

# **Tema 3:**

# **Frameworks JS en el**

# **cliente**

## **Tema 3: Frameworks web en el cliente**

### **3.1**

**¿Por qué frameworks  
en el cliente?**

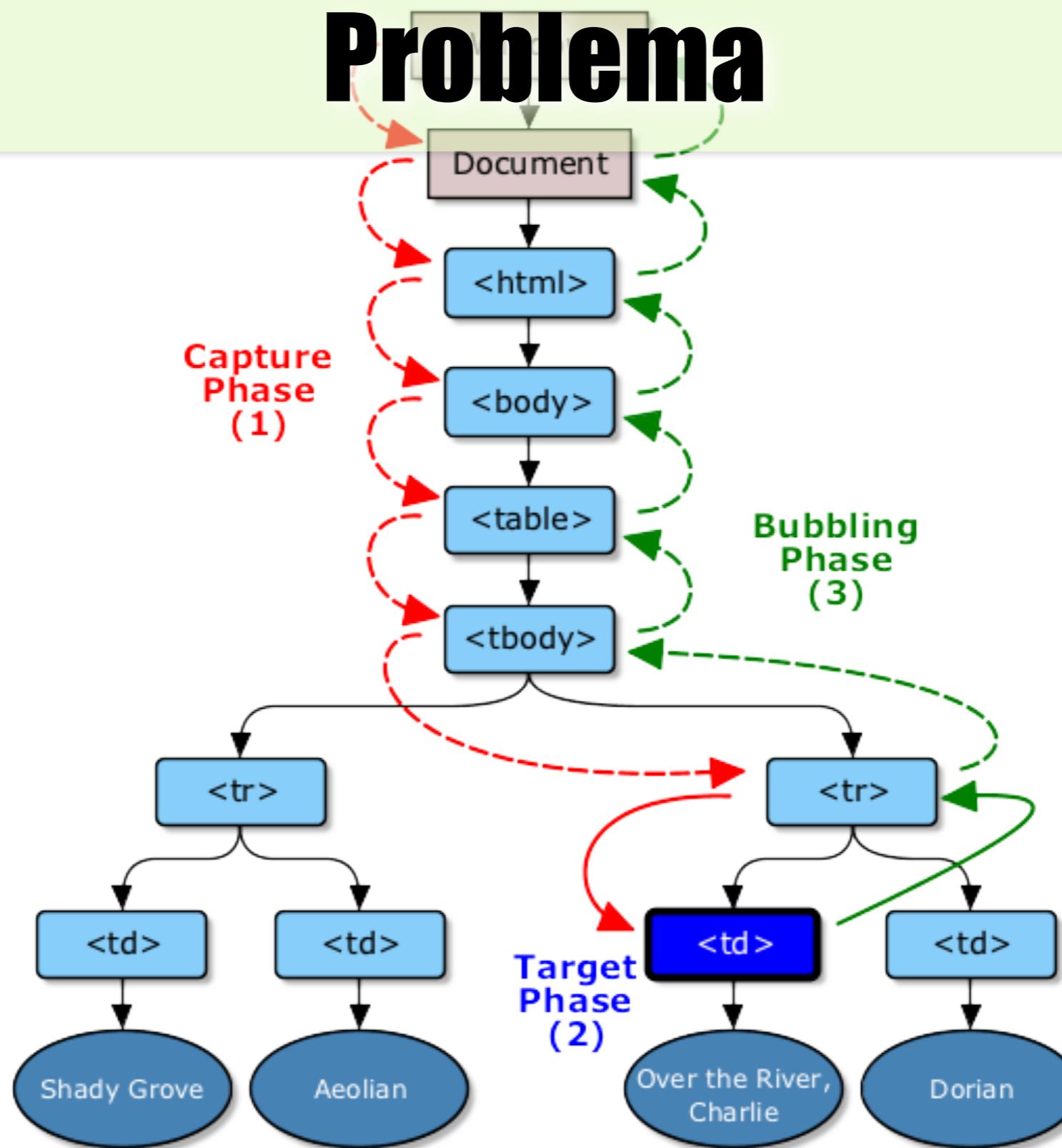
# APIS estándar vs. frameworks

- Hasta ahora, en la parte de teoría hemos visto lo que se puede hacer con los APIs nativos del navegador
- Muchas **funcionalidades**
  - **Comunicación con el servidor:** fetch API
  - **Manipulación dinámica del HTML y del CSS:** DOM
  - **Almacenamiento local:** local storage, IndexedDB, ...
- ¿Por qué muchos desarrolladores **no los consideran suficientes?**. Hay razones de distintos tipos...

Volvamos por un momento a mediados de los 2000



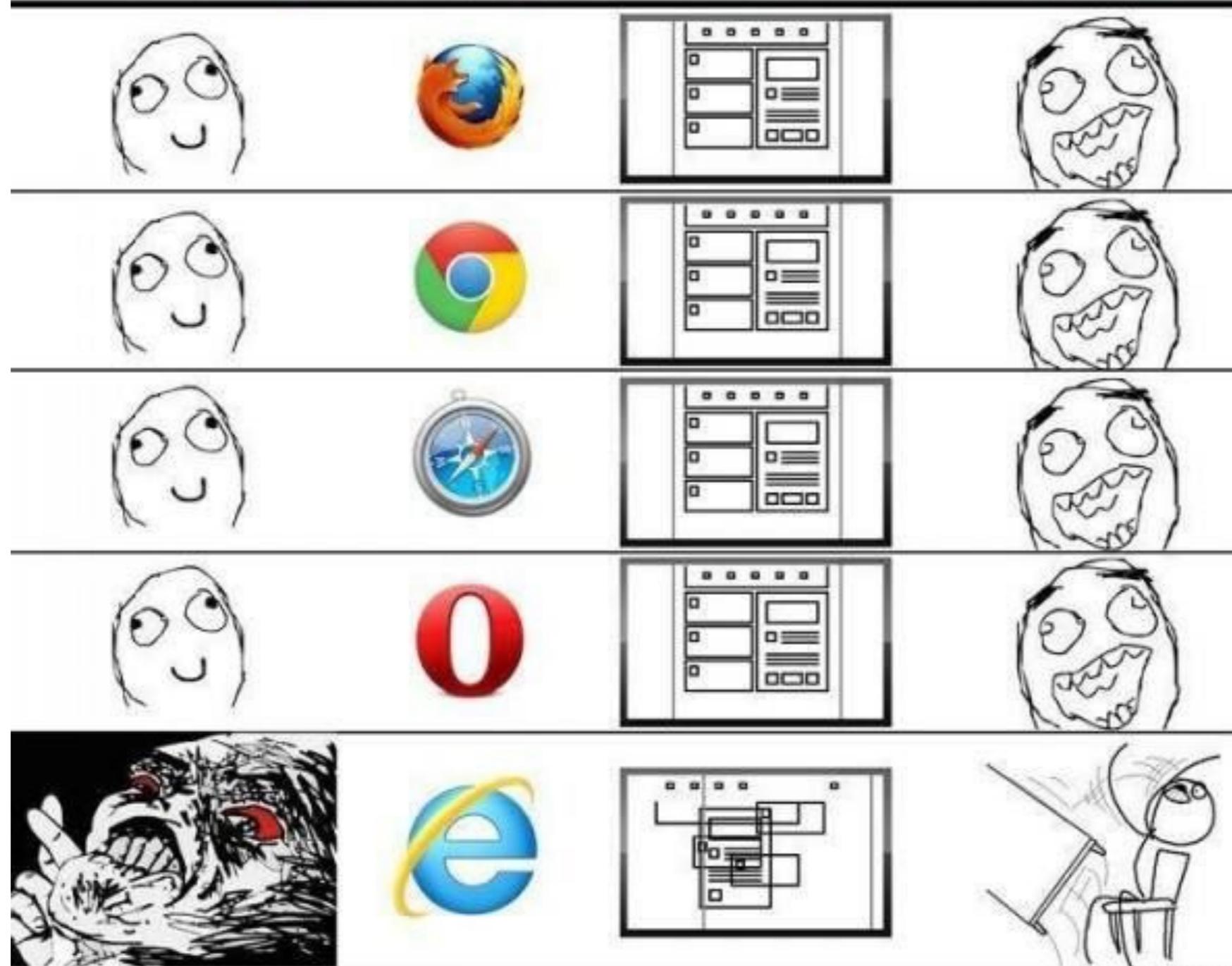
# Problema



## APIs web tediosos y complejos

# Problema

Testing web pages on different browsers



We Know Meme

# Incompatibilidades entre navegadores

**2006**



**La sensación: jQuery**

# La era de las librerías y toolkits en el frontend



*write less, do more.*



- Ofrecían un API más sencillo de usar y sin embargo más potente que el estándar
- Ofrecían elementos que no tenía el estándar (como widgets)
- Proporcionaban compatibilidad entre navegadores

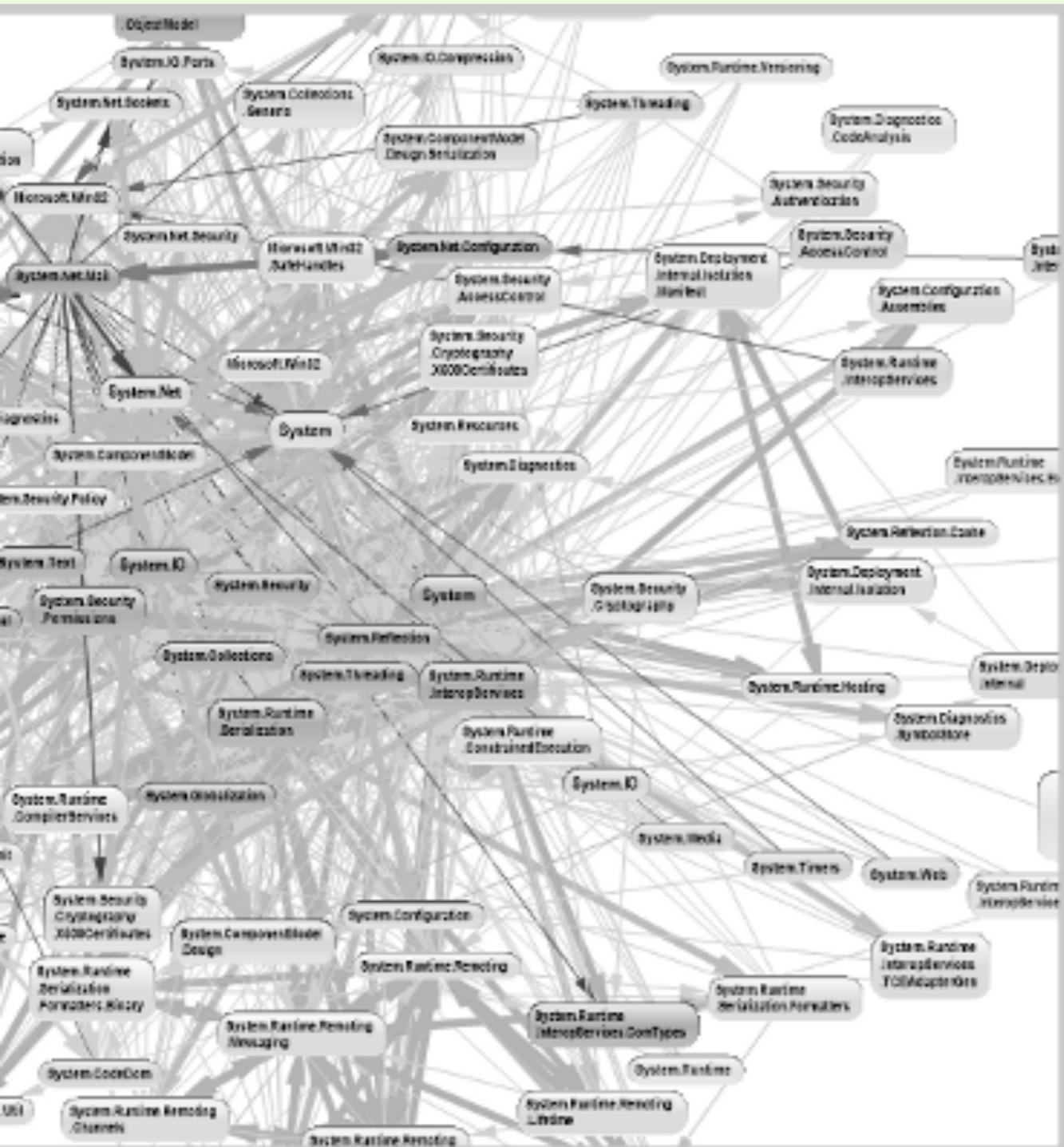
# Modernización de los APIs estándar

En los últimos años se han modernizado/simplificado/mejorado muchos APIs estándar

- `fetch` vs `XMLHttpRequest`
- `document.querySelector()` está claramente “inspirado” en jQuery
- `innerHTML` vs. la forma de trabajar nativa del DOM 1



# Problema

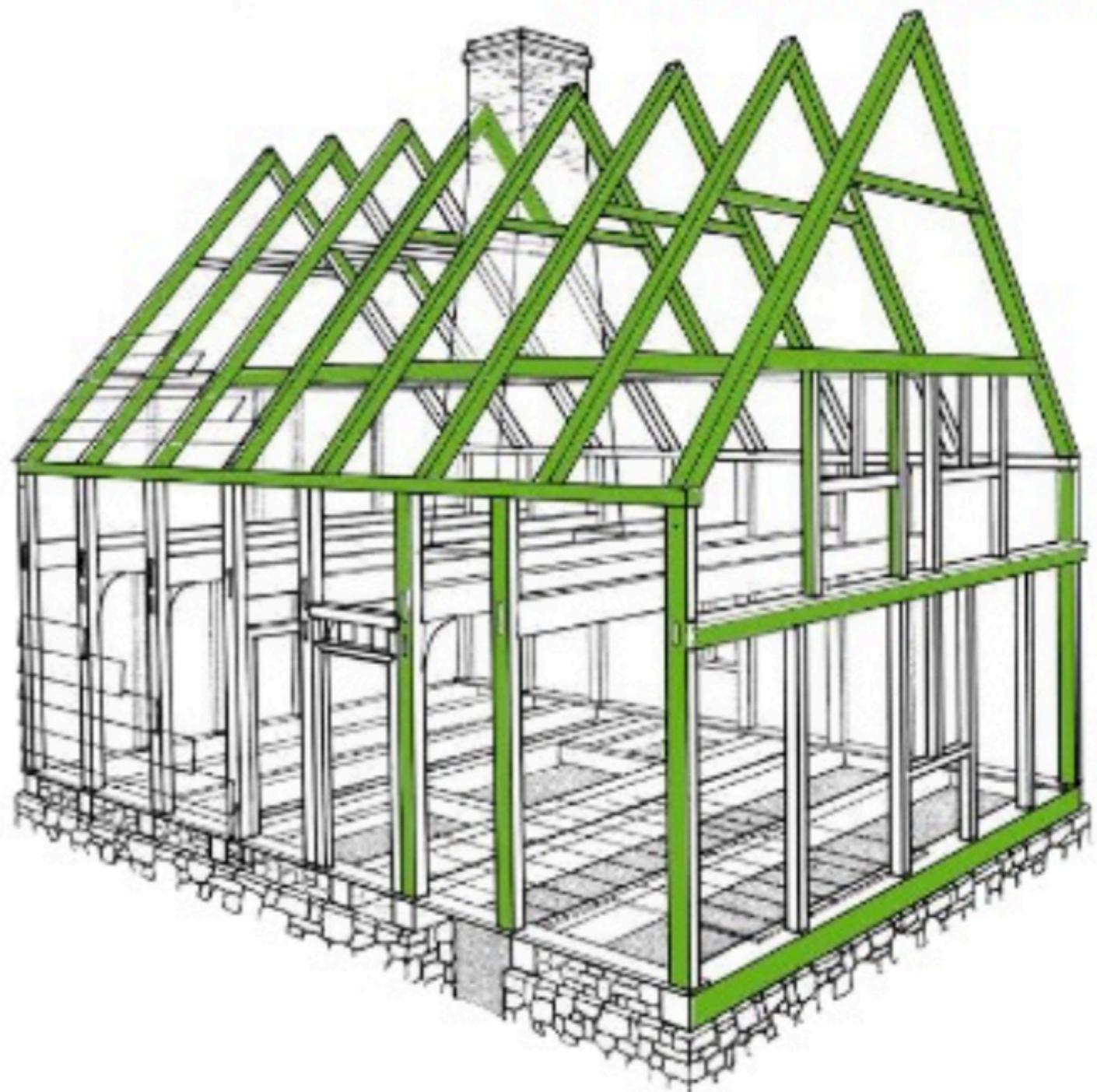


## Evitar el “big ball of mud”

# Código espaguetti

```
var item_sel = e.target
if (e.target.nodeName=='LI') {
    if (item_sel.classList.contains("tachado")) {
        item_sel.classList.remove("tachado")
        var res = await cli.toggleItem(e.target.id, false)
    }
    else {
        item_sel.classList.add("tachado")
        var res = await cli.toggleItem(e.target.id, true)
    }
}
if (e.target.nodeName=='BUTTON') {
    console.log('Boton: ' + e.target.id)
}
```

Los frameworks  
nos dan **una**  
**guía de cómo**  
**estructurar y**  
**organizar**  
**nuestro código**



# Problema

The screenshot shows a Gmail inbox with the following layout:

- Compose** button (red)
- Inbox (7)** link
- Starred**, **Drafts**, **Sent Mail** links
- Search people...** input field
- Primary** tab selected
- Social** tab (3 new): Google+, YouTube, Emi...
- Promotions** tab (2 new): Google Offers, Zagat
- Updates** tab (2 new): Shoehop, Blitz Air
- More** button
- Gmail** dropdown menu
- 1-100 of 2,067** link

The inbox list contains the following items:

- Google+**: You were tagged in 3 photos on Google+ - Google+ You were tagged in three photos in an album titled [REDACTED]
- YouTube**: LauraBlack just uploaded a video. - Jess, have you seen the video LauraBlack uploaded...
- Emily Million (Google+)**: [Knitting Club] Are we knitting tonight? - [Knitting Club] Are we knitting tonight?
- Sean Smith (Google+)**: Photos of the new pup - Sean Smith shared an album with you. View album be thoughtful about who
- Google+**: Kate Baynham shared a post with you - Follow and share with Kate by adding her to a circle. Don't know
- Google+**: Danielle Hoodhood added you on Google+ - Follow and share with Danielle by adding her to a circle.
- YouTube**: Just for You From YouTube: Daily Update - Jun 19, 2013 - Check out the latest videos from your char
- Google+**: You were tagged in 3 photos on Google+ - Google+ You were tagged in three photos in an album titled [REDACTED]
- Hilary Jacobs (Google+)**: Check out photos of my new apt - Hilary Jacobs shared an album with you. View album be thoughtful
- Google+**: Kate Baynham added you on Google+ - Follow and share with Kate by adding her to a circle. Don't know

# Desarrollo de Single Page Apps

# ¿Qué necesitamos para una SPA?

Los *frameworks* suelen ofrecer una serie de componentes/funcionalidades:

- ***Routing***
- **Modelos/Lógica de negocio/Gestión del estado de la aplicación**
- **Conexión con APIs externos**
- ***Rendering***

# Routing

La web está *basada en las URLs*. Queremos mapear la URL que ve el usuario con el código de nuestra app



## REACT ROUTER INTRODUCTION

```
<Switch>
  <Route
    path="/about"
    render={props => <About {...props} admin="Bean" />}
  />
  <Route path='/:user' component={User} />
  <Route component={NoMatch} />
</Switch>
```

# Modelos/Gestión del estado

**Gestionar el estado** de la app de manera efectiva, organizada y con código sencillo de depurar

# **Conexión con APIs externos**

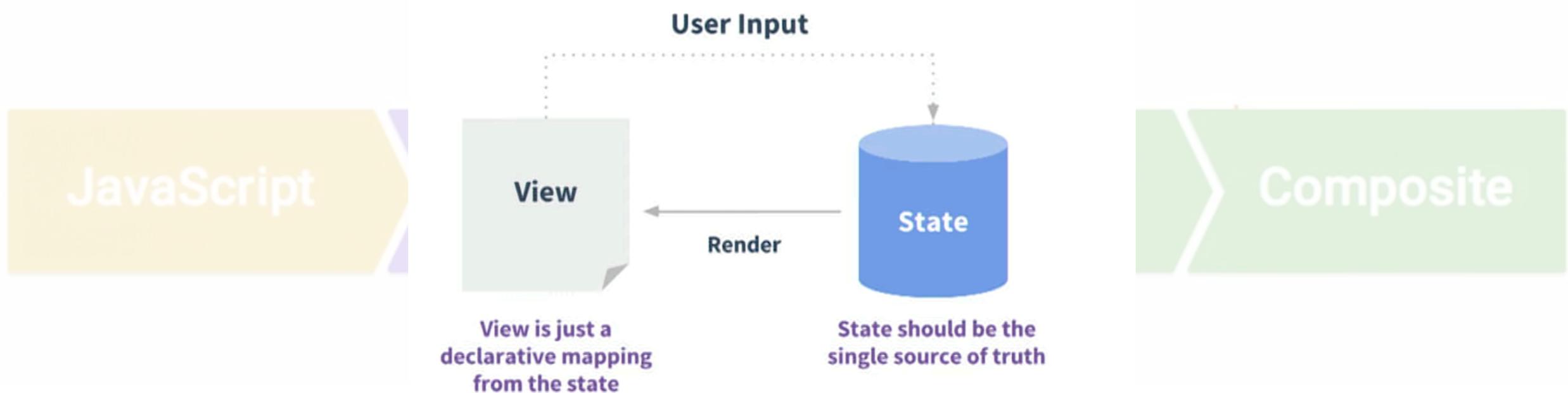
**Simplificar lo más posible el código** que nos conecta con **APIs REST**. O aún más: **sincronizar (semi)automáticamente** los datos en el cliente con los del servidor

# APIs externos en Backbone

```
var Libro = Backbone.Model.extend({});  
var Biblioteca = Backbone.Collection.extend({  
    model: Libro,  
    url: "/api/libros",  
})  
  
var miBiblioteca = new Biblioteca()  
  
//El fetch de Backbone lanza un GET a la url base  
miBiblioteca.fetch().then(function() {  
    miBiblioteca.get(1).set("titulo","Nuevo título")  
    var libro1 = new Libro({titulo:'Juego de tronos', anyo:1996});  
    miBiblioteca.add(libro1)  
    //libros modificados y ya existentes -> PUT a la url base + id  
    //libros nuevos -> POST a la url base  
    miBiblioteca.save()  
})
```

# Rendering

Poder modificar el HTML para **reflejar el estado actual** de la app de forma **sencilla para el desarrollador** y además **eficiente**



## **Tema 3: Frameworks web en el cliente**

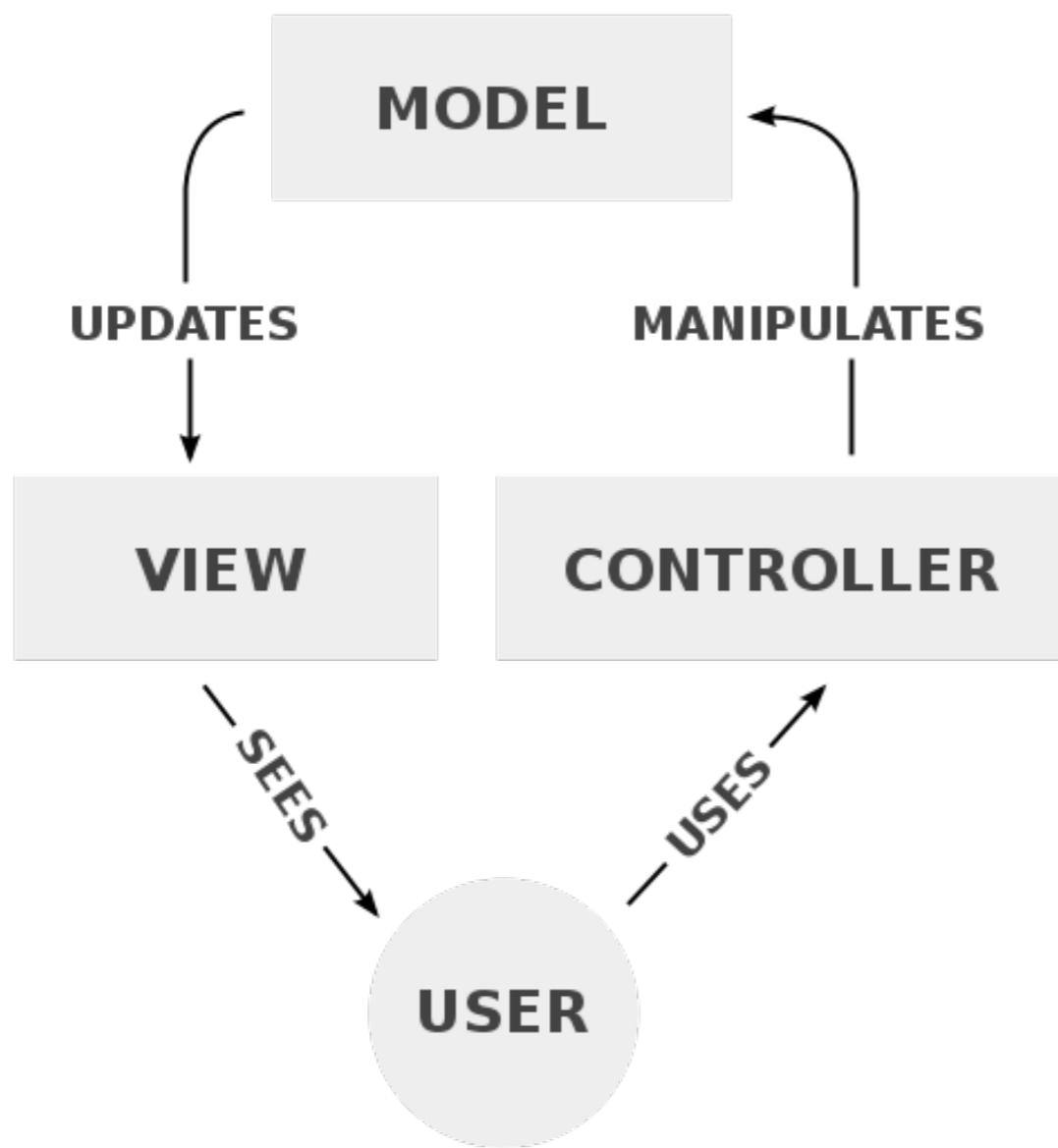
**3.2**

**2010-15**

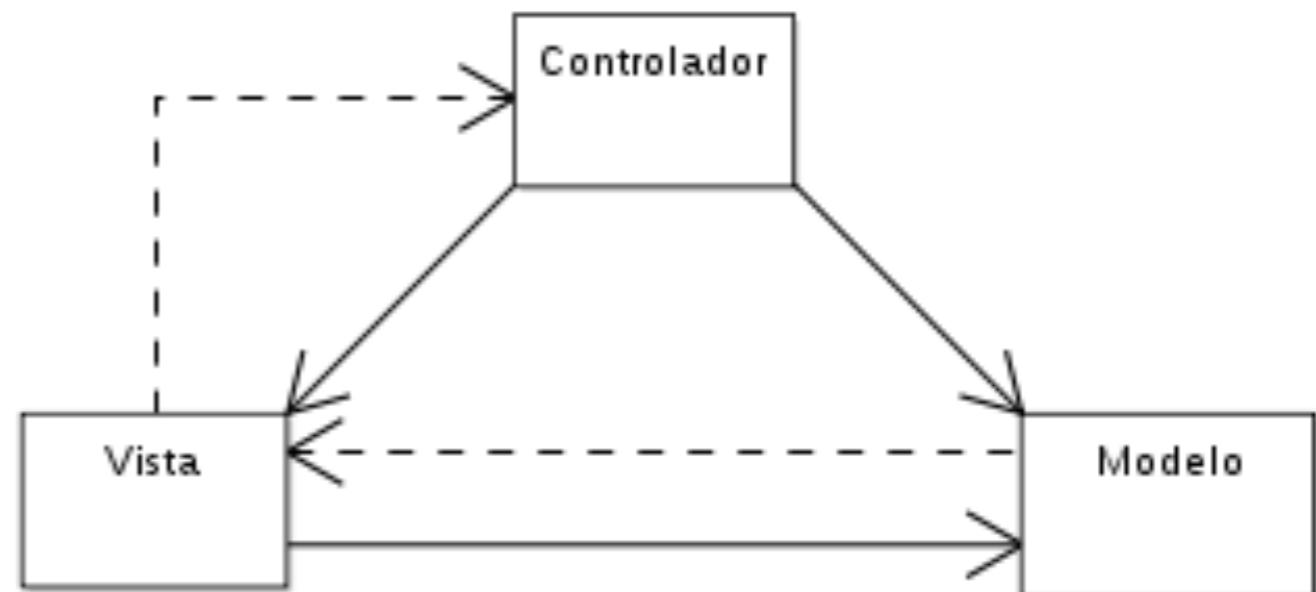
**Auge y caída de MVC**

# MVC

Patrón de diseño arquitectónico que **separa el modelo** del dominio **de su representación** en la interfaz (**vista**)



<http://en.wikipedia.org/wiki/Model-view-controller>



<http://es.wikipedia.org/wiki/Modelo-vista-controlador>

# MVC en el servidor web

**MVC** lleva siendo usado con éxito en aplicaciones web **en el lado del servidor** desde hace +20 años



Symfony



Struts



laravel



Para los quisquillosos: MVC tiene muchas variantes, muchos de estos *frameworks* no son MVC “puros” (suponiendo que tal cosa exista).



Igor Minar, *lead developer* de Angular 1.x, acuñó la expresión Model/View/Whatever para describir esto

Si MV\* funciona tan bien en el servidor...¿por qué no llevarlo también al **cliente**?

2010



La sensación: MVC en el cliente

# 2010-2015: La explosión de los frameworks MVC

Spine



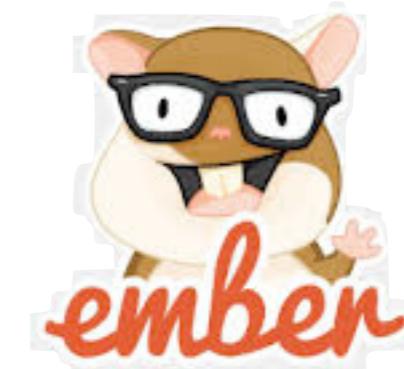
Maria

***Knockout.***



cujoJS

**canjs**



 **AGILITY.JS**

# MVC en Backbone



Ejemplo: [https://ottocol.github.io/widget-tiempo-frameworksJS/MVC/tiempo\\_backbone.html](https://ottocol.github.io/widget-tiempo-frameworksJS/MVC/tiempo_backbone.html)

Fuente: [https://github.com/ottocol/widget-tiempo-frameworksJS/blob/master/MVC/tiempo\\_backbone.html](https://github.com/ottocol/widget-tiempo-frameworksJS/blob/master/MVC/tiempo_backbone.html)



Backbone tenía un estilo muy Do-It-Yourself

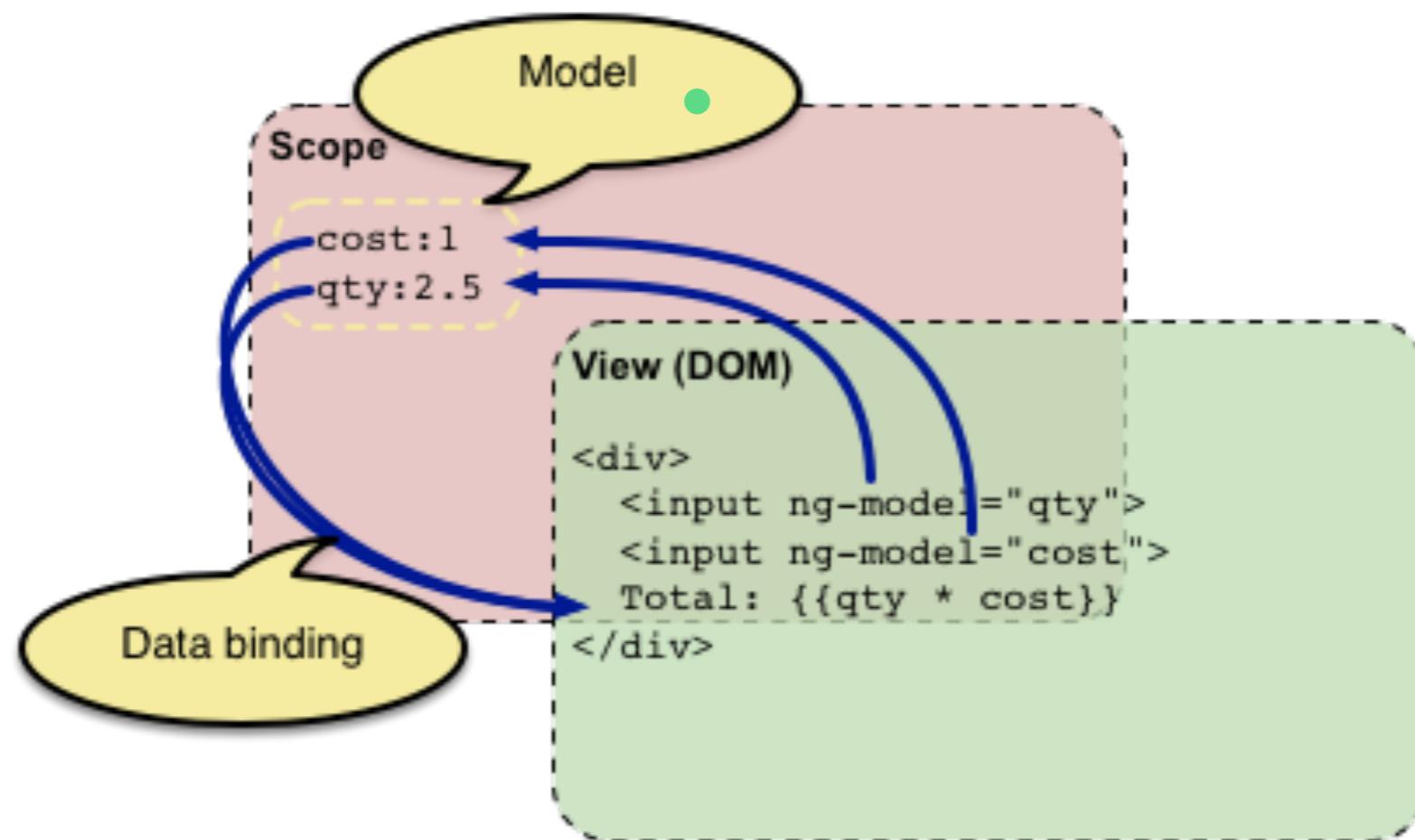
# AngularJS

- <https://angularjs.org>
- Se convirtió en el framework más popular de la “era MVC”
- **Todo-en-1:** muchas funcionalidades
  - Ventaja: realizar aplicaciones sencillas era también sencillo
  - Inconveniente: la curva de aprendizaje era más empinada para cosas más complicadas...



# Modelos

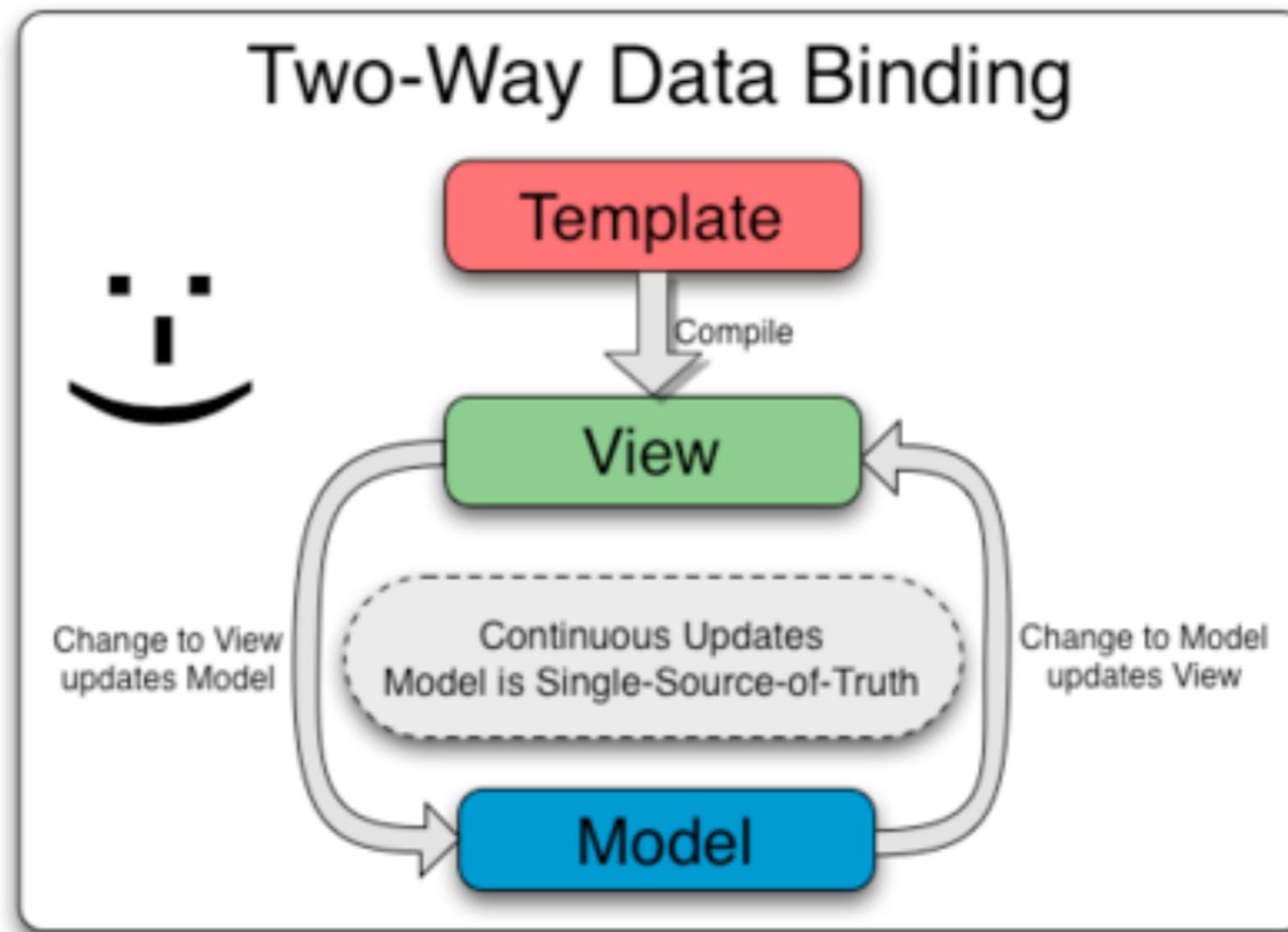
- POJOs (Plain Old Javascript Objects), o sea **objetos convencionales**
- **scope**: el “pegamiento” entre la vista y el modelo. Como el ViewModel de MVVM



<http://plnkr.co/edit/nV6oo9hFuL62gqeAuwx?p=preview>

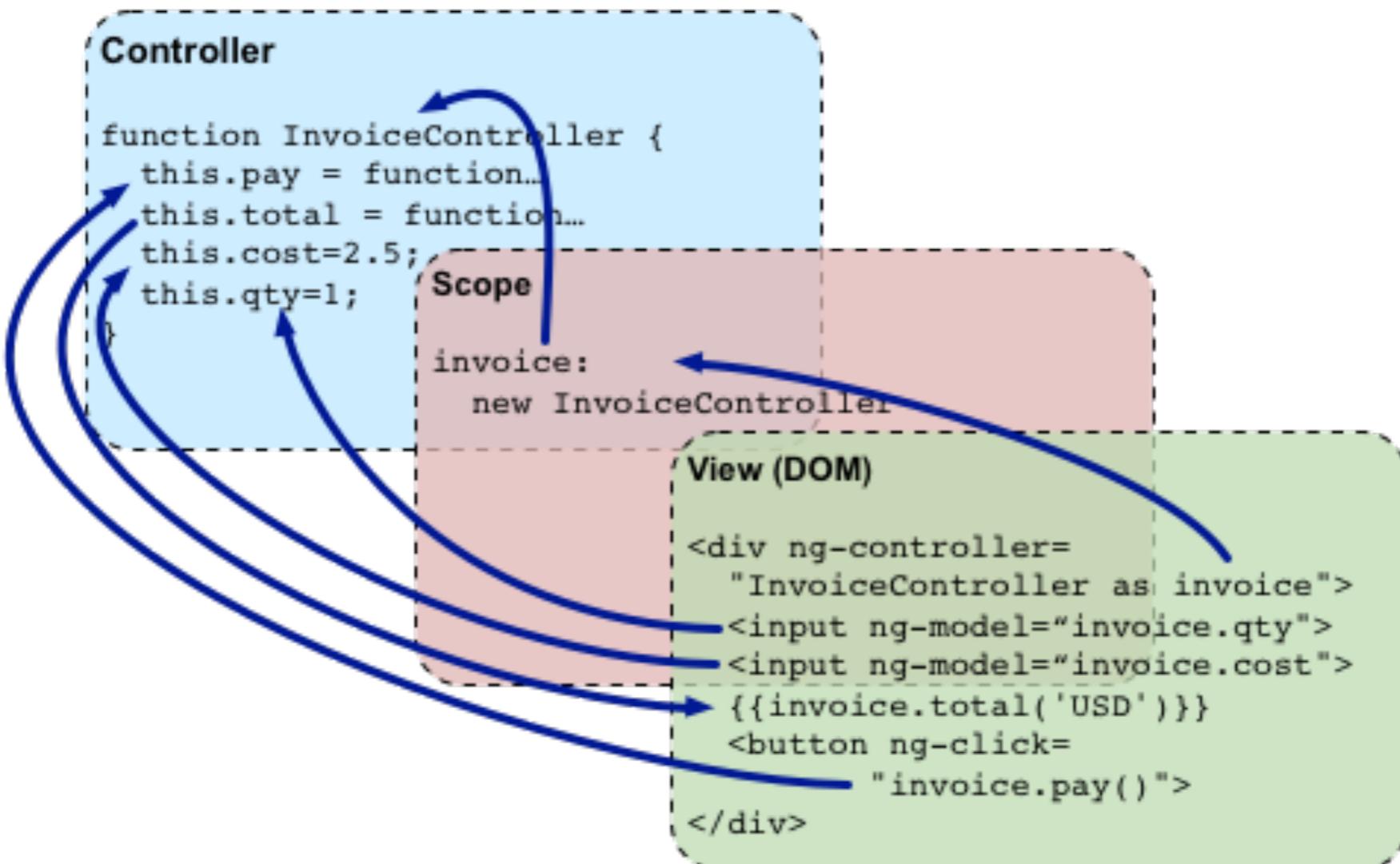
# Vistas

- AngularJS “extiende” el HTML con atributos propios (**directivas**)
  - Podemos definir nuestras propias directivas, como atributos o etiquetas HTML
- Plantillas:** muestran datos del modelo



# Controladores

- Donde ponemos la lógica de la aplicación

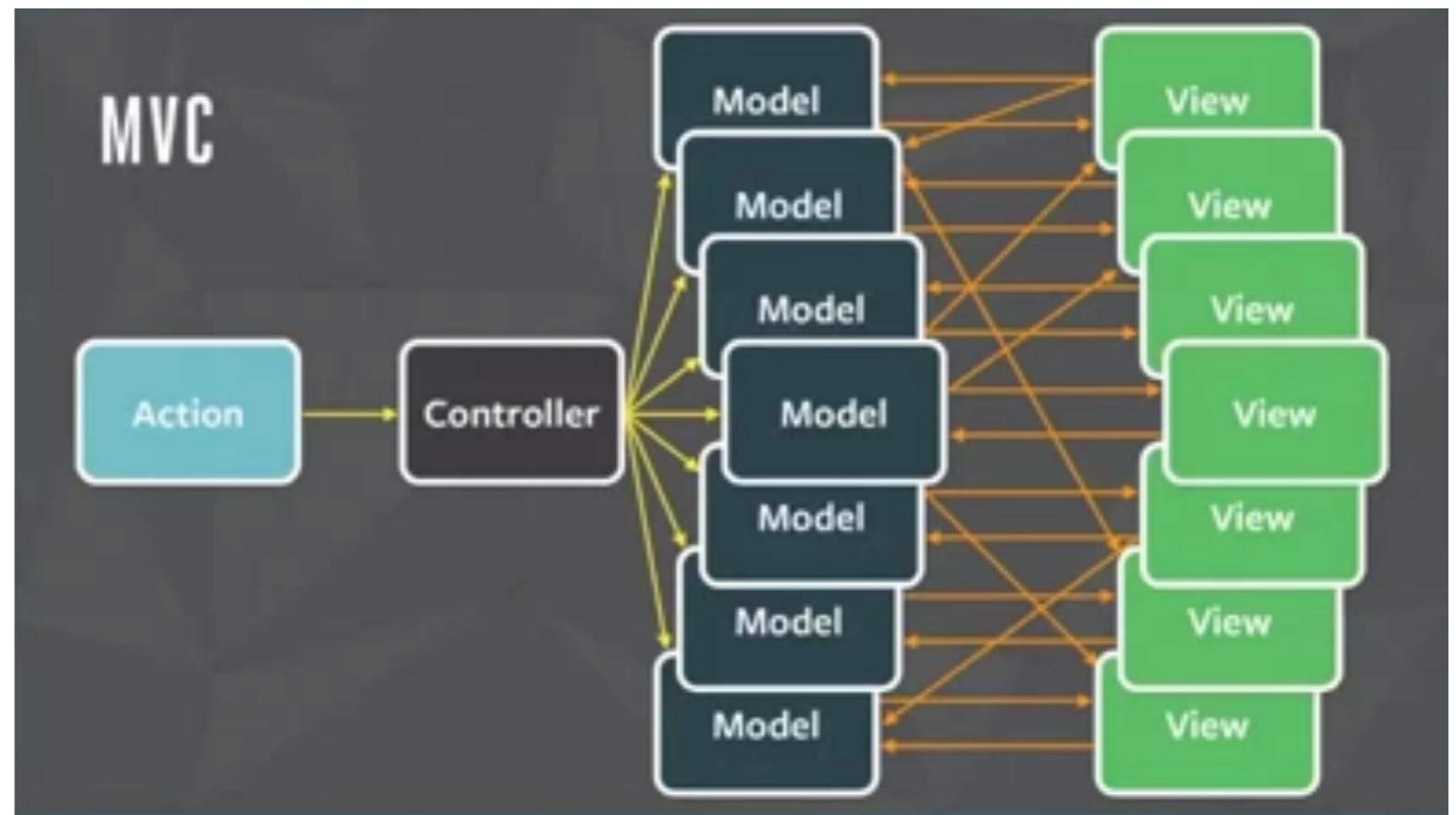


<http://plnkr.co/edit/Jf8D1IFCXoLw9ei7ISpS?p=preview>

Ejemplo del tiempo: [https://ottocol.github.io/widget-tiempo-frameworksJS/MVC/tiempo\\_angularJS.html](https://ottocol.github.io/widget-tiempo-frameworksJS/MVC/tiempo_angularJS.html)

# ¿Por qué “cayó” MVC en el frontend?

- Nunca estuvo muy claro cómo resolver la parte “C”
  - ¿Cuál es el papel exacto del controlador?
  - ¿Dónde colocamos la lógica de negocio? ¿Y la conexión con APIs externos?
- Tampoco estaba claro cuántos M, V y C debería tener una página
- Escalabilidad
- ...¿Moda?





**We will never forget your  
contribution**

**FRONTEND MVC  
2010 - 2016**

## **Tema 4: Frameworks web en el cliente**

**3.3**

# **Frameworks en la actualidad**

# React

A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES

[Get Started](#)[Download React v0.12.1](#)

## JUST THE UI

Lots of people use React as **the V in MVC**. Since React makes no assumptions about the rest of your technology stack, it's easy to try it out on a small feature in an existing project.

## VIRTUAL DOM

React uses a *virtual DOM* diff implementation for ultra-high performance. It can also render on the server using Node.js — no heavy browser DOM required.

## DATA FLOW

React implements one-way reactive data flow which reduces boilerplate and is easier to reason about than traditional data binding.

React popularizó varias ideas, entre ellas la de  
**Componentes**

# Conceptos comunes

Ideas presentes en la actualidad en la mayoría de frameworks

- **Componentes** como unidad básica de abstracción
- App como **jerarquía** de componentes
- “**Data Down, Actions Up**”
- Gestión **centralizada** del **estado**
- **Routing**: asociar URLs a componentes

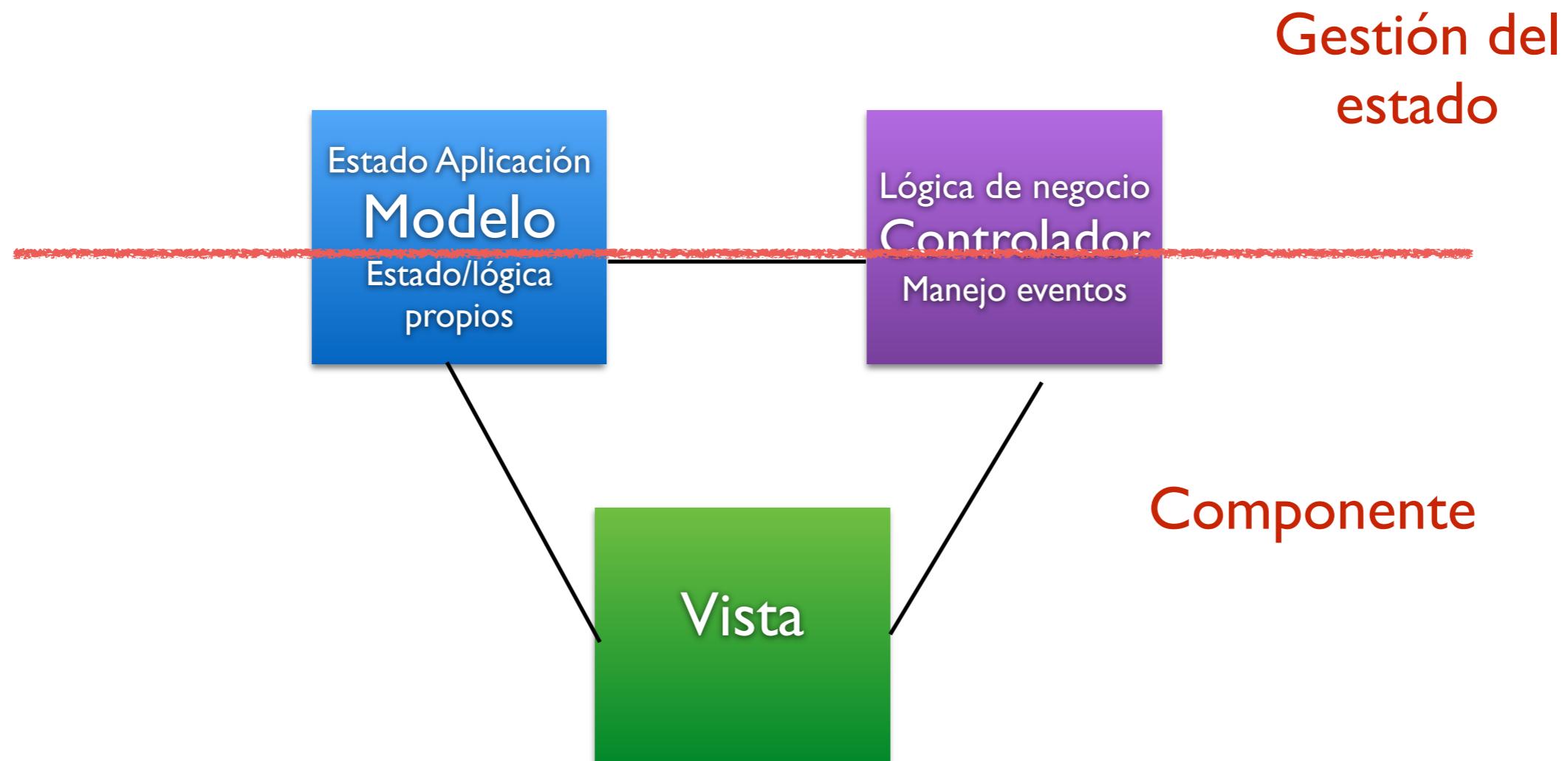
**2015**



**La sensación: Componentes**

# Componentes

- **Componente:** vista + manejo de eventos + lógica/estado propio



<https://medium.freecodecamp.org/is-mvc-dead-for-the-frontend-35b4d1fe39ec>

# Componente en Angular

Vista: HTML+CSS

```
@Component({
  selector: 'app-tarjeta-visita',
  templateUrl: './tarjeta-visita.component.html',
  styleUrls: ['./tarjeta-visita.component.css'],
})
export class TarjetaVisitaComponent {

  nombre = ""
  apellidos = ""
  puesto = ""

  constructor(private puestoService : PuestoService ) {}

  get nombreCompleto() {
    return this.nombre + " " + this.apellidos
  }

  generarPuesto() {
    this.puestoService.generarPuesto().subscribe(datos=> this.puesto = datos)
  }
}
```

Estado

Lógica

<https://stackblitz.com/edit/angular-zwu7ad>

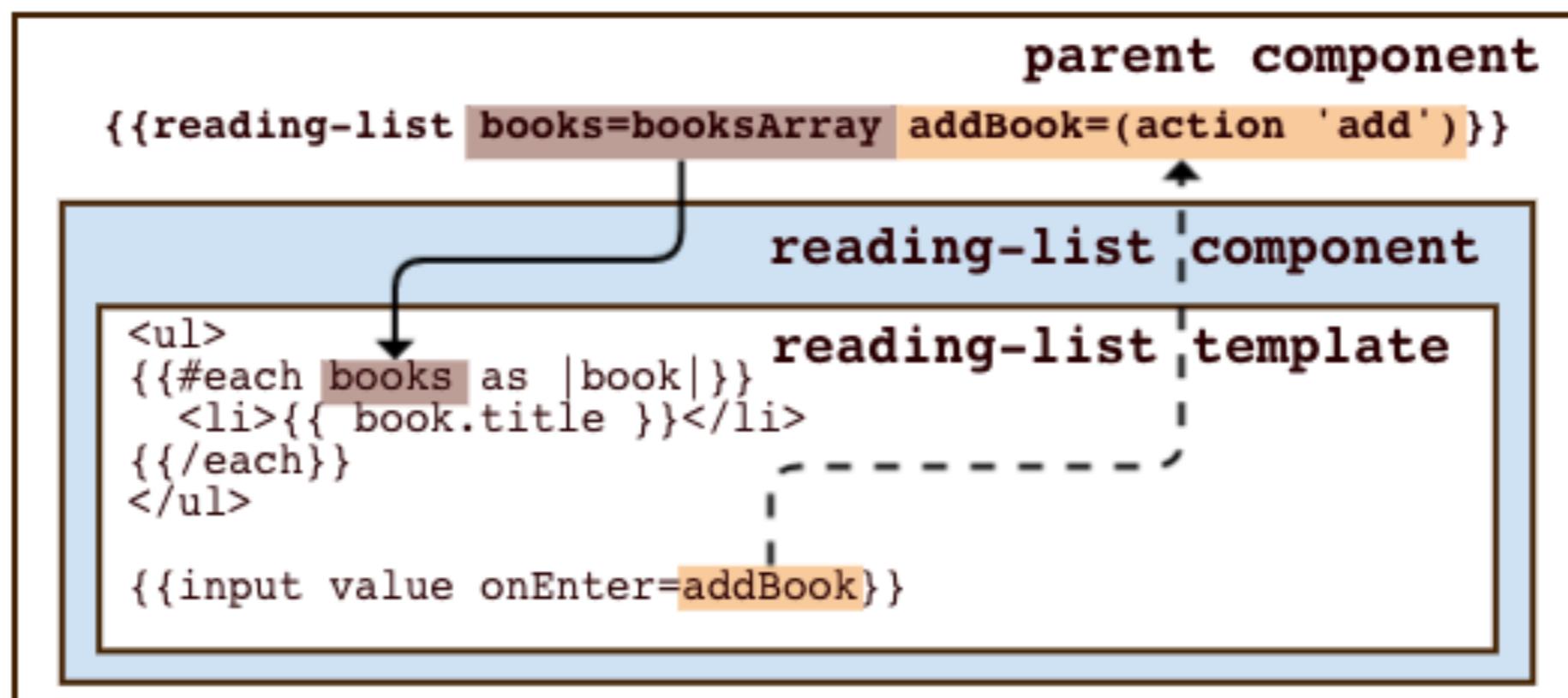
# Template del componente

## Manejo de eventos

```
<div>
  Nombre: <input type="text" [(ngModel)]="nombre"> <br>
  Apellidos: <input type="text" [(ngModel)]="apellidos"> <br>
  <button (click)="generarPuesto()">Generar puesto</button>
  <div>
    <div class="tarjeta">
      <h2>{{nombreCompleto}}</h2>
      <h3>{{puesto}}</h3>
    </div>
  </div>
</div>
```

# “Data Down, Actions Up”

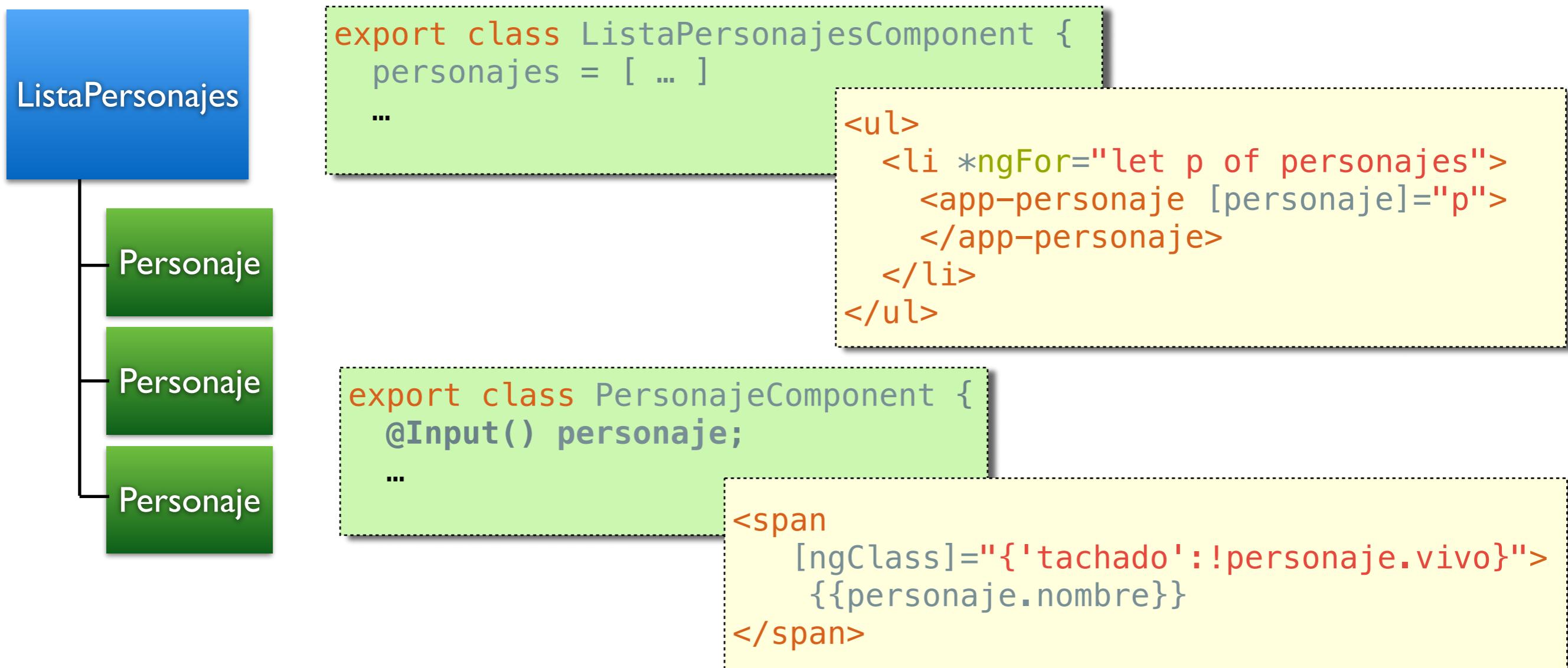
- Los **datos** pasan de componentes padres a hijos (**down**) y los **eventos** suben (**up**) hasta el componente capaz de procesarlos
- La frase del título es del framework **Ember** (<https://www.emberjs.com/>) pero también es así en **React, Angular, Vue,...**



De un tutorial de cómo hacer una app de una lista de libros pendientes en Ember siguiendo la idea de Data Down Actions Up <https://emberigniter.com/getting-started-ember-cli-data-down-actions-up-tutorial/>

# Data down

- Los componentes padres pasan a los hijos los datos necesarios para el *rendering*



Código simplificado (sin eventos) de  
<https://stackblitz.com/edit/angular-tqyzeu>

# Actions up

```
export class ListaPersonajesComponent {  
  personajes = [  
    {id:1, nombre:"Cersei Lannister",  
     vivo:true},  
    ...  
  ]  
  
  onChange(id) {  
    var pos = this.personajes.findIndex(  
      elemento=>elemento.id==id)  
    if (pos>-1) {  
      this.personajes[pos].vivo =  
        !this.personajes[pos].vivo  
    }  
  }  
}
```

⑤ Como no hay más niveles hacia arriba, este en lugar de emitirlo de nuevo, lo procesa, en este caso cambiando el estado del componente. El

**“Data Down” vuelve a comenzar**

```
export class PersonajeComponent {  
  @Input() personaje;  
  @Output() onChange =  
    new EventEmitter<number>()  
  
  ...  
  
  cambiarStatus() {  
    this.onChange.emit(this.personaje.id)  
  }  
}
```

③ se genera un evento “onchange” definido por el desarrollador, no es del navegador

```
<ul>  
  <li *ngFor="let p of personajes">  
    <app-personaje  
      [personaje]="p"  
      (onChange)="onChange($event)">  
    </app-personaje>  
  </li>  
</ul>
```

④ El evento lo captura el nivel inmediatamente superior.

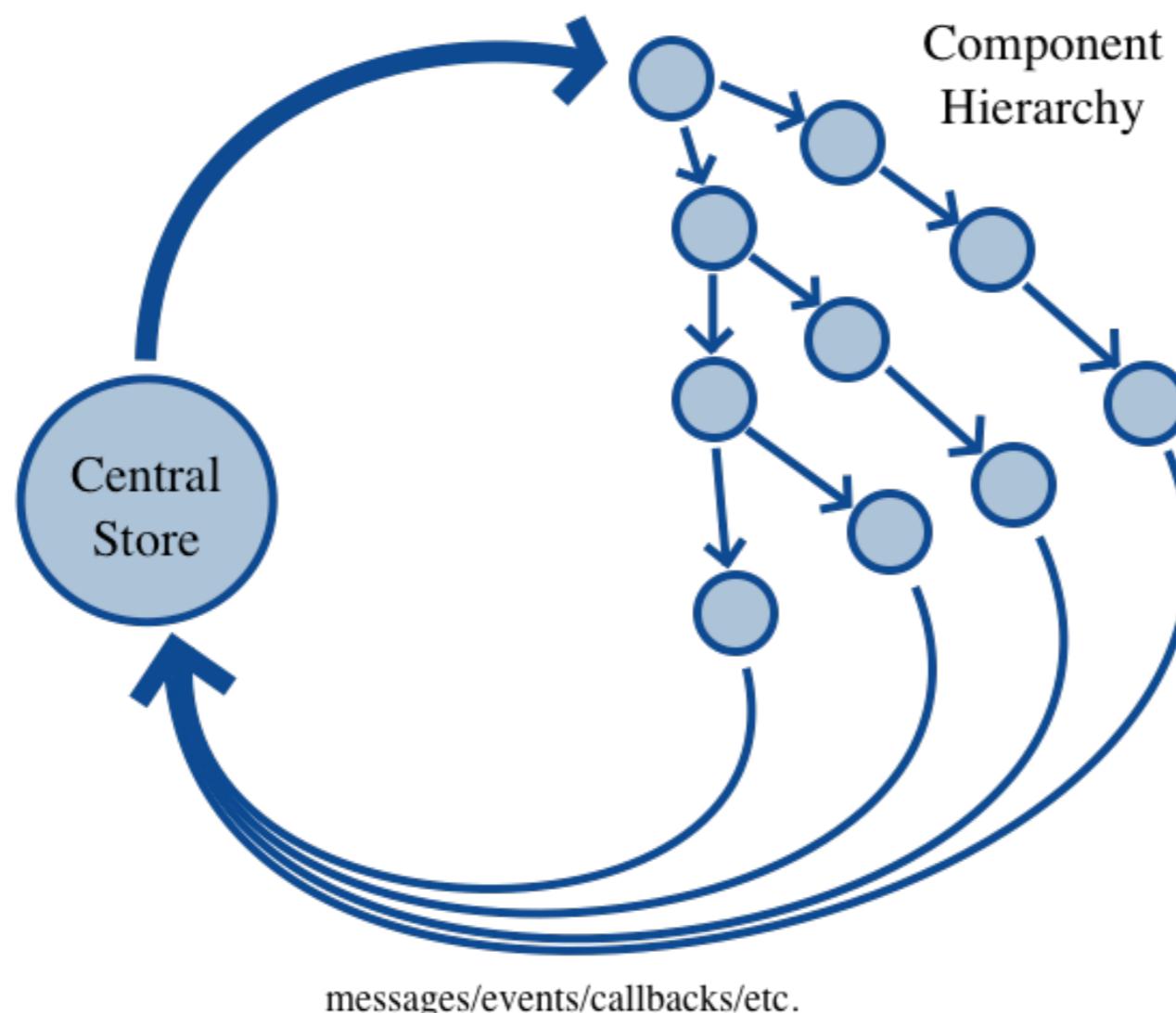
```
<span [ngClass]="{{'tachado' : !personaje.vivo}}"  
      (click)="cambiarStatus()"  
      {{personaje.nombre}}>  
</span>
```

② El manejador de evento llama a un método del componente

① El usuario hace clic en un personaje

# Gestión centralizada del estado

- Ya hemos visto la filosofía de subir el estado a los componentes de nivel superior
- Ir un paso más allá: sacar el estado a un “almacén externo”



La gestión “centralizada” del estado se ha convertido en **un estándar** en **React** (la arquitectura es original de Facebook), pero también muy usada en **Angular** y **Vue**. La variante más conocida se llama **Redux**



<https://github.com/reactjs/react-redux>



<https://github.com/angular-redux/ng->



<https://github.com/vuejs/vuex>