Aplicaciones JavaScript I

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:

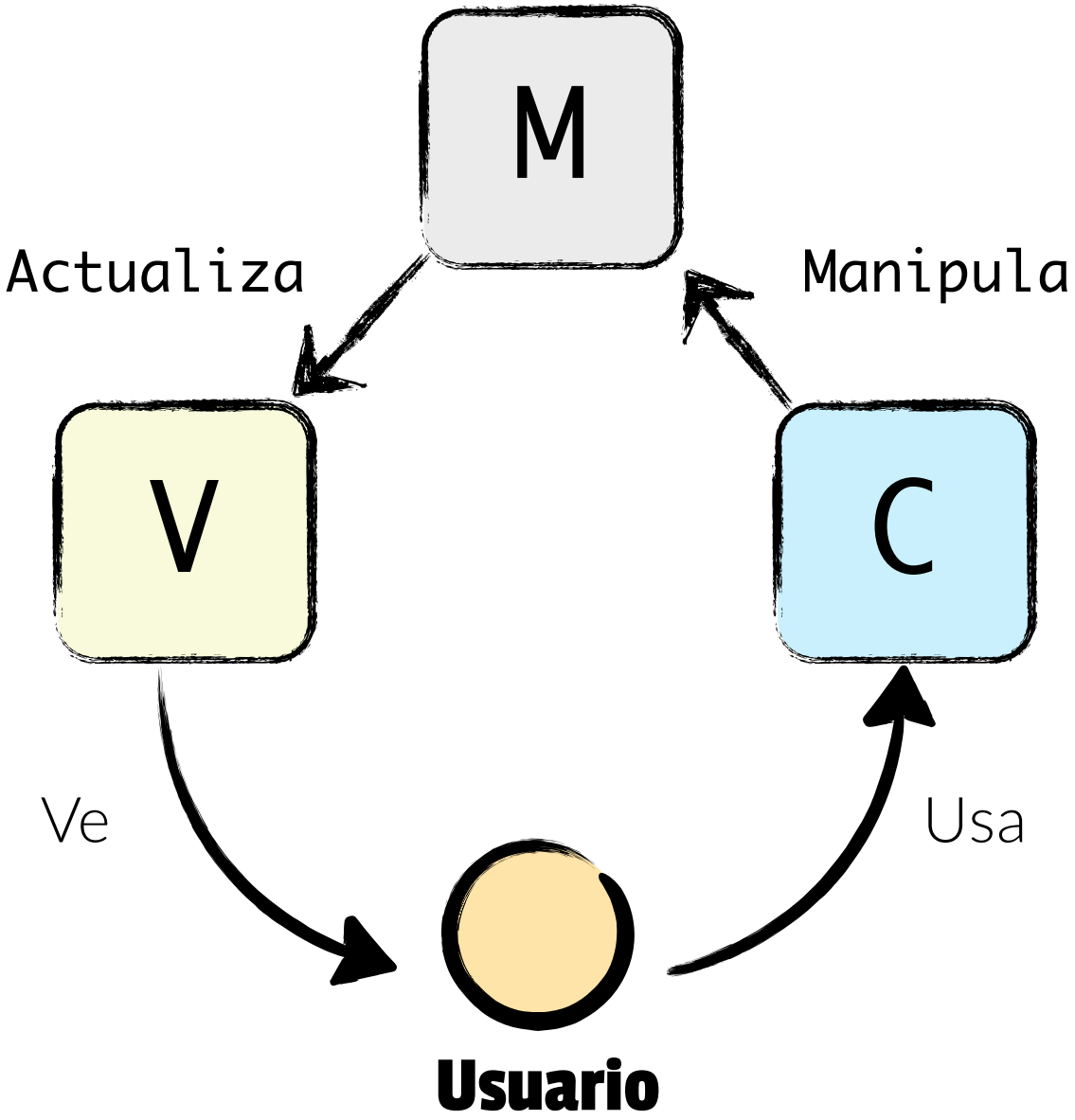
- Separar el código en 3 componentes:
  - Modelo: datos y lógica de negocio
  - Vista: presentación de los datos
  - Controlador: gestión de interacciones
- Limitar su comunicación
- Muy popular desde hace mucho tiempo!
  - Aplicaciones de escritorio
  - Aplicaciones móviles
  - Una interpretación peculiar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

MVC

Se ha extendido hace “poco” por JS

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

MVC



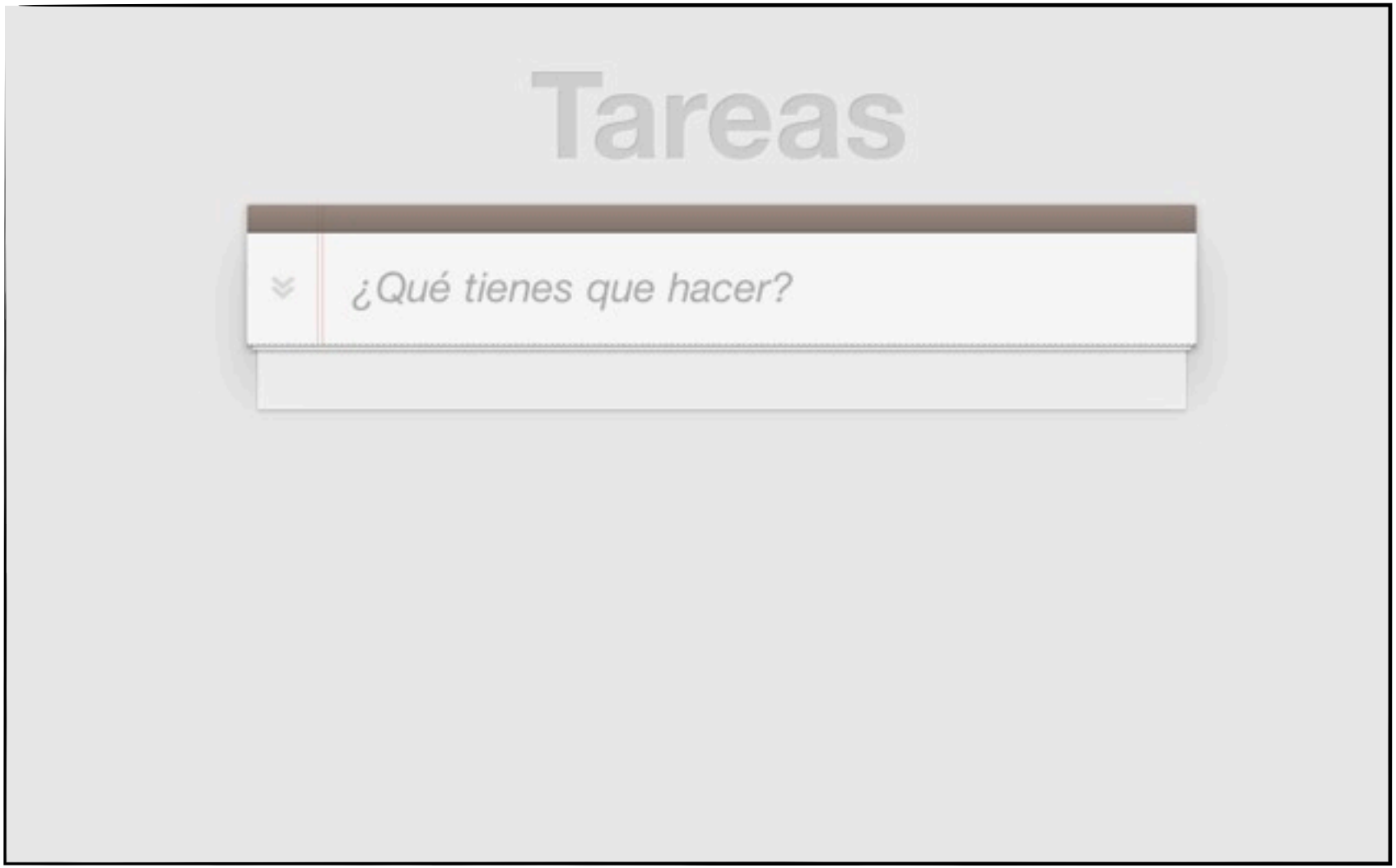
MVC

Vamos a implementar el patrón MVC

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

MVC

http://jsbin.com/adojuc/2/edit



Modelo

JS con todas las utilidades necesarias:

- https://bitbucket.org/werelax/projs-material/raw/257ed76a7af8/tema4/prelude.js
  - `Class` con herencia de propiedades de clase
  - `bind, curry, clone, merge`
  - Namespaces
  - Observable
  - Los DSLs que vimos en el tema 3
    - Plantillas
    - Prototipos aumentados (`times`, `format`, `to_f`)

Modelo

El Modelo se encarga de manejar los datos

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

Modelo

Lo queremos usar así:

```
var Tarea = Model.extend({
  defaults: {
    completed: false,
    title: "Tarea sin título",
    date: function() { return new Date(); }
  },
  completar: function() {
    this.set({completed: true});
  }
});

var miTarea = new Tarea({title: "Esta es una tarea"});
miTarea.get('title');
miTarea.set({title: "Nuevo título!"});
miTarea.toJSON(); // {completed: ..., title: ..., date: ...}
```

Modelo

Primera aproximación

- http://jsbin.com/adojuc/3/edit

Modelo

Persistencia

- localStorage
- Alguna forma de identificar cada modelo: CID
- Guardar
- Recuperar
- Borrar

localStorage

Persistencia simple con HTML5

- `localStorage[key]`
- `localStorage[key] = value`
- `localStorage.removeItem(key)`

Modelo

Completa el modelo (¡evita el acoplamiento!)

```
var miTarea = new Tarea({title: "Esta es una tarea"}),
    miTareaCid = miTarea.cid;

miTarea.save();

Tarea.find(miTareaCid, {success: function(tarea) {
  console.log(tarea.toJSON());
  tarea.remove();
}});

Tarea.find(miTareaCid, {error: function(cid) {
  console.log("No existe modelo con CID %1".format(cid));
}});
```

Modelo

Una posible implementación:

- http://jsbin.com/adojuc/8/edit

Modelo

Validaciones:

- Responsabilidad del Modelo
- Dos métodos: `.validate()` y `.isValid()`
- `.validate()` se invoca en `.set()` y `.save()`
  - No devolver nada o devolver `undefined` = OK!
  - Cualquier otro valor = Error!
- `.isValid()`
  - `true` si el modelo es válido
  - `false` si no lo es

Modelo

```
var miTarea = new Tarea({title: "Una tarea"});

miTarea.set({date: "no válida"}); // false
miTarea.get('date'); // el valor anterior!

miTarea.set({date: "no válida"}, {skipValidation: true});
miTarea.get('date'); // no válida

miTarea.save(); // false

console.log(miTarea.toJSON()); // {..., date: "no valida"}
```

Modelo

Con validaciones:

- http://jsbin.com/adojuc/9/edit

Modelo

Colecciones:

- Lista de modelos
- Operaciones:
  - Añadir
  - Eliminar
  - Serializar
  - Encontrar
- Útil para inicialización

Modelo

```
var ListaDeTareas = Collection.extend({
  model: Tarea
});

var listaDeTareas = new ListaDeTareas();

"10".times(function(i) {
  listaDeTareas.add({title: "Tarea: %1".format(i)});
});

console.log(listaDeTareas.get("model-8"));
listaDeTareas.remove("model-8");
var otraListaDeTareas = new ListaDeTareas(ListaDeTareas.toJSON());
```

Modelo

Posible solución:

- http://jsbin.com/adojuc/11/edit

Modelo

Los Modelos pueden tener URL

- Guardar y recuperar datos
- Ruta del modelo: `this.url/this.cid`
- Ruta de la colección: `this.url`

Modelos

Con URLs:

- http://jsbin.com/adojuc/24/edit

Vista

La Vista es una representación del modelo

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

Vista

Vamos a empezar por algo así:

```
var VistaTarea = View.extend({
  template: ProJS.Plantillas.byId('item-template')
});

var tarea = new Tarea({title: "Tarea 1"});
var vista = new VistaTarea({model: tarea});

console.log(vista.render());
```

Vista

El código:

- http://jsbin.com/adojuc/12/edit

Vista

Pero... ¿Qué hacemos con esto?

```
<div class="view">
  <input class="toggle" type="checkbox"/>
  <label>Tarea 1</label>
  <button class="destroy"></button>
</div>
<input class="edit" value="Tarea 1">
```

Vista

Vamos a generar un solo nodo

```
< ??? >
  <div class="view">
    <input class="toggle" type="checkbox"/>
    <label>Tarea 1</label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="Tarea 1">
</ ??? >
```

Vista

¿Qué tal `this.tagName` (o `<div>`)?

```
< this.tagName >
  <div class="view">
    <input class="toggle" type="checkbox"/>
    <label>Tarea 1</label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="Tarea 1">
</ this.tagName >
```

Vista

En nuestro caso, queremos **<li>**

```
< li >
  <div class="view">
    <input class="toggle" type="checkbox"/>
    <label>Tarea 1</label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="Tarea 1">
</ li >
```

Vista

Pero:

- Primero generamos el nodo
- Y luego rellenamos el contenido

```
render: function() {
  var data = this.model.toJSON();
  if (!this.el) this.el = $(this.tagName);
  this.el.html(this.template(data));
  return this.el;
}
```

Vista

Como en:

- http://jsbin.com/adojuc/13/edit

Vista

¿Y ahora qué hacemos con `this.el`?

- Si se ha especificado un padre (`this.parent`)
- Al generar `this.el` por primera vez
- Lo añadimos al padre (`this.parent`)

Vista

Lo queremos usar así:

```
var VistaTarea = View.extend({
  template: ProJS.Plantillas.byId('item-template'),
  tagName: "li",
  parent: $("#todo-list")
});
```

Vista

Probando que funciona hasta ahora:

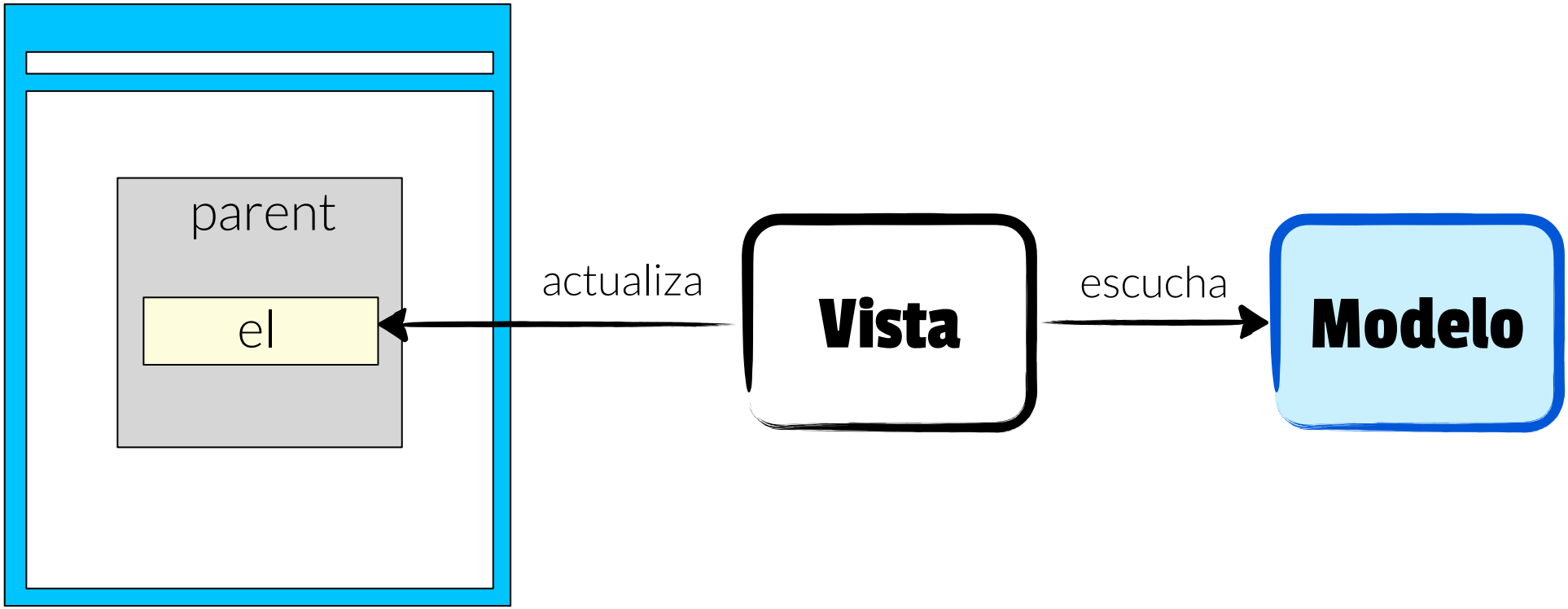
- http://jsbin.com/adojuc/14/edit

Vista

Anatomía de una vista:

- `model`: instancia de modelo asociada
- `template`: función para generar HTML
- `tagName`: etiqueta del elemento a crear (si fuera necesario)
- `el`: el elemento (nodo DOM) asociado
- `parent`: el padre de `el` (donde se añadirá el elemento)
- `render()`: rendea el template con los datos del modelo y lo mete en `el`

Vista



Vista

Escuchar al modelo:

- Patrón Observador
- ¡El modelo ha de emitir eventos!
  - `change`
  - `deleted`
- La vista se suscribe a los eventos del modelo

Vista

Hacer que el modelo emita **change** al hacer `.set()`

```
var View = Class.extend({
  init: function(options) {
    augment(this, pick(options,
                       'model', 'template', 'tagName',
                       'parent', 'el'));
    if (this.model) {
      this.model.subscribe('change', bind(this, this.render));
      this.model.subscribe('deleted', bind(this, this.remove));
    }
  },
```

Vista

```
var listaDeTareas = new ListaDeTareas();

"5".times(function(i) {
  var tarea = new Tarea({title: "Tarea: %1".format(i)}),
      view = new VistaTarea({model: tarea}).render();
  listaDeTareas.add(tarea);
});

setTimeout(function() {
  listaDeTareas.each(function(tarea) {
    tarea.set({title: "Cambio!"});
  });
}, 10000);
```

Vista

Mi versión:

- http://jsbin.com/adojuc/17/edit

Vista

Las colecciones también emiten eventos!

- `added` (con el nuevo modelo como parámetro)
- `deleted`

Chequeo

Muestra 5 tareas

```
var listaDeTareas = new ListaDeTareas();

listaDeTareas.subscribe('added', function(tarea) {
  var vista = new VistaTarea({model: tarea}).render();
});

"5".times(function(i) {
  var tarea = new Tarea({title: "Tarea: %1".format(i)});
  listaDeTareas.add(tarea);
});
```

Chequeo

En las 5 tareas pone “Cambiado...”

```
var listaDeTareas = new ListaDeTareas();

listaDeTareas.subscribe('added', function(tarea) {
  var vista = new VistaTarea({model: tarea}).render();
});

"5".times(function(i) {
  var tarea = new Tarea({title: "Tarea: %1".format(i)});
  listaDeTareas.add(tarea);
});

listaDeTareas.each(function(tarea) {
  tarea.set({title: "Cambiado..."});
});
```

Chequeo

Si te has perdido, aquí tienes un checkpoint

- http://jsbin.com/adojuc/26/edit