(u.get("nombre"));
```

```
u.set({nombre: "Pepito"});  
console.log(u.get("nombre"));
```

Modelo

Un modelo es un objeto, ¡Pero ten cuidado!

```
var Usuario = Backbone.Model.extend({  
  defaults: {  
    nombre: "Anónimo",  
    nacionalidad: "Español"  
  },  
  saludar: function() {  
    return "Hola, soy " + this.nombre;  
  }  
});
```



```
var u = new Usuario();  
u.set({nombre: "Pepito"});  
console.log(u.saludar()); // "Hola, soy undefined"
```

Modelo

Los atributos del modelo se consultan con **get/set**

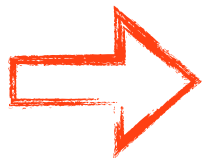
```
var Usuario = Backbone.Model.extend({  
  defaults: {  
    nombre: "Anónimo",  
    nacionalidad: "Español"  
  },  
  saludar: function() {  
    return "Hola, soy " + this.get("nombre");  
  }  
});  
  
var u = new Usuario();  
u.set({nombre: "Pepito"});  
console.log(u.saludar()); // "Hola, soy Pepito"
```

Modelo

Serializar los atributos: `.toJSON()`

```
var Usuario = Backbone.Model.extend({
  defaults: {
    nombre: "Anónimo",
    nacionalidad: "Español"
  },
  saludar: function() {
    return "Hola, soy " + this.get("nombre");
  }
});
```

```
var u = new Usuario();
u.set({nombre: "Pepito"});
```



```
var attrs = u.toJSON();
console.log(attrs);
```


Modelo

Validez: método `.validate(attrs)`


- El método no viene definido por defecto
- Nosotros lo creamos para validar nuestro modelo
- Se llama automáticamente desde **save**
- Se puede forzar al hacer `.set()`

Modelo

Validez: método `.validate(attrs)`

- Si los valores de **attrs** son válidos:
 - Devuelve una descripción del error
 - Puede ser cualquier cosa...
- Si el no son válidos:
 - No se devuelve nada

Modelo



```
var Usuario = Backbone.Model.extend({
  defaults: {
    nombre: "Anónimo",
    nacionalidad: "Español"
  },
  validate: function (attrs) {
    if (/^\s*$/.test(attrs["nombre"])) {
      return "El nombre no puede quedar en blanco!";
    }
  }
});
```

```
var u = new Usuario();
```

```
u.set({nombre: "Pepito"}, {validate: true});
console.log(u.get("nombre"));
```

```
u.set({nombre: " "}, {validate: true});
console.log(u.get("nombre")); // "Pepito"
```



Modelo

Backbone.js es una librería agnóstica en muchos sentidos

- No impone templates
- No impone estructura
- No impone almacenamiento

Modelo

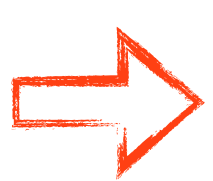
Backbone.js es una librería agnóstica en muchos sentidos

- No impone templates
- No impone estructura
- No impone almacenamiento

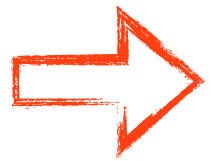
Excepto...

- Impone su propio sistema de herencia y clases
- Muy simple
- Muy limitado

Modelo



```
var Usuario = Backbone.Model.extend({  
  initialize: function () {  
    // Constructor  
  }  
});
```



```
var Ninja = Usuario.extend({  
  initialize: function () {  
    Usuario.prototype.initialize.apply(this, arguments);  
    // Subclase  
  }  
});
```

Modelo

Parecida a la nuestra, pero:

- No se pueden crear objetos que no hereden de una clase base de Backbone (**Model**, **View**, **Collection** o **Router**)
- La manera de llamar al súper método es incómoda y viola el principio DRY

Modelo

Por suerte...

- Javascript es tremendamente flexible
- El modelo de Backbone es muy simple
- klass.js la hemos escrito nosotros y entendemos cómo funciona

Modelo

Podemos integrar ambas librerías fácilmente!

- “Envolviendo” las clases base de Backbone en clases de `klass.js`
- Así tenemos lo mejor de las dos

klass.js ↔ backbone.js

```
var ProJS = (function (my) {  
  var wrapBackboneClass = function(className) {  
    var backboneWrapped = Backbone[className],  
        F = function() {},  
        K = function() {};  
    F.prototype = backboneWrapped.prototype;  
    K.prototype = new F();  
    _.extend(K, ProJS.Class, {constructor: backboneWrapped});  
    _.extend(K.prototype, ProJS.Class.prototype);  
    K.prototype.init = function() {  
      return backboneWrapped.apply(this, arguments);  
    };  
    return K;  
  };  
  
  my.Model = wrapBackboneClass('Model');  
  my.View = wrapBackboneClass('View');  
  my.Collection = wrapBackboneClass('Collection');  
  
  return my;  
})(ProJS || {}));
```

klass.js ↔ backbone.js

Ahora podemos hacer:

```
var Usuario = ProJS.Model.extend({  
  init: function() {  
    // constructor  
    this._super();  
  },  
  defaults: {  
    // etc...  
  }  
});
```

Modelo

Ejercicio (uno fácil para empezar):

<tema4/model-1/index.html>

- Escribe un modelo Producto
 - Id (incremento automático)
 - Nombre (obligatorio)
 - Categoría (obligatorio)
 - País (España, Portugal o Francia)
 - Precio (obligatorio, sin IVA)
- Escribe un decorador para obtener precios con IVA

Modelo

Fíjate en...

- Lo fácil que resulta sobrescribir la funcionalidad de un método con `klass.js`
- Si no tuviéramos `klass.js`, ¿Cómo habríamos hecho el decorador? ¿De quién tendría que haber heredado?
- Puedes pasar varios atributos a la vez a `.set()`
- Puedes pasar los atributos directamente al constructor
- Es una base bastante sólida para controlar los datos de nuestra aplicación!

Modelo

Los modelos generan eventos

- Cuando alguno de sus atributos cambia: **change**
- Cuando un atributo en concreto cambia: **change:[attr]**
- Cuando las validaciones fallan: **invalid**
- Al ser destruidos: **destroy**
- Al ser sincronizado con el servidor: **sync**
- Si surge algún error al guardar: **error**

Modelo

```
var p1 = new Producto();

p1.on("change", function (model, options) {
  console.log("El producto", model.get("nombre"), "ha cambiado!");
});

p1.set({
  nombre: "Jamón",
  categoria: "Comida",
  pais: "España",
  precio: 65
});
```

Modelo

Ejercicio:

- Modifica el ejercicio anterior de modo que:
- Cuando una validación falla, se informe al usuario del error por la consola

Modelo

Otras operaciones útiles:

- `.unset()`: elimina un atributo
- `.clear()`: elimina todos los atributos
- `.previous(attr)`: durante un evento **change**, devuelve el valor anterior de un atributo
- `.has(attr)`: ¿Tiene el modelo el atributo **attr**?
- `.escape(attr)`: como `.get()`, pero escapando el HTML

Modelo

Persistencia

- La “gracia” de Backbone es que sabe como hablar con el servidor
- Pedir y guardar modelos automáticamente por AJAX
- Muy flexible
- Soporta también localStorage

Modelo

Para hablar con el servidor, el modelo necesita:

- **urlRoot**: la base con la que construir su URL
- **id**: el identificador del recurso
- **parse** (opcional): una función que interprete la respuesta del servidor

Modelo

Las operaciones fundamentales de persistencia:

.fetch()

1. dado un id, construye una url del tipo [baseURL]/[id]
2. GET al servidor
3. pasa la respuesta a **.parse()**
4. con el resultado llama a **.set()** para establecer los atributos del modelo

Modelo

Ejercicio

[tema4/model-2/index.html](#)

Con una api AJAX tal que:

GET /products -> lista de {nombre: "", id: #}

GET /products/:id -> detalles del producto :id

PUT /products/:id -> guarda cambios del producto :id

Haz:

- Construye un array con un modelo por cada producto del listado
- Modifica algún producto y guarda los cambios
- Escucha los eventos “change” y “sync” e informa por consola

Colecciones


Colección: conjunto ordenado de modelos

- Se identifica con un listado de recursos
 - GET a URLs de tipo “índice” (/users, /products, etc...)
- Los modelos de una colección son del mismo tipo

Colecciones

- Objetivo similar al del un modelo, pero con conjuntos
 - Manejar colecciones de datos
 - Ordenar, añadir, eliminar entidades a la colección
 - Consultar y guardar la colección en el servidor
 - Serializar el conjunto de datos

Colecciones



```
var ListadoProductos = ProJS.Collection.extend({  
    model: Producto  
});
```

```
var listado = new ListadoProductos();
```


Colecciones

Operaciones fundamentales:

- `add(model, {at: i})`: añadir un modelo a la colección
- `remove(model|id|cid)`: eliminar un modelo
- `get(id|cid)`: acceder a un objeto de la colección por id
- `at(idx)`: acceder a un objeto de la colección por índice
- `length`: número de elementos en la colección
- `where(attrs)`: query de atributos
- `push/pop, shift/unshift`

Colecciones

```
var listado = new ListadoProductos();  
  
listado.add(new Producto({nombre: "Uno"}));  
listado.add(new Producto({nombre: "Dos"}));  
listado.add(new Producto({nombre: "Tres"}));  
  
console.log(listado.at(2).toJSON());
```

Colecciones

Para identificar a los modelos dentro de una colección, podemos usar:

- **id**

- Generalmente otorgado por el servidor
- Se utiliza para construir la URL del modelo
- Universal dentro de la app
- Se corresponde, habitualmente, con el ID de la tabla en BBDD

- **cid**

- Generado automáticamente por Backbone
- Válido solo dentro de la página
- Modelos no guardados o que no tienen que ver con BBDD

Colecciones

```
console.log(new Producto().cid); // c1  
console.log(new Producto().cid); // c2  
console.log(new Producto().cid); // c3
```

Colecciones

```
var p = new Producto({nombre: "Zapatos"}),  
    cid = p.cid;
```

```
console.log(listado.get(cid));    // undefined
```

```
listado.add(p);  
console.log(listado.get(cid));    // p
```

```
var resultado = listado.where({nombre: "Zapatos"});  
console.log(resultado.toJSON()); // p
```

Colecciones

Una colección sabe como interpretar datos
“crudos” (instancia automáticamente un modelo)

```
listado.add({  
  nombre: "Corbata",  
  categoría: "Caballero",  
  precio: 40  
});
```

```
listado.at(0).constructor === Producto; // true
```

Colecciones

Cargar datos iniciales en una colección: **reset**

```
var listado = new ListadoProductos();  
  
listado.reset([  
    {nombre: "Uno", categoria: "A", precio: 2},  
    {nombre: "Dos", categoria: "B", precio: 1},  
    {nombre: "Tres", categoria: "C", precio: 8},  
]);
```

Colecciones

Las 28 funciones de underscore para manipular listas se pueden aplicar a colecciones

map

reduce

find

filter

max/min

sort

shuffle

etc...

Colecciones

Persistencia:

url: dirección para pedir la colección

- Los modelos de la colección usarán esta URL como **urlRoot** para construir sus URLs individuales

fetch: pide la colección al servidor

- GET a url, esperando un array de hashes (atributos)

update: refresca los datos de la colección

- Aproximadamente igual que hacer un fetch y mergear

Colecciones

Ejercicio:

[tema4/collection-1/index.html](#)

Crea una colección de Productos que lea de `/products`

Y tenga los métodos:

`listado()`: todos los productos

`ordenaPorNombre()`: filtro

`precioMenorQue(p)`: filtro

`borrarProducto(id)`: lo elimina de la BBDD

`nuevoProducto(attrs)`: añade el producto a la colección y lo guarda en BBDD

Colecciones

Las colecciones también emiten eventos:

`add(model, col)`: se ha añadido un nuevo modelo

`remove(model, col)`: se ha eliminado un modelo

`sort(col)`: se ha reordenado

Vista

Es una representación del modelo

- Asociada a una instancia de un modelo
- Generalmente utiliza un template
- Actualización automática

Vista

Templates

- “Plantillas” que mezclan HTML y código JS
- Backbone funciona con cualquier librería de templates
- Trae una preinstalada: `_.template()`

Vista

`_.template(texto, datos)`

- **texto**: el texto de nuestra plantilla
- **datos**: un objeto con las variables que queremos utilizar al evaluar el template

Vista

En el texto de la plantilla:

`<%= expresión %>`

Se sustituye por el resultado de evaluar la expresión

`<%- código %>`

Ejecuta el código javascript

Vista

```
var plantilla = "<h1> Hola! </h1>";  
console.log(  
  _.template(plantilla, {})  
);
```


Vista

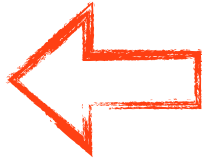
```
var plantilla =  
" <h1> \  
    <%- console.log('Hola!') %> \  
    </h1>";  
  
console.log(  
    _.template(plantilla, {})  
);
```

Vista

```
var plantilla =  
" <h1> \  
    <%= 10 + 10 %> \  
    </h1>";  
  
console.log(  
    _.template(plantilla, {})  
);
```

Vista

```
var plantilla =  
" <h1> \  
    Bienvenido, <%= nombre %> \  
    </h1>";  
  
console.log(  
    _.template(plantilla, {nombre: "Ulises"})  
);
```

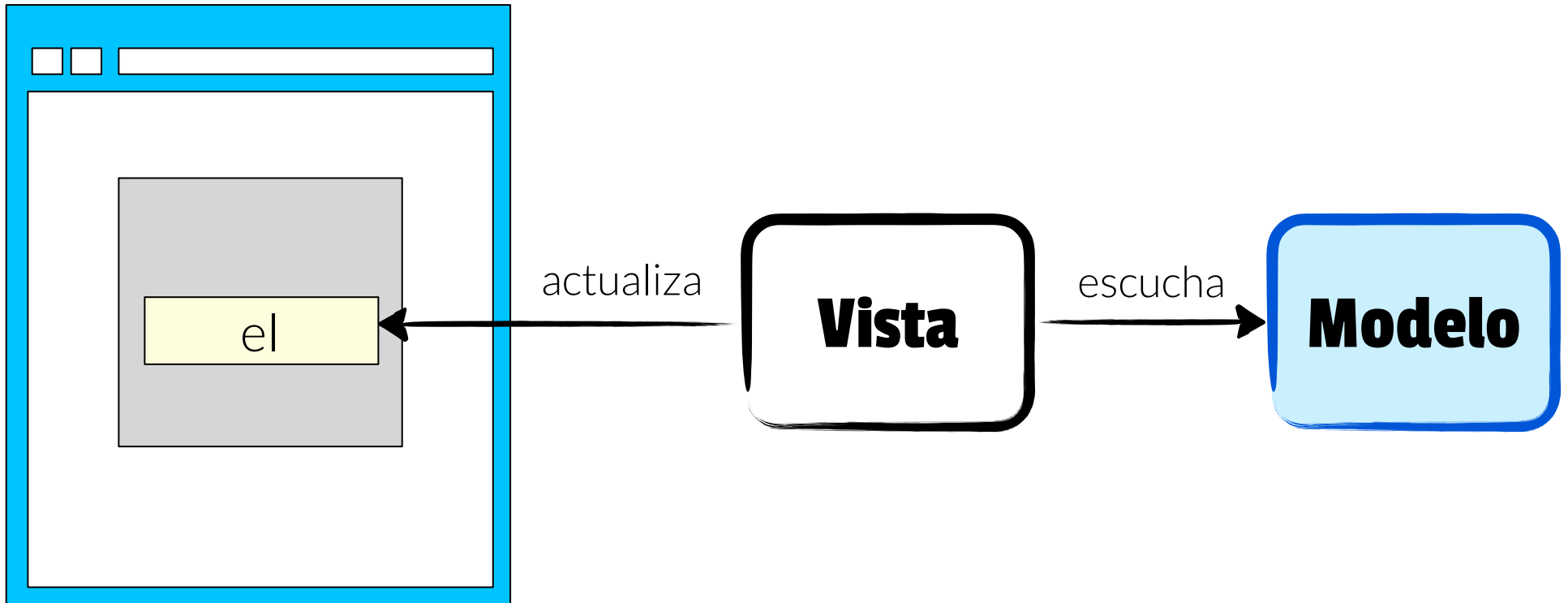


Vista

Anatomía de una vista

- Un modelo del que se extraen los datos
- Un template que se rendera con los datos del modelo
- Un nodo del DOM donde se inserta el template renderado
- Un método **.render()** que se encarga de ejecutar este proceso

Vista



Vista

En Backbone:

- `.model`: el modelo asociado a la vista
- `.tagName`: la etiqueta para generar el nodo del DOM
- `.attributes`: los atributos para generar el nodo
- `.el`: el nodo, ya creado
- `$.el`: el nodo, envuelto con jQuery (o Zepto)
- `.$`: un atajo a jQuery pero con el scope fijado en `.el`
- `.render`: el método que se encarga de renderear el template y actualizar el contenido de `.el`

Vista

```
var MiVista = ProJS.View.extend({
  init: function(options) {
    this._super(options);
  },
  tagName: "div",
  attributes: {class: "box large"},
  template: "<h1> <%= nombre %> </h1>",
  render: function() {
    var data = this.model.toJSON();
    this.$el.html( _.template(this.template, data) );
    return this;
  }
});

var producto = new Producto({nombre: "Vino"}),
    miVista = new MiVista({model: producto}).render();

$("body").append(miVista.el);
```

Vista

Ejercicio: una vista sencilla

[tema4/view-1/index.html](#)

Crea una vista que utilice el template **#producto-template** para mostrar una instancia de **Producto**

Vista

Ejercicio: vistas y colecciones

[tema4/view-2/index.html](#)

Crea una colección que se inicialice con los datos de la ruta /products

Para cada uno de los modelos:

- Instancia una vista VistaListado

- Muéstrala por pantalla


Una pista: ¡las colecciones emiten eventos!

Vista

La vista debería “escuchar” al modelo

Cambios automáticos cuando el modelo cambia

El patrón habitual:



```
var VistaListado = ProJS.View.extend({  
  init: function (options) {  
    this._super(options);  
    this.model.on("change", bind(this, this.render));  
  },  
  /* ... */  
});
```

Y ahora, ¿qué?

Tenemos Modelo y Vista

- Todo el mundo está de acuerdo en estos dos puntos
- Datos + presentación
- Todavía falta algo...

MV*

MV*

- El papel del Controlador no está tan claro
 - Gestionar la interacción?
 - Gestionar los eventos de la vista?
 - Gestionar las rutas de la página?
 - Gestionar al modelo?
 - ...

MV*

La visión tradicional:



MV*

En JavaScript...



MV*

En JavaScript...



Proxy ← ¡Todo lo demás! → Template

Controlador

El modelo “estándar” de Backbone.js

- Nadie lo dice abiertamente, pero:
- No hay Backbone.Controller
- La vista propiamente dicha es el template
- Pero Backbone.View gestiona la interacción...
- Es decir, hace de controlador

Controlador

Si no te he convencido...

¿Quién reacciona al input del usuario?

Backbone.View

¿Quién se encarga de actualizar los datos del modelo según ese input?

Backbone.View

¿Dónde se programa la lógica del interfaz de usuario?

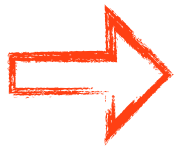
Backbone.View

Controlador

Escuchar eventos en la vista:



```
var VistaProducto = ProJS.View.extend({  
  events: {  
    "click a": "marcarComoActivo"  
  },  
  init: function(options) {  
    this._super(options);  
    this.template = $("#template-producto").html();  
    this.model.on("change", bind(this, this.render));  
  },
```



```
  // Event handlers  
  marcarComoActivo: function() {  
    this.model.set({active: true});  
  }  
  
  /* ... */  
});
```

Controlador

Ejercicio: cogiendo el feeling del controlador

[tema4/controller-1/index.html](#)

Haz que las vistas VistaListado:

- Escuchen los cambios del modelo y se auto-rendeen
- Escuchen el evento click en el `<a>` dentro de `this.el` y lo asocien a `marcarComoActivo`
- `marcarComoActivo` pone a `true` la propiedad “activo” del modelo
- Si la propiedad “activo” del modelo es `true`, añade la clase CSS “active” a `this.el` (que debería ser un ``)

Controlador

El ejemplo anterior tiene problemas:

- ¿Cómo podemos decirle a los demás elementos que se desactiven?
- ¿Cómo podemos avisar al resto de la aplicación que se ha seleccionado un nuevo Producto?

MV*

¡Un Mediador!

- La mejor solución para coordinar componentes de una página
- Muy bajo acoplamiento:
 - Modificar los componentes sin problemas
 - Añadir o eliminar funcionalidad en la página
- MV* + Mediador = un patrón muy común y muy flexible

MV*

Un ejemplo:

[tema4/controller-2/index.html](#)

Fíjate como VistaListado simplemente notifica al mediador

Y el mediador es quien se encarga de orquestar los demás elementos

Las ventajas son:

- Tenemos todo el flujo “a vista de pájaro” de la página en un solo sitio: el mediador

- Ningún componente está acoplado a ningún otro, solo notifica a su mediador

MV*

Ejercicio: un poco de todo!

Modifica el ejemplo anterior para que al hacer click en un elemento de la barra lateral se muestre ese modelo con una vista VistaProducto (la tabla de los primeros ejercicios)