UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Cuore.js: Implementación de web framework siguiendo la arquitectura de servicios

Proyecto final de carrera de Ingeniería Informática

Bogdan Dobrev Gochev

DIRIGIDO POR: Cèsar Ferri Ramírez

Abril 2014

Resumen

Las aplicaciones Web se han convertido en una de las herramientas más importantes de nuestra sociedad y una parte íntegra de ella. La cada vez creciente demanda de aplicaciones Web por sectores muy distintos y el crecimiento de complejidad en sus requerimientos hace necesario la aparición de herramientas cada vez más robustas y flexibles.

Las herramientas y métodos clásicos de desarrollo no son suficientes para la construcción de aplicaciones Web modernas.

El crecimiento en la complejidad e infraestructura de las aplicaciones Web conlleva a la aparición de nuevos frameworks y herramientas que permiten el desarrollo de estos sistemas más complejos.

Por otro lado la creciente expansión de internet hace que estas aplicaciones Web tengan que escalar para abastecer a millones de usuarios.

Cuore.js es un framework que permite desarrollar aplicaciones Web modernas y escalables. Su diseño está pensado para simplificar el desarrollo de aplicaciones complejas, permitiendo trabajar de una manera más estructurada y rápida.

Por otro lado la escalabilidad forma parte del núcleo de Cuore.js, por eso cualquier sistema desarrollado con él, puede disfrutar de una gran escalabilidad que es limitada únicamente por los requisitos del sistema a desarrollar y no por el framework.

Gracias a su arquitectura modular Cuore.js permite que diferentes de personas trabajen sobre diferentes partes de una aplicación Web sin interferir unos con otros, lo cual es muy útil en proyectos más grandes.

Por último en el desarrollo de Cuore.js se han seguido técnicas como TDD, que hacen que el framework disponga de una gran batería de pruebas que demuestran su madurez y buen funcionamiento.

Cuore.js está escrito en JavaScript y tiene una licencia MIT, así que está disponible libremente para su distribución y uso en aplicaciones comerciales.

Primero quiero agradecer a mis padres su ayuda y apoyo incondicional en todo!

También quiero dar las gracias a mi tutor Cèsar Ferri. Por su ayuda y consejos en la realización de este PFC.

Por último quiero dar las gracias a Xavier Gost, de la empresa BeCodeMyFriend. Sin su gran ayuda este proyecto nunca podría haberse realizado.

Tabla de Contenidos

1 Introducción

1.1 Motivación

Antes de empezar a hacer una aplicación Web hay que pensar en que herramientas y frameworks se van a usar para llevar a cabo nuestro trabajo.

Hoy en día existen multitud de frameworks y librerías que nos prometen que son fáciles de utilizar, potentes, flexibles, multi-plataformas, robustas, etc. Pero la realidad es que no podemos evaluar estas herramientas tan solo por sus características técnicas. La complejidad de las aplicaciones y las franjas temporales que tenemos para desarrollar nuestro proyecto y nuestro dominio tecnológico, muchas veces, juegan papeles más importantes que las características técnicas y los largos documentos de especificaciones de las herramientas anteriores.

Todo esto nos lleva a pensar que no necesariamente el framework con la mayor cantidad de características es el más adecuado ya que es posible que su complejidad no aporte nada al proyecto, y sin embargo tenemos que gastar mucho tiempo aprendiendo.

Por otro lado no siempre la herramienta más fácil sería la más adecuada, ya que por lo general suelen ser tecnologías que imponen sus principios de hacer las cosas. Esto puede tener graves consecuencias si estamos intentando hacer algo diferente de lo común y nos estamos continuamente peleando con la herramienta.

De estos dos ejemplos podemos ver que no existe una o unas herramientas perfectas para todos los casos. Tenemos que ser muy cuidadosos a la hora de planificar nuestra aplicación Web, para evitar posibles problemas en el futuro.

Por estos motivos hemos optado por hacer un framework que se adapte perfectamente a nuestra forma de pensar y nuestro tipo de proyectos. Un framework que utiliza una combinación de aquellas características que son valiosas para nosotros y que se pueden encontrar dispersas en diversos proyectos.

1.1.1 Problemas en técnicas clásicas de desarrollo Web

Los problemas con el desarrollo de aplicaciones Web modernos utilizando técnicas clásicas son muchos.

- **Complejidad del backend**. Las aplicaciones Web son cada vez más complejas y dinámicas. Requieren cada vez más comunicación con los servidores centrales y son cada vez partes de una infraestructura más global que incluye también aplicaciones móviles. Por todo eso el backend tiene que ser agnóstico en la forma que se presenta la información y computar solamente lógica de negocio. Esto permitiría aprovechar el mismo backend para diferentes aplicaciones Web/móviles.

- **Frontend cada vez más dinámico.** Las interfaces de usuario de las páginas web son cada vez más dinámicos, requieren menos refrescos globales de la página e incorporan entornos gráficos completos. Esto hace que la presentación ya no sea parte del backend sino que se ejecute directamente en el navegador del cliente.

- **Técnicas estructuradas de desarrollo.** Las aplicaciones web, como cualquier otro software, requieren la aplicación de buenas técnicas de desarrollo software. Algo que durante mucho tiempo ha faltado en la industria. En los últimos años se ha visto una expansión y adopción de buenas técnicas en el campo de las aplicaciones Web.

- **Escalabilidad.** Las aplicaciones Web ya no son simples aplicaciones monolíticas sino que son aplicaciones distribuidas que se ejecutan en una gran variedad de máquinas, desde móviles hasta servidores grandes. Esto obliga ejecutar diferentes partes de la aplicación en diferentes lugares físicos, y abastecer la creciente demanda por parte de los usuarios.

1.2 Contexto

Este trabajo se ha desarrollado en el contexto del Departamento de Sistemas Informáticos y Computación (DSIC), basándose en trabajo previo realizado en la empresa BeCodeMyFriend.

1.3 Objetivo

Este proyecto final de carrera tiene como objetivo hacer un análisis en el desarrollo de un framework web llamado Cuore.js, explicando en detalle todos los diferentes pasos que se han tomado y las decisiones que han dictado la implementación.

Por otro lado el objetivo es también presentar nuevos métodos y técnicas de desarrollo de aplicaciones web dinámicas, distribuidas y SPA (Single-Page Application).

2 Premisas de Cuore.js

Antes de empezar con el desarrollo del proyecto hace falta especificar sus cualidades o requisitos, tanto técnicos como no técnicos. En el caso de Cuore.js se tenía muy claro qué cualidades se estaban buscando y el porqué de ellas.

2.1 SOA (Service Oriented Architecture)

SOA significa un acoplamiento débil entre sistemas. Acoplamiento débil significa la eliminación implícita dependencias entre sistemas y declarando todas las dependencias legítimas en la forma de contratos explícitos entre ellos. (Se han subrayado los términos cruciales.)

Las aplicaciones que utilizan la arquitectura SOA se componen de agentes de software discretos que tienen interfaces simples y bien definidas y que se comunican mediante un acoplamiento débil para realizar una función requerida.

Hay dos funciones o roles en una arquitectura SOA - un proveedor de servicios y un consumidor de servicios. Un agente de software puede desempeñar ambos papeles.

- Un proveedor es una función que brinda un servicio en respuesta a una llamada o petición desde un consumidor.
- Un consumidor es una función que consume el resultado del servicio provisto por un proveedor.

Un servicio en la arquitectura SOA es una implementación de una función de negocio claramente definido que funciona independientemente del estado de cualquier otro servicio. Cuenta con un conjunto bien definido de interfaces independientes de la plataforma y opera a través de un contrato pre-definido con el consumidor del servicio. Los servicios están débilmente acoplados - un servicio que no tiene porqué conocer los detalles técnicos de otro servicio para poder trabajar con él - toda la interacción se lleva a cabo a través de las interfaces.

La comunicación entre el consumidor y el servicio se realiza en formato de datos XML o JSON a través de una variedad de protocolos. Los principales protocolos de servicios web utilizados hoy en día son SOAP (Simple Object Access Protocol) y REST (Representational State Transfer). Mientras que REST utiliza la infraestructura existente de Internet (HTTP), SOAP es independiente de la capa de red y puede usar una variedad de protocolos de red como HTTP, SMTP y similares.

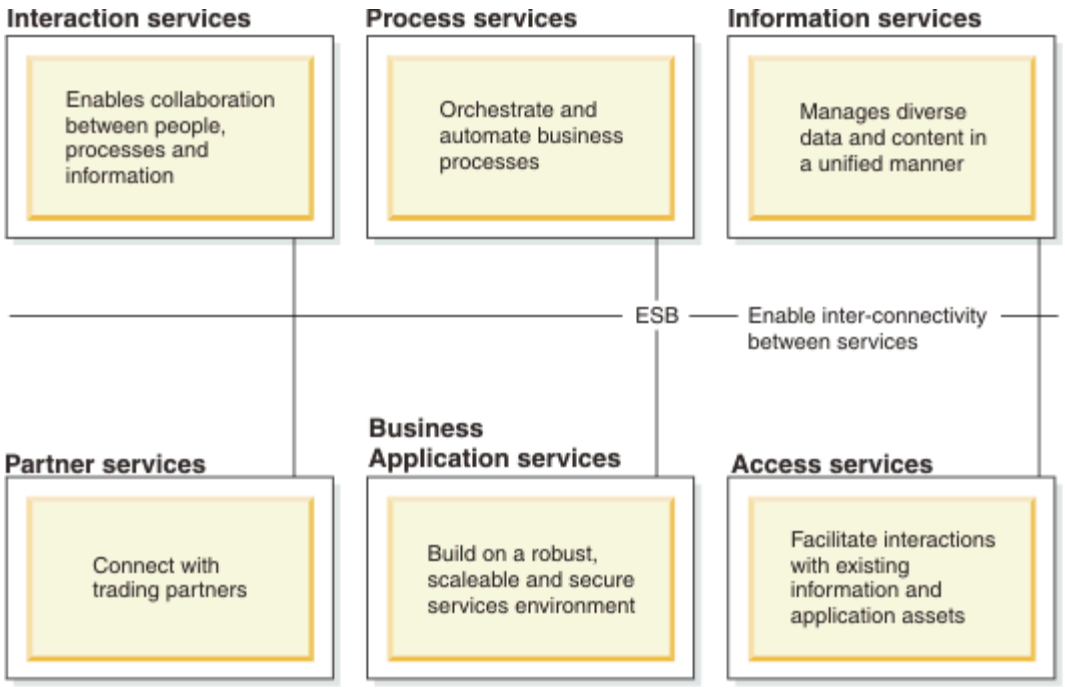


Diagrama del uso clásico de SOA en una organización

2.2 SOFEA (Service-Oriented Front-End Architecture)

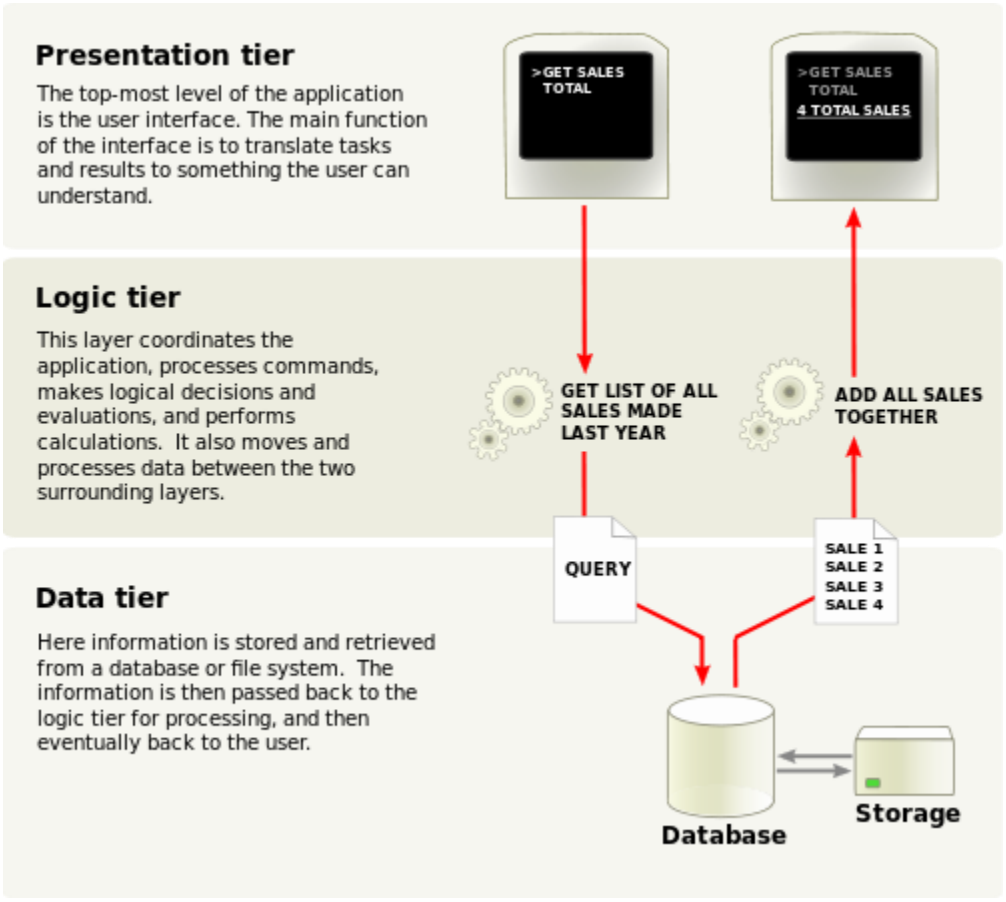
Una de las arquitecturas más extendidas en el mundo de las aplicaciones web es la programación por capas o arquitecturas multinivel y más en concreto una solución de 3 capas. Esta solución se base en :

1. **Capa de presentación** : Es la interfaz con la que interactúa el usuario del sistema. También es conocida como interfaz gráfica y debe tener la característica de ser "amigable" (entendible y fácil de usar) para el usuario. Esta capa se comunica exclusivamente con la capa de negocio.

2. **Capa de negocio** : En esta capa se reciben las peticiones del usuario y se envían las respuestas. También es la capa que se ocupa de hacer la validación de los datos, el cumplimiento de las reglas del negocio y la ejecución de programas. Esta capa se comunica con la capa de presentación para recibir y enviar información y con la capa de datos para almacenar u obtener datos.

3. **Capa de datos** : En esta capa residen los datos y es la encargada de acceder a los mismos. Está formada por uno o más gestores de bases de datos que realizan todo el

almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.



Arquitectura de tres capas

Normalmente la capa de Datos y Lógica se encuentra en los servidores de la organización y la capa de presentación está en el cliente. Sin embargo esta arquitectura de cliente ligero (Thin Client) tiene 3 fallos graves:

1. **Los datos viajan en formato plano** sin ninguna estructura, por ejemplo los datos que se envían entre un servidor y un cliente no respetan ninguna estructura y son simples campos de nombre-valor.

2. **El acoplamiento del flujo de presentación y el intercambio de datos**, es decir no es posible moverse por el flujo de presentación sin iniciar un intercambio de datos entre el servidor y el cliente. Un ejemplo de este problema se presenta con el botón de “Atrás” del navegador en algunas páginas. Hay patrones como POST-Redirect-GET que solucionan ese problema, pero más que una solución son un hack para arreglar un sistema fundamentalmente roto.

3. **Sin soporte para intercambio de datos Peer-To-Peer**, por ejemplo el servidor responde solo a peticiones del cliente, sin posibilidad de avisos o notificaciones iniciados por su parte.

Como una posible solución muchas personas pueden caer en la tentación de pensar que AJAX es la solución a estos problemas. Sin embargo eso no es así, porque AJAX es tan solo una tecnología cruda, con la que podemos seguir construyendo aplicaciones que están rotas y tienen algunos de los fallos nombrados anteriormente.

Utilizando AJAX no impone ninguna restricción en el intercambio de datos o en la forma de mezclar flujo de datos y de presentación, confirmando que es tan solo una tecnología cruda y no un framework o un patrón de diseño.

El problema se puede observar en el siguiente diagrama, donde sigue apareciendo el problema de datos en formato plano, el flujo de datos y presentación está mezclado, etc. Sin embargo hemos utilizamos AJAX sin solucionar los problemas antes comentados.

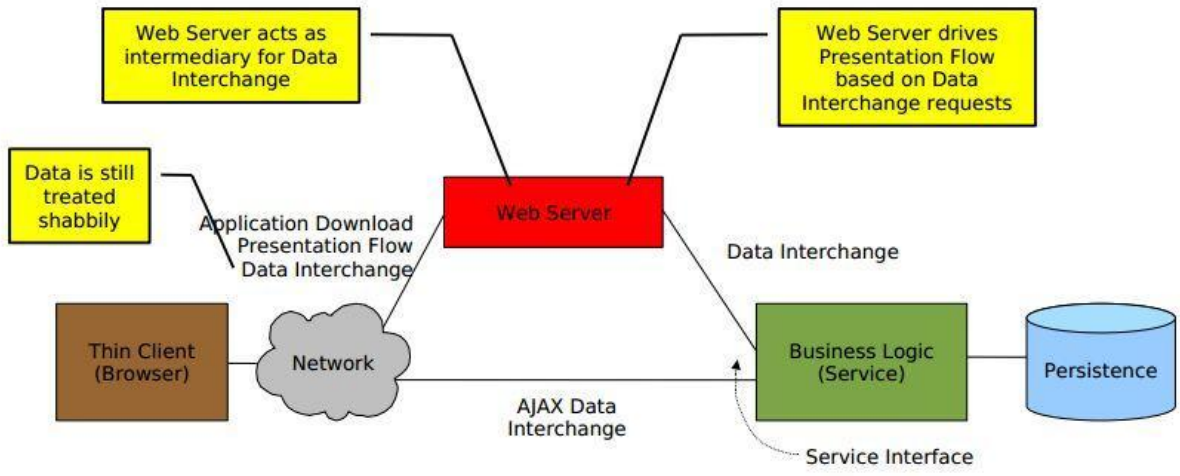


Diagrama de un uso incorrecto de AJAX

Como verdadera solución a los tres problemas presentados anteriormente surge SOFEA, que intenta aplicar las técnicas de SOA (Service Oriented Architecture) a la capa de presentación.

La primera diferencia de SOFEA y un cliente ligero (Thin Client) es que existe un servidor cuya única misión es distribuir nuestra aplicación y no participa en posteriores comunicaciones de datos.

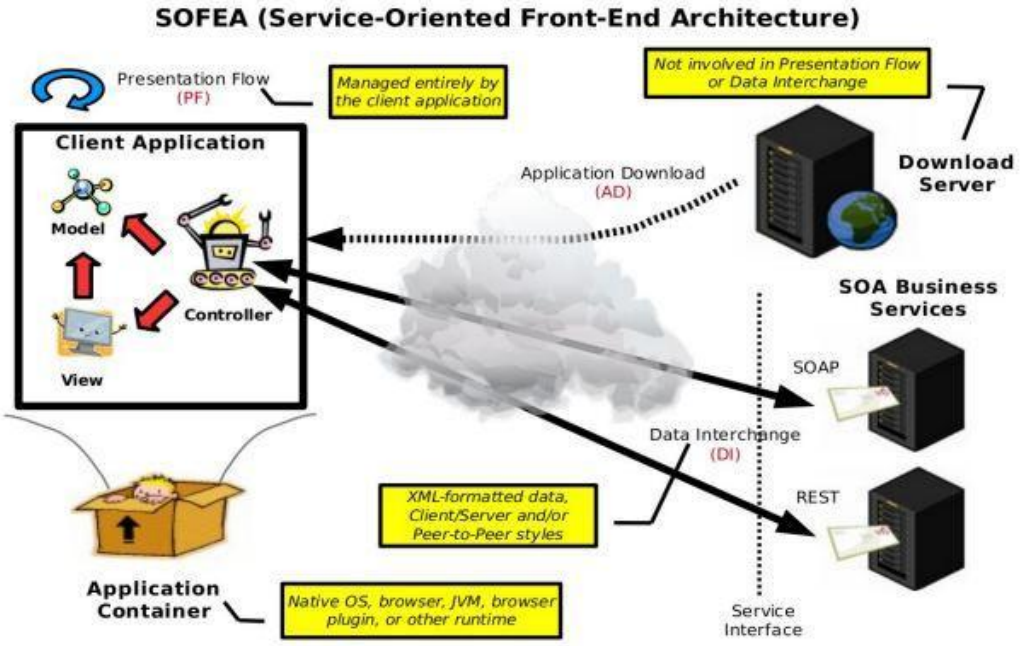
Una vez descargada, nuestra aplicación SOFEA se encarga de respetar el formato de los datos, sin convertirlos en una estructura plana.

Por otro lado se encarga también de la separación de los conceptos de flujo de datos y flujo de presentación y permite la comunicación P2P, es decir, una comunicación que puede ser iniciada bidireccionalmente, tanto por el cliente como por el servidor. Esto nos permite comunicarnos con cualquier arquitectura SOA que tenemos ya existente en nuestra organización.

Todo esto se debe a que SOFEA se estructura siguiendo un patrón de diseño muy extendido llamado MVC (Model-View-Controller) o Modelo-Vista-Controlador.

El Controlador se encarga de comunicar con diferentes servidores o servicios (SOAP,REST,etc.) utilizando comunicación asíncrona, soportando formato de datos XML o JSON. Esto nos permite integrar nuestra aplicación en una arquitectura SOA muy fácilmente.

El Modelo contiene un estado local de los datos con los que trabaja el usuario y la Vista se encarga de presentar y visualizar estos datos. En el siguiente diagrama se muestra todo el sistema unido.



Arquitectura de una aplicación siguiendo la metodología SOFEA

El modo de funcionamiento general de una arquitectura SOFEA seria :

1. El cliente accede a una aplicación web.
2. Un servidor de descarga envía la aplicación al cliente. Este servidor de descarga ya no vuelve a usarse más y su única misión es distribuir la aplicación web a los clientes que acceden por primera vez.

3. La aplicación web está construida siguiendo un patrón MVC. Por lo tanto el usuario interactúa con las Vistas, los Modelos guardan la información del estado de la aplicación y los Controladores interactúan con Servicios web.

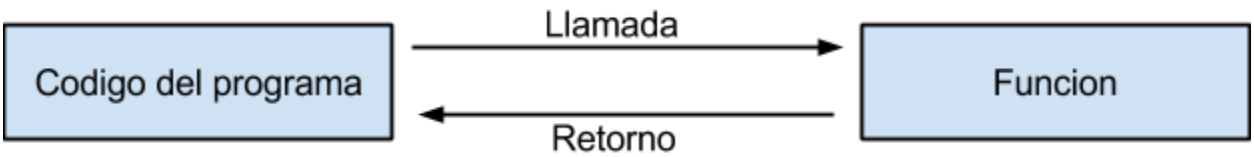
4. Cuando el usuario interactúa con la aplicación web, esta se comunica con servidores web remotos para recibir o enviar datos.

2.3 Comunicación Asíncrona

La comunicación asincrónica es aquella comunicación que se establece entre sistemas de manera diferida en el tiempo, es decir, cuando no existe coincidencia temporal. Hoy en día es una característica muy importante en los sistemas distribuidos ya que permite la ejecución de trabajos en paralelo sin dependencia temporal.

2.3.1 Eventos y Callbacks

Tradicionalmente los lenguajes de programación incluyen una primitiva llamada función (o método en un lenguaje orientado a objetos). Cuando una función es llamada se le pasa el control de ejecución del programa. La función lleva a cabo el cálculo y retorna un valor o en algunos casos no devuelve nada, simplemente devuelve el control de la ejecución a la parte del código que le ha llamado.

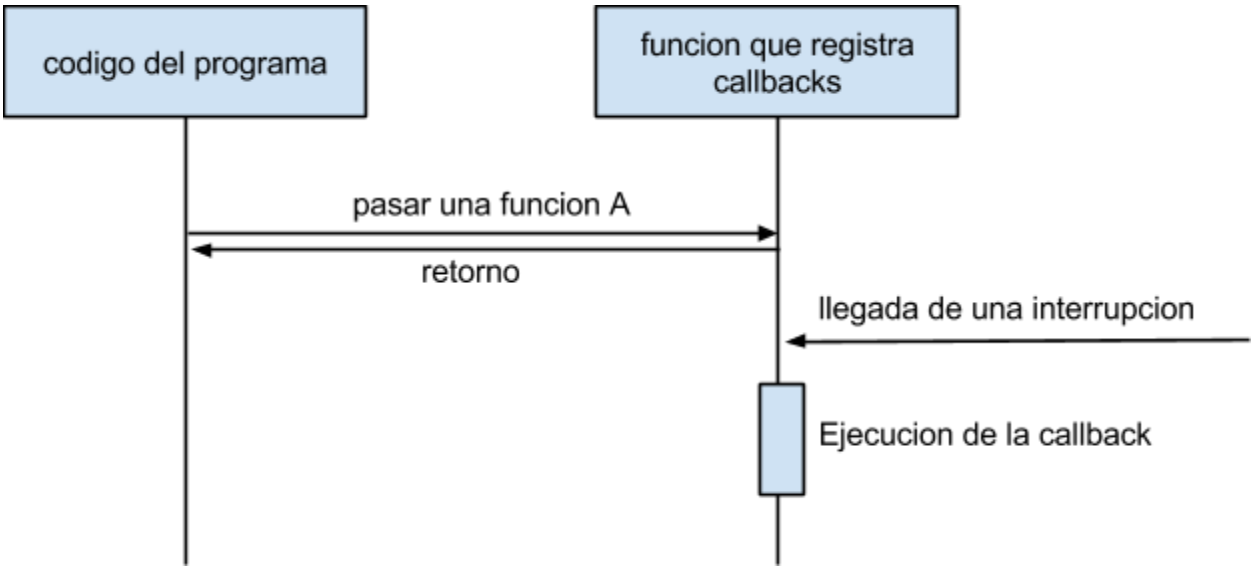


Llamada a una función síncrona normal

Sin embargo cuando se trata de una función o un comportamiento que queremos que se ejecuta al darse un evento en particular, no podemos utilizar el método clásico de la llamada a función.
En estos casos utilizamos un callback, que no es más que una función normal que no se llama inmediatamente sino que se registra en el sistema y se llama solamente cuando ocurre dicho evento. El mecanismo de registro en el sistema normalmente ocurre mediante el paso de una función como parámetro de otra función, que mantiene una lista con todos los eventos y sus respectivas funciones que tienen que ejecutarse al darse dichos eventos.

Esta función que registra nuestra callback es avisada mediante una interrupción de que un

evento ha llegado y la función callback asociada a este evento se ejecuta. De este modo se crea una comunicación asíncrona, ya que el código del programa no tiene que estar esperando que la función se ejecute, sino que tan solo la registra como un callback.



El registro y posterior llamada a una función Callback

La utilización de callbacks tiene varios inconvenientes, uno de los principales es el llamado Callback Hell. El Callback Hell consiste en que podemos hacer un anidamiento de callbacks, consiguiendo un código muy difícil de depurar y muy difícil de razonar.

Podemos evitar el Callback Hell si seguimos unas buenas prácticas de programación y diseño. Un ejemplo de eso sería no utilizar funciones anónimas, sino que siempre nombrar las funciones y asignarles variables, así como tampoco utilizar muchos niveles de anidamiento.

2.3.2 Promesas

En un mundo asíncrono, no se puede devolver valores en el instante que lo pedimos: simplemente no están listos a tiempo. Del mismo modo, no se puede lanzar excepciones, porque nadie está ahí para su captura. Así que volvemos al llamado "Callback Hell", en donde la composición de los valores de retorno implica callbacks anidadas, y la composición de errores consiste en pasarlos por la cadena de forma manual, sin la posibilidad de utilizar excepciones.

Las promesas son una abstracción de software que hace que el trabajo con operaciones asincrónicas sea mucho más agradable y fácil. En la definición más básica, el código pasara del estilo de programación con callbacks a uno donde sus funciones devuelven un valor, llamado una promesa, que representa a los eventuales resultados futuros de esa operación.

Sin embargo las promesas no son acerca de la agregación de callbacks. Las promesas

proporcionan una correspondencia directa entre las funciones sincrónicas y asincrónicas del código. Esto se puede apreciar en las dos características claves que hacen de las promesas unas herramientas muy valiosas :

1. Devuelven un resultado, como si fuese una función normal.
2. Lanzan excepciones al fallar y no poder ejecutarse correctamente.

Ambas son esenciales para la composición. Es decir, se puede pasar el valor de retorno de una promesa a otra promesa u otra función normal, y seguir esto indefinidamente. Más importante aún, si en algún momento ese proceso falla, una de las funciones en la cadena de composición puede lanzar una excepción, que luego pasa por todas las capas de composición hasta llegar a las manos de alguien que puede manejar la situación con una captura de esa excepción.

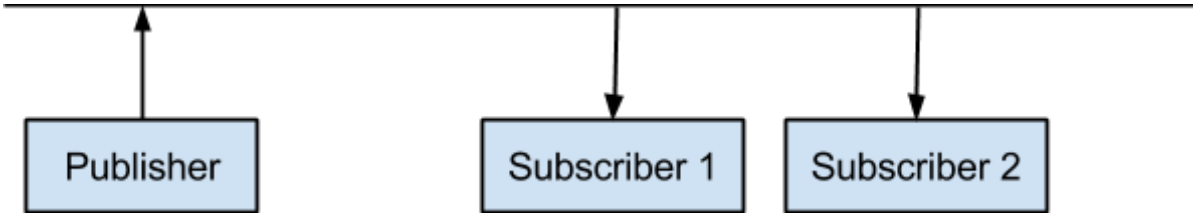
Aquí se especifica un código en JavaScript en el que es muy fácil de ver como se especifican las promesas y como se pueden utilizar para anidar operaciones.

```
getTweetsFor("Bogdan") // Función que devuelve una promesa
.  then(function (tweets) {
    var shortUrls = parseTweetsForUrls(tweets);
    var mostRecentShortUrl = shortUrls[0];
    return expandUrlUsingTwitterApi(mostRecentShortUrl); // Función que devuelve otra promesa
  })
.  then(httpGet) // Función que devuelve otra promesa más
.  then(
    function (responseBody) {
      console.log("El texto del enlace más reciente:", responseBody);
    },
    function (error) {
      console.error("Ha ocurrido un error con twitter:", error);
    }
  );
```

2.3.3 Patrón Publicador-Suscriptor (Bus de servicios)

El patrón de publicación-suscripción es un patrón de comunicación asincrónica donde los remitentes de los mensajes, llamados en inglés publishers, no envían los mensajes directamente a receptores específicos, llamados en inglés subscribers. En su lugar, los mensajes se clasifican en clases, y se publican por los publishers. Del mismo modo, los subscribers expresan interés en una o más clases, y sólo reciben mensajes que son de su interés. De ese modo se desacoplan los publishers y los subscribers, permitiendo hacer una difusión de mensajes muy eficientemente. Hay algunos sistemas que permiten la persistencia de los

mensajes, aumentando aún más las posibilidades y combinaciones que se pueden hacer con este patrón.



Patrón Publicador-Suscriptor de forma visual

Las principales ventajas que ofrece un sistema que sigue este patrón son las siguientes :

1. **Acoplamiento débil** : Los publishers están débilmente acoplados a los suscribers, y ni siquiera es necesario que sepan de su existencia. El concepto fundamental en la comunicación en ese patrón es la clase de los mensajes y de ese modo tanto los subscribers como los publishers pueden ignorar la topología del sistema. Por otro lado en el paradigma tradicional, cliente-servidor estrechamente acoplado, el cliente no puede enviar mensajes al servidor, mientras el proceso del servidor no está en ejecución, ni puede el servidor recibir mensajes a menos que el cliente se está ejecutando. Muchos sistemas publicador / suscriptor desacoplan no sólo las ubicaciones de los publishers y subscribers, sino también los desacoplan temporalmente.

2. **Escalabilidad** : El patrón de Publicador-Suscriptor proporciona una mejor escalabilidad que el tradicional cliente-servidor, a través de las operaciones en paralelo, caché de mensajes, el enrutamiento basado en red o basada en árbol, etc. Un ejemplo de escalabilidad puede ser una arquitectura donde tenemos un publishers que publica trabajos y un conjunto de subscribers consume los trabajos conforme van llegando y los procesas, creando una estructura conocida en computación paralela como productor - consumidor.

2.3.4 Mensaje Estructurado

Define el proceso de determinar el contenido del mensaje (lo que constituye un mensaje) y la estructura (cómo está organizado el contenido del mensaje), sin el cual las aplicaciones no tienen la base para comunicarse.

En una organización pueden existir muchos tipos diferentes de software y cada uno utilizando su propia estructura de datos para la comunicación. Muchas organizaciones funcionan a base de la arquitectura SOA, donde el intercambio de mensajes es la principal herramienta de comunicación.

Por todo esto es muy importante disponer de un sistema muy flexible que puede aceptar y tratar con una gran variedad de mensajes. Hoy en día, la gran mayoría de software utiliza los siguientes dos tipos lenguajes para estructurar sus mensajes.

XML (eXtensible Markup Language)

XML es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible.

Una de las principales ventajas que tiene XML es que al ser un estándar es ampliamente soportado por muchos sistemas informáticos.

Por otro lado algunas de las críticas para el uso de XML son su complejidad y gran nivel de detalle. Por ejemplo el paso de un documento XML a un sistema de tipos de cualquier lenguaje de programación puede ser difícil, especialmente cuando se utiliza XML para el intercambio de datos altamente estructurados entre aplicaciones, lo que no era su objetivo primario de diseño. Un ejemplo en XML sería el siguiente :

```
<menu id="fichero" value="Fichero">
  <popup>
    <menuitem value="Nuevo" onclick="CrearNuevoDoc()" />
    <menuitem value="Abrir" onclick="AbrirDoc()" />
    <menuitem value="Cerrar" onclick="CerrarDoc()" />
  </popup>
</menu>
```

JSON (JavaScript Object Notation)

Como alternativa a la complejidad de XML surge otro formato para estructurar un mensaje llamado JSON. Este formato es mucho más ligero que el lenguaje XML y tiene una sintaxis mucho más simple.

Todo esto hace que para el mismo tipo de información JSON ocupa mucho menos espacio que su alternativa XML. Por eso muchas grandes compañías, donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia, prefieren utilizar JSON como estructura para sus mensajes. El ejemplo anterior expresado en JSON seria :

```
{"menu": {
   "id": "fichero",
   "value": "Fichero",
   "popup": {
     "menuitem": [
       {"value": "Nuevo", "onclick": "CrearNuevoDoc()"},
       {"value": "Abrir", "onclick": "AbrirDoc()"},
```

```
      {"value": "Cerrar", "onclick": "CerrarDoc()"}
    ]
  }
 }
}
```

2.4 Object Oriented Programming (OOP) y JavaScript

El framework que estoy describiendo en este PFC es un framework para el desarrollo de aplicaciones web. Esto ha dictado la decisión de elegir JavaScript como lenguaje en el que hacer la implementación de Cuore.js, ya que es el único lenguaje de scripting soportado nativamente en todos los navegadores.

Por otro lado OOP pretende promover una mayor flexibilidad y facilidad de mantenimiento en la programación y es muy popular en la ingeniería de software a gran escala. Gracias a su fuerte énfasis en la modularidad, el código orientado a objetos está concebido para ser más fácil de desarrollar y más fácil de entender posteriormente.

Sin embargo JavaScript no tiene un soporte completo de una programación orientada a objetos (OOP), sino que dispone de una herencia prototípica con la que se pueden emular muchas de las características de los lenguajes OOP tradicionales. Esto se consigue mediante varios patrones de diseño, como por ejemplo Module Pattern o Constructor/Prototype Pattern, se pueden llegar a emular otras características de los lenguajes OOP.

Como consecuencia no hay modelo estándar de prácticas OOP en JavaScript y cada uno puede implementar los diferentes patrones de una forma diferente. Por eso cada vez más se está intentando invirtiendo mucho esfuerzo en la construcción de un framework estándar que unifique las diferentes técnicas y homogenice el código JavaScript que utiliza objetos.

2.4.1 Modelo Prototípico de JavaScript

En este apartado se discuten las características OOP de JavaScript.

Clase

JavaScript es un lenguaje basado en prototipos que no contiene ninguna declaración de clase, tal como se encuentra en C++ o Java. Esto es a veces confuso para los programadores acostumbrados a lenguajes con una declaración de clase. En lugar de ello, JavaScript utiliza funciones como clases. La definición de una clase es tan fácil como la definición de una función. En el siguiente ejemplo se define una nueva clase llamada Persona.

```
function Persona() { }
```

Objeto (Instancia de una clase)

Para crear una nueva instancia de un objeto “obj” usamos la declaración “new obj”, asignando el resultado (que es de tipo obj) a una variable para acceder a ella más tarde. En el siguiente ejemplo se define una clase llamada Persona y creamos dos instancias.

```
function Persona() { }
var persona1 = new Persona();
var persona2 = new Persona();
```

El Constructor

El constructor se llama en el momento de la creación de instancias (el momento en que se crea la instancia del objeto). El constructor es un método de la clase. En JavaScript, la función sirve como el constructor del objeto; por lo tanto, no hay necesidad de definir explícitamente un método constructor. Cada acción declarada en la clase es ejecutada en el momento de la creación de instancias.

El constructor se utiliza para establecer las propiedades del objeto o para llamar a métodos que preparan el objeto para su uso. La adición de métodos a la clase y sus definiciones se produce utilizando una sintaxis diferente se describe más adelante.

En el siguiente ejemplo, el constructor de la clase Persona, que muestra una alerta cuando se crea una instancia de una persona.

```
function Persona() {
  alert('Persona instanciada');
}

var persona1 = new Persona();
var persona2 = new Persona();
```

Métodos

En JavaScript, los métodos a su vez son función normales que están vinculados a una clase / objeto mediante una propiedad. Esto implica que los métodos son objetos también, ya que las funciones son constructores de objetos. Por todo esto los métodos pueden ser invocados "fuera de contexto".

```
function Persona(genero) {
  this.genero = genero;
  alert('Persona instanciada');
}

Persona.prototype.genero = '';
```

```
var persona1 = new Persona('Hombre');
var persona2 = new Persona('Mujer');

//muestra el género de una persona
alert('persona1 es ' + persona1.genero); // persona1 es Hombre
```

Este ejemplo demuestra muchos conceptos a la vez. Por un lado demuestra que no hay métodos privados en un solo objeto en JavaScript porque todas las referencias del método señalan a la misma función, la que hemos definido en el primer lugar en el prototipo. JavaScript "une", el "contexto del objeto" actual con la variable especial "this" cuando se invoca una función como un método (o propiedad para ser exactos) de un objeto.

Es muy importante señalar que solamente los métodos que forman parte de la propiedad “prototype” serán heredados en las instancias de una clase, es decir, si “genero” fuese una propiedad directa de la clase Persona, no se hereda a las instancias persona1 y persona2.

Herencia

La herencia es una forma de crear una clase como una versión especializada de una o más clases (JavaScript sólo admite la herencia de clases individuales, no múltiple). La clase especializada que comúnmente se llama hijo, y la otra clase es comúnmente llamada el padre. En JavaScript hace esto mediante la asignación de una instancia de la clase padre a la clase hija, y luego se especializa. En los navegadores modernos también se puede utilizar Object.create para implementar herencia. En el siguiente ejemplo, se define la clase Student como una clase hija de la clase Persona. Entonces volvamos a definir el método decirHola() y añadimos el método decirAdios().

```
// define a la clase Persona
function Persona() {}

Persona.prototype.caminar = function(){
  alert ('Estoy caminando!');
};
Persona.prototype.decirHola = function(){
  alert ('Hola');
};

// define la clase Estudiante Student
function Estudiante() {
  // Llama al constructor del padre
  Persona.call(this);
};
```

```
// heredar de Persona
Estudiante.prototype = new Persona();

// corregir el puntero del constructor, porque ahora apunta a Persona
Estudiante.prototype.constructor = Estudiante;

// reemplaza el método para decirHola de Persona
Estudiante.prototype.decirHola = function(){
  alert('Hola, soy estudiante');
};

// método para decir adiós
Estudiante.prototype.decirAdios = function(){
  alert('Adiós');
};

var estudiante1 = new Estudiante();
estudiante1.sayHello();
estudiante1.walk();
estudiante1.sayGoodBye();

// comprobar herencia
alert(estudiante1 instanceof Persona); // true
alert(estudiante1 instanceof Estudiante); // true
```

3 Tecnologías Existentes

Antes de lanzarse y ponerse a trabajar sobre un nuevo software, es muy importante comprobar que no existe alguna herramienta que ya satisfaga los requisitos que nosotros queramos. En nuestro caso estábamos interesados en algún framework que cumpla todas las premisas explicadas anteriormente. Un resumen de estas premisas sería :

1. Seguir la metodología SOFEA.
2. Comunicación asíncrona, mediante diferentes métodos de comunicaciones : promesas, bus de servicios (publisher / subscript) y callbacks.
3. La utilización de mensajes estructurados, respetando el formato de datos intercambiados.
4. Utilización de técnicas OOP en el lenguaje JavaScript.

Después de tener claro las premisas que debe cumplir el framework, se han estudiado varios frameworks y se ha contemplado que características tienen y cuales les faltaban para llegar a reemplazar la necesidad de crear Cuore.js.

3.1 JQuery

JQuery no es un framework per se, sino que es una librería JavaScript. Es uno de los proyectos JavaScript más extensamente utilizado y con un impacto muy grande en el desarrollo de aplicaciones web. Es Open Source, con licencia GPL y MIT.

Algunas de sus funcionalidades más importantes incluyen interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, animaciones, simplificación en la utilización AJAX y api homogéneo en diferentes navegadores.

Al tratarse de una librería y no de un framework es entendible que no ofrece una metodología SOFEA para la construcción de aplicaciones web. Sin embargo muchas de las herramientas que ofrece podrían ser utilizadas para la base de un framework que sigue SOFEA.

JQuery proporciona mecanismos de comunicación asíncrona como las callbacks. Sin embargo si la complejidad de la aplicación aumenta se puede sufrir el llamado efecto Callback Hell, por lo que su utilización tiene que ser muy cuidadosa. También se ofrece una especia de Promesas,

llamadas Deferred, que si bien no llegan a ser lo mismo que las Promesas clásicas ofrecen una API y comportamientos similares.

Otra característica importante que incluye JQuery son las utilidades para comunicación AJAX mediante callbacks. Sin embargo no ofrece ninguna herramienta para mensaje estructurado, así que la serialización de los datos tiene que hacerse manualmente o con otra herramienta diferente.

En cuanto a técnicas OOP, JQuery tampoco dispone de ninguna facilidad. Si se quiere seguir la metodología OOP habría que hacerlo manualmente o integrando JQuery con algún sistema de objetos de terceros.

Por todo esto JQuery no es una buena opción ya que aunque sea muy extendido y muy ligero, es demasiado simple y le faltan muchas de las características comentadas en los apartados anteriores.

3.2 Backbone

Backbone.js es uno de los frameworks más extendidos de JavaScript para la escritura de single-page application (SPA).

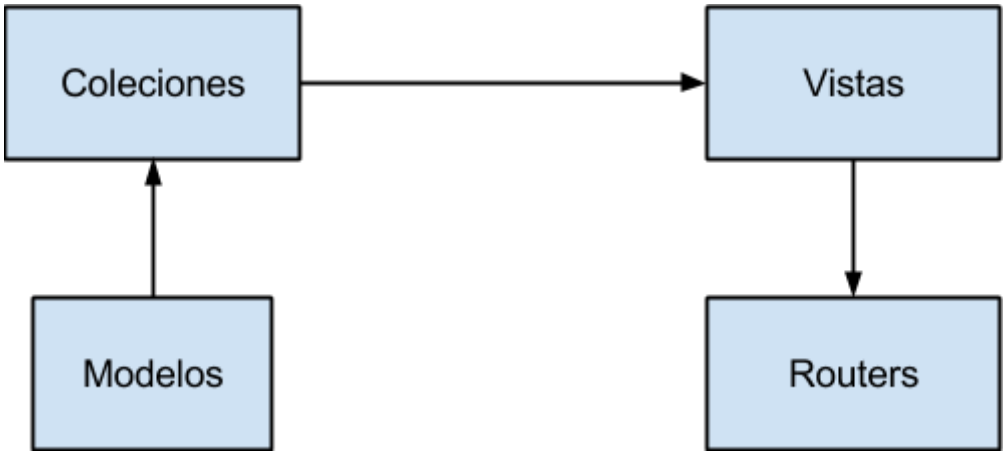
La arquitectura del framework está formada por cuatro conceptos claves :

1. **Modelo** : Los Modelos son el corazón de cualquier aplicación Backbone.js. Contienen los datos interactivos, así como una gran parte de la lógica que los rodea: conversiones, validaciones, propiedades calculadas y control de acceso. Se extiende Backbone. Model con tus métodos específicos de dominio y los modelos proporcionan un conjunto básico de funcionalidades para la gestión de cambios, así como su almacenamiento en un servidor remoto o almacenamiento local mediante HTML5.

2. **Colección** : Las Colecciones son conjuntos ordenados de modelos. Puede enlazar los eventos de "cambio" para recibir una notificación cuando cualquier modelo de la colección se ha modificado, la escucha de los eventos "añadir" y "eliminar", ir a buscar una colección desde el servidor, etc.

3. **Vista** : Las Vistas en Backbone.js son más convenciones que código - no determinan nada de tu HTML o CSS por ti, y se pueden utilizar con cualquier biblioteca de plantillas de JavaScript. La idea general consiste en organizar la interfaz en vistas lógicas, respaldados por los modelos, cada uno de los cuales se pueden actualizar de forma independiente cuando el modelo cambia, sin tener que volver a dibujar la página completa.

4. **Router** : Los Routers proporcionan métodos para conectar las páginas del lado del cliente, y conectarlas a las acciones y eventos. Para los navegadores que aún no admiten la API de Historia (disponible en HTML5), el Router se encarga de traducir transparentemente la URL a la versión con fragmentos hash.



La estructura global de una aplicación construida con Backbone.js

Podemos argumentar que esta estructura sigue una metodología más o menos SOFEA.

Por otro lado tenemos que este framework también soporta una comunicación asíncrona con un servidor remoto. Esto se debe a la funcionalidad transparente de los modelos para mantenerse sincronizados con un backend. Sin embargo no tenemos mucha flexibilidad en esa comunicación y tampoco disponemos de las llamadas promesas. Por tanto la comunicación asíncrona con la que viene predefinido Backbone.js es muy simplista y sirve sobre todo para aplicaciones tipo CRUD (Create-Read-Update-Delete).

Backbone.js depende de una librería auxiliar llamada Underscore.js, denotada muchas veces simplemente como "_". Esta librería ofrece primitivas para el desarrollo con objetos, de las cuales Backbone.js se aprovecha y sigue las técnicas clásicas de OOP en su código, como uso de objetos, herencia y sobre escritura de métodos.

Aparte de estas funcionalidades OOP, Underscore.js también dispone de algunas funciones cuyo comportamiento se acerca a los lenguajes funcionales, como por ejemplo map y reduce.

Backbone.js fue una de los proyectos que más se acercaba a las premisas definidas en los apartados anteriores. Sin embargo, su simplicidad y el hecho de tener que escribir una gran capa sobre él, hizo empezar directamente un nuevo proyecto. De ese modo se podría empezar con una arquitectura que desde las raíces está preparada para las características que nos interesan.

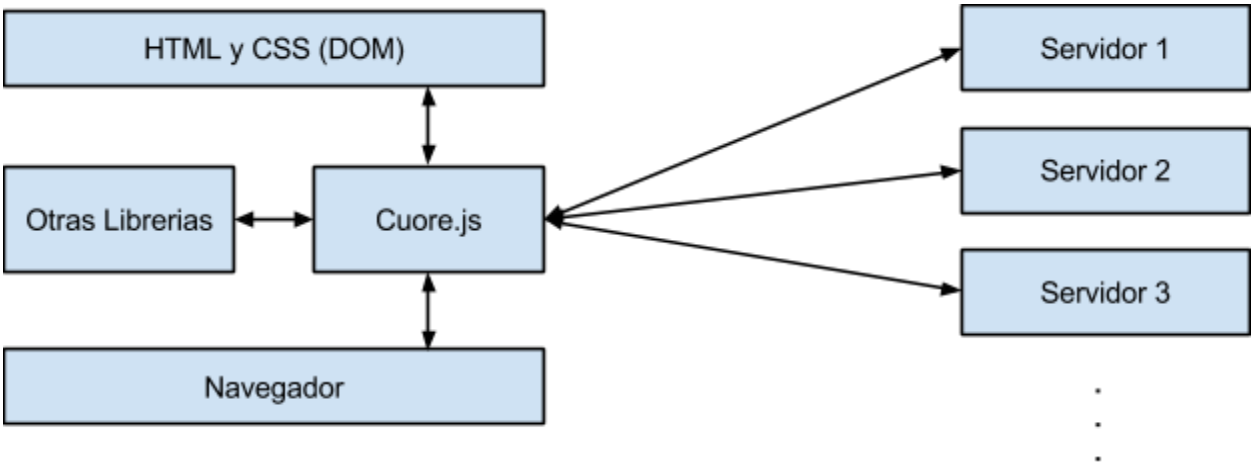
4 Arquitectura de Cuore.js

4.1 Visión de Conjunto

Al no poder encontrar una alternativa clara que cumpla todos los requisitos establecidos, se ha optado por desarrollar nuestro propio framework.

Para el desarrollo de Cuore.js hemos implementado todas las características que estábamos buscando, añadiendo también algunas buenas prácticas en la programación como DRY (Don't Repeat Yourself) y SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion).

La principal idea a la hora de diseñar Cuore.js es hacer una estructura que sea muy fácilmente extensible y muy flexible ante cambios en protocolos de comunicación o formas de presentación. Esto es debido a que el framework va a ser la principal vía de comunicación y lógica de una aplicación web.



Contexto en el que se ejecuta Cuore.js y como interactúa con las demás tecnologías

En el diagrama UML que se muestra abajo se aprecian todas las principales clases que forman el núcleo del framework. La descripción de estas clases es :

- **Clase_**: Cabe destacar que la mayoría de estas clases heredan de esta clase. Esto es así, porque JavaScript no tiene un sistema de clases propio y hemos optado por desarrollar nuestro sistema de clases. Ese sistema de clases es un sistema muy minimalista y simple pero sin embargo permite tener clases, herencia entre clases y sobre escritura de métodos de una manera muy parecida a los lenguajes OOP.

- **Page** : Es el punto de partida para la escritura de una aplicación web. Para empezar una

aplicación web con Cuore.js hay que heredar de esta clase.

- **Component** : Es una de las clases más importantes ya que se encarga de hacer la unión entre los mensajes enviados por el Bus y diferentes handlers que tiene asociado este componente. También se encarga de llamar a los diferentes servicios de los que depende este componente. Por último mantiene relación con el Renderer, que se encarga de visualizar el componente en pantalla. Normalmente representan partes gráficas que pueden formar parte de la aplicación, por ejemplo un botón, un formulario, un campo, etc., pero pueden representar a otros elementos invisibles en el código HTML. Es el Modelo en MVC.

- **Registry** : Representa un registro con todos los componentes de la aplicación. Es único por cada clase Page.

- **Handler** : Define la respuesta a un evento. Hay que destacar que un componente puede tener diferentes tipos de handlers, dependiendo de los eventos a los que queremos que responda, así como señalar que un evento puede responder a muchos tipos diferentes de eventos. Un handler puede pertenecer solamente a un componente. Es el Controller en MVC.

- **HandlerSet** : Es un registro de todos los handlers de un componente (los HandlerSets no se comparten entre componentes).

- **Bus** : Es un sistema clásico de bus encargado de registrar suscriptores y emitir mensajes, es decir, llamar a esos suscriptores pasándoles el mensaje que se ha emitido.

- **Service** : Representa a un servicio genérico. Puede ser compartido por varios Components. Normalmente se utiliza para hacer peticiones al servidor o realizar operaciones lógicas locales y emitir o no eventos, por ejemplo botón pulsado o cambio del texto de un label. Pueden ser llamados desde cualquier componente mediante la clase Directory.

- **Directory** : Es un directorio (listado) con todos los servicios disponibles en el sistema. Estos servicios pueden ser llamados desde varios Components. El directorio es global para toda la clase Page.

- **Renderer** : Esta clase es un render genérico, del que los renders concretos heredarán. Se encarga de “dibujar” sobre el DOM, es decir, crear elementos HTML y aplicarles diferentes clases CSS. Es el View en MVC.

- **Decoration** : Esta clase decora el output de la clase Renderer y sirve para modificar su comportamiento. Un Renderer puede tener una seria de Decorators que se enlazan

como una cadena.

- **State** : Representa el estado de una clase Page. Es un almacén global de datos que sirve para guardar datos en forma de nombre-valor. El valor puede representar a su vez otra estructura del tipo nombre-valor, permitiendo crear arboles de datos.

- **Message** : Esta clase representa al mensaje estructurado que se envía al exterior y que se recibe del exterior. Cuore.js tiene métodos auxiliares de comunicación que se encargan de convertir los datos a la clase Message de forma transparente al usuario.

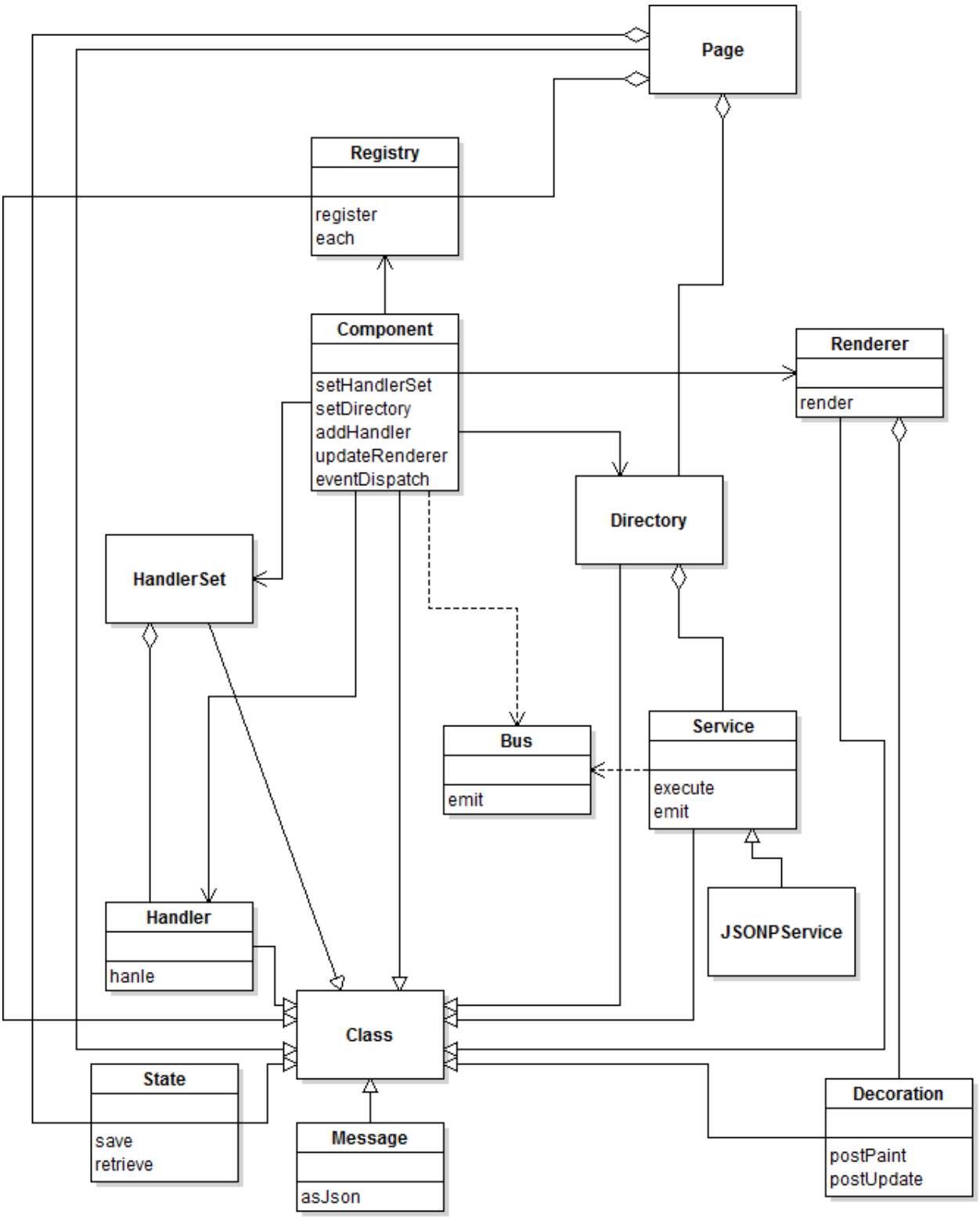


Diagrama UML de las clases más importantes de Cuore.js

4.2 Patrones de Diseño

En este apartado se discuten varios patrones de diseño utilizados en Cuore.js, que hacen su arquitectura muy flexible y adaptativa a cambios.

4.2.1 Publish–subscribe

Los publicadores y los suscriptores son aplicaciones que envían y reciben mensajes (publicaciones) utilizando el método de mensajería de publicación/suscripción. Los publicadores y los suscriptores están desasociados, de manera que los publicadores no conocen el destino de la información que envían y los suscriptores no conocen el origen de la información que reciben.

El proveedor de la información recibe el nombre de publicador. Los publicadores suministran información sobre un tópico sin necesidad de saber nada acerca de las aplicaciones que están interesadas en la información.

El consumidor de la información recibe el nombre de suscriptor. El suscriptor decide qué información le interesa y luego espera a recibir dicha información. Los suscriptores pueden recibir información de muchos publicadores diferentes, y la información que reciben también puede enviarse a otros suscriptores. Los suscriptores reciben solamente mensajes de los tópicos a los que se han suscrito.

En nuestro caso todos los Servicios serían publicadores ya que emiten mensajes en el Bus y los Componentes son los suscriptores ya que se suscriben al Bus para recibir mensajes.

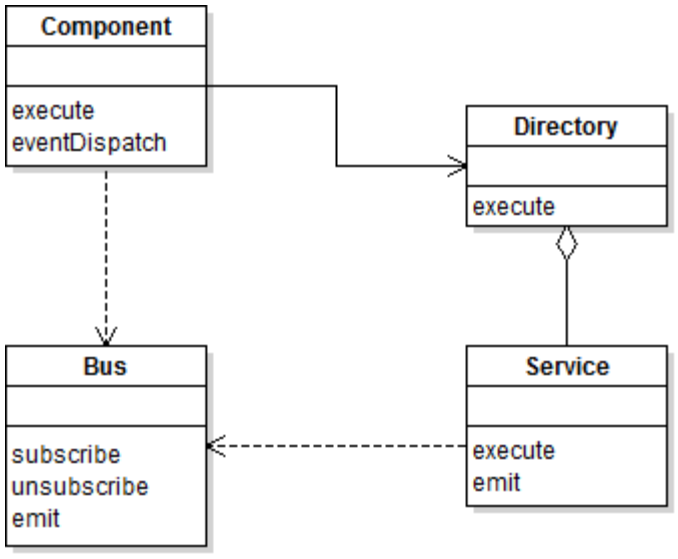


Diagrama UML del patrón pub/sub en Cuore.js

El flujo de información es el siguiente:

1. El componente se suscribe en el Bus para un evento con el método **subscribe**.
2. El componente ejecuta un servicio (mediante el directorio de servicios) con el método **execute**.
3. El Directorio de servicios buscan el servicio adecuado para la petición y lo ejecuta mediante **execute**.
4. El servicio se ejecuta y emite la respuesta al Bus mediante **emit.**
5. El Bus busca a todos los suscriptores del tópico del mensaje y llama a sus métodos **eventDispatch**. De esta manera vuelve al inicio, en la clase Component.

Este sistema proporciona una gran flexibilidad ya que los Servicios no tienen que saber nada acerca de los Componentes y pueden existir sin ningún componente. Por otro lado se desacopla la recepción de mensajes ya que más de un componente puede estar esperando para un evento o mensaje y de este modo la ejecución de varios componentes se hace transparente al usuario.

La utilización de este patrón de diseño mejor la arquitectura del framework en dos aspectos claves. Mejora el desacoplamiento entre componentes y servicios, haciéndolo prácticamente cero. También mejora la escalabilidad porque los componentes y servicios se podrían ejecutar en diferentes hilos, mediante el nuevo estándar Web Workers. Esto aprovecharía mucho mejor los nuevos CPU, que normalmente disponen de 2, 4 o 8 núcleos.

Una de las posibles mejoras es hacer que los mensajes se almacenan durante un cierto tiempo en el Bus por si queremos transmitir mensajes a componentes aún no inicializados o deshabilitados.

Sin embargo esto introduciría una elevada complejidad en nuestro sistema que no está suficientemente justificada ya que los casos en los que queremos almacenar mensajes para su posterior transmisión son muy escasos dado la natura de las aplicaciones web.

4.2.2 Command

Este patrón reemplaza a las callbacks en los lenguajes orientados a objetos. Command es un patrón de comportamiento en el que un objeto se utiliza para representar y encapsular toda la información necesaria para llamar a un método en un momento posterior. Esta información incluye el nombre del método, el objeto que posee el método y los valores de los parámetros del método. De ese modo se puede llamar a una operación de un objeto sin realmente conocer el contenido de esta operación ni el receptor real de la misma. Existen cuatro términos asociados con este patrón :

- **Command** : Este objeto se encarga de ejecutar la lógica del comando o comportamiento deseado. Recibe el objeto Receiver y ejecuta su método. En nuestro framework este objeto es la clase *Handler*.

- **Receiver** : Este objeto es el objeto que realiza realmente el trabajo. Contiene la acción que hay que ejecutar. La funcionalidad de este objeto también es implementado por la clase *Handler* de nuestro framework. Los motivos de esto se discuten más adelante en este apartado.

- **Invoker** : Se encarga de llamar al objeto Command, y adicionalmente puede guardar trazas de la ejecución, como una pila, para poder deshacer operaciones. En Cuore.js esta función se realiza por la clase *HandlerSet.*

- **Client** : El cliente contiene la lógica de decisión sobre qué comando ejecutar en que momento. Por eso tiene referencias tanto al objeto Invoker y a los objetos Command. En el framework que estamos discutiendo, esta función se realiza por la clase *Component*.

Este patrón es utilizado en muchos de los software que de uso diario. Algunos de los usos más extendidos de este patrón son :

- **Thread pools** : Una clase típica y genérica Thread Pool o piscina de hilos de ejecución,

puede tener un método público llamado addTask() que añade una unidad de trabajo a la cola interna de tareas. La clase Thread Pool mantiene un grupo de subprocesos que ejecutan las tareas de la cola. Los elementos de la cola son objetos Command. Normalmente estos objetos implementan una interfaz común, como java.lang. Runnable que permite al grupo de subprocesos ejecutar la tarea a pesar de que la propia clase del grupo de subprocesos fue escrita sin ningún conocimiento de las tareas específicas para las que se utilizaría.

- **Parallel Processing** : Cuando los comandos se escriben como tareas que se mandan a un recurso compartido y estos son ejecutados por muchos hilos en paralelo, posiblemente en máquinas remotas (esta variante se refiere a menudo como el patrón Master/Worker).

- **Barras de progreso** : Supongamos que un programa tiene una secuencia de tareas que se ejecutan en orden. Si cada tarea tiene un método obtenerDuracionEstimada(), el programa puede calcular fácilmente la duración total. Puede mostrar una barra de progreso que refleja de manera significativa lo cerca que el programa es de completar todas las tareas.

Como se puede ver en el diagrama de abajo, este patrón fue ligeramente modificado y adaptado a nuestras necesidades.

El mayor cambio está en la unión de las funcionalidades de los objetos Command y Receiver en una sola clase llamada Handler (muy parecido a como se implementaría un Thread Pool). Esto es así porque nuestro framework no requería la funcionalidad que se introduciría al tener dos objetos separados. Sin embargo esta separación de funciones implica un aumento en la complejidad del diseño del sistema que no está justificado.

El funcionamiento del nuestro sistema consiste en los siguientes pasos :

1. El **Component** registra un nuevo **Handler** en el **HandlerSet.** El **Component** también se registra en el **Bus,** con el nombre del evento que el **Handler** requiere.

2. Al llegarle un evento por el **Bus,** el **Component** notifica al **HandlerSet** y le pase el mensaje.

3. El **HandlerSet** pasa el mensaje al **Handler** o Handlers que están suscritos a este evento.

4. El **Handler** en concreto ejecuta su método handle para realizar la acción requerida.

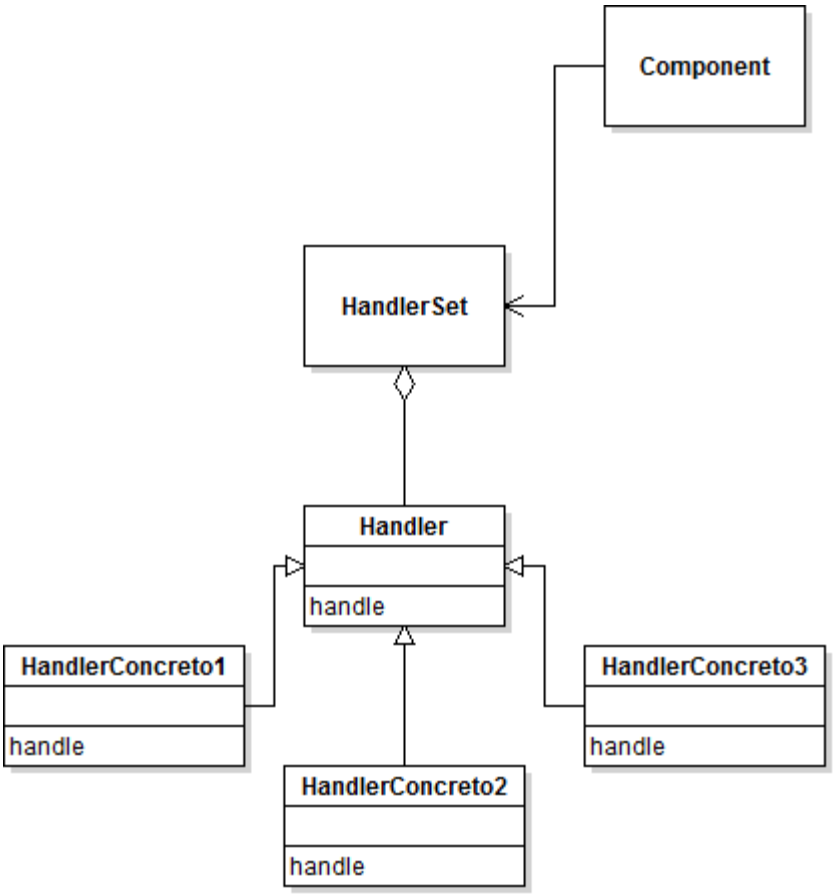


Diagrama UML del patrón Command implementado en Cuore.js

Cuore.js viene con un conjunto de Handlers predefinidos. De este conjunto el handler más interesante es el Executor. Al crearse el handler, se le asigna un callback o método del componente que le ha creado. A su vez cuando este handler es llamado, llama al método del componente "padre" que lo ha creado. De ese modo se consigue llamar a métodos del padre sin romper la estructura del patrón de diseño.

El uso de este mejora considerablemente la estructura del framework ya que se independiza la invocación los handlers y la implementación de los mismos. Por otra parte al tratar los handlers como objetos se puede realizar herencia de los mismos, o composiciones de handlers. También se facilita la aplicación en conjunto de muchas órdenes o handlers con una misma orden o evento.

4.2.3 Directory (First Class Collection)

Este patrón es muy simple y fácil de utilizar, pero añade mucha flexibilidad y simplicidad en la utilización de los servicios. Consiste en definir una clase que lista a todos los servicios disponibles. Esta clase es la principal interacción entre los componentes y los servicios. Su funcionamiento es el siguiente :

1. El componente llama a un servicio mediante su nombre y el nombre de su método o función que requiere mediante la función **execute.** Por ejemplo {nombre Servicio : ServicioHora, nombre método : horaActual}, esto especifica que queremos un servicio de hora y obtener la hora actual.

2. El Directorio se encarga de buscar el servicio adecuado basándose en el nombre del servicio que hemos pasado como parámetro y lo llama mediante la función **execute**.

3. El servicio ejecuta su método **execute.** Este método es el núcleo del servicio y contiene toda la lógica. En este método por ejemplo se puede desde establecer una comunicación entre el servicio y un servidor remoto hasta implementar un sistema de caché local. El servicio puede utilizar su método emit para emitir mensajes por el Bus.

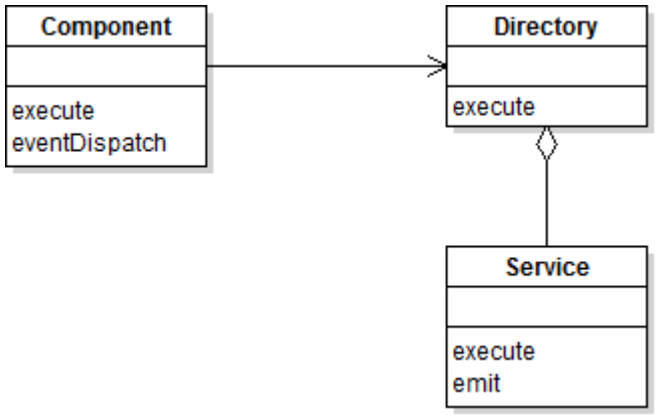
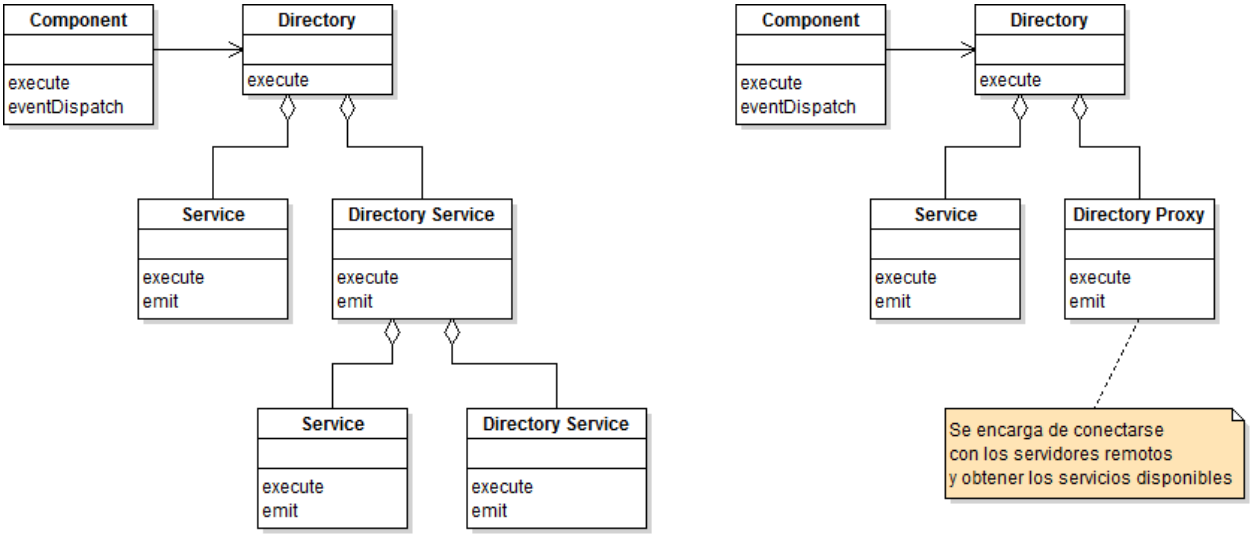


Diagrama Uml del patrón Directory en Cuore.js

La facilidad del uso del Directorio y el desacoplamiento que ofrece entre los componentes del y los servicios hace de él una parte muy importante del sistema. En la versión actual de Cuore.js el componente tiene que saber el nombre del servicio que quiere ejecutar, pero en futuras versiones se podría implementar un sistema de pattern matching inteligente en el que dado un nombre aproximado el directorio sea capaz de buscar un servicio adaptado al nombre. Otra de las posibilidades que ofrece este patrón de diseño es que en futuras extensiones del framework se podría implementar un comportamiento en el cual, si el Directorio no encuentra el servicio en local, pueda preguntar a servidores remotos que ofrezcan ese servicio.

Actualmente también se puede lograr un comportamiento igual que el descrito anteriormente, implementando un servicio especial llamado Servicio Remoto, cuya única misión es llamar a servicios que se encuentran en otros servidores y devolver su respuesta. Este comportamiento es muy parecido al patrón de diseño llamado Proxy. Esto permite combinar los dos patrones, Directory y Proxy, es decir un servicio que nos ofrezca un directorio de servicios remotos a los que podemos acceder. Este patrón es tan flexible y nos ofrece tantas posibilidades, que hasta nos permite crear una estructura de árbol para organizar de manera diferente los servicios ofrecidos por el sistema.



Diagramas UML con diferentes extensiones del patrón Directorio

4.2.4 Observer (Observador / Observado)

Este patrón es encontrado muy frecuentemente en frameworks y aplicaciones que siguen el patrón MVC (Model-View-Controller). En Java forma parte de la librería (java.util.Observer) y en C# forma parte del lenguaje (sintaxis “event”). Es uno de los patrones más extendidos y más utilizados en la actualidad.

El patrón Observador define una dependencia de uno a muchos entre objetos, para que cuando uno de ellos cambie de estado, todos los que dependan de él sean avisados y se actualicen automáticamente.

El patrón se compone principalmente por dos componentes :

- Sujeto (**Subject**) : Este objeto mantiene una lista con los Observadores y se encarga de seleccionar y avisar a los Observadores concretos ante algún cambio en el objeto que está siendo observado. Este objeto también se encarga de proporcionar una API pública

para modificar esa lista. Esto permite que código de fuera pueda controlar quien recibirá los avisos ante cambios en el estado del objeto observado. Muchas veces se implementa como un objeto aparte del objeto observado, pero puede formar parte de él también.

- Observador (**Observer / Listener**) : Esto representa una interfaz que todos los observadores concretos deben implementar. Los propios objetos observadores se encargan de llevar a cabo acciones ante determinados cambios de estado en el objeto observado.

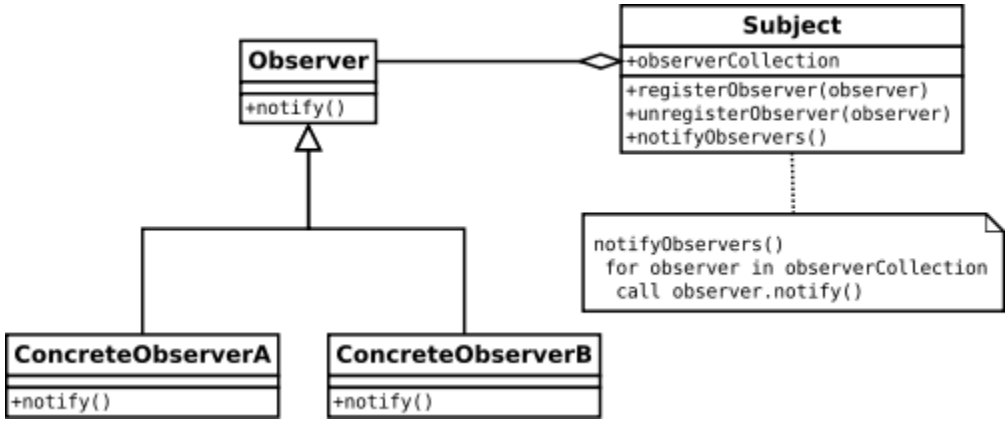


Diagrama UML del patrón Observer

En nuestro framework podemos encontrar este patrón en la parte del componente que trata con el rendering de un componente. El Component es nuestro objeto Subject, que se encarga de avisar al Render ante cambios de estado. Por otro lado el Renderer es nuestro objeto Observer, que se encarga de observar y actuar ante actualizaciones en el componente.

Diagrama UML de la aplicación del patrón Observer en Cuore.js

Esto representa un caso clásico donde se sigue muy estrechamente su aplicación en el patrón MVC, donde el Modelo es el objeto Observado y el Vista es el objeto Observador.

El comportamiento de sistema de rendering en Cuore.js es el siguiente:

1. **Component** crea un nuevo **Renderer**. Después puede llamar a su método draw cuando quiere dibujar algo sobre la pantalla. Este método acepta al propio objeto Component como su argumento.

2. **Renderer** ejecuta su método render, que dependiendo del comportamiento del **Component** pasado como argumento, reemplaza lo que hay dibujado en el lugar del **Component** o añade alguna cosa más.

3. Este paso es opcional y ocurre solo si el **Renderer** tiene registrados **Decorators**, que son clases que dibujan una vez que el **Renderer** ha acabado su trabajo.

Todo esto es muy importante ya que ofrece un gran desacoplamiento y reúso de los Renders. Por ejemplo nos permite reutilizar un renderer para dos componentes que tienen la misma apariencia, pero con lógica diferente. Otro ejemplo es que este sistema nos permite crear diferentes renders para un componente. De ese modo dependiendo del contexto en el que se encuentra dicho componente puede utilizar el render que más le convenga. Un ejemplo de esto último sería un componente persona. Este componente se podría dibujar como un texto si forma parte de una lista o como una imagen si forma parte de una galería.

4.2.5 Decorator

El patrón Decorator responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos evita tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

Una metáfora muy buena para entender este concepto sería : *"envolver un regalo, ponerlo en una caja, y envolver la caja"*. En esa metáfora tenemos tres decoradores del regalo, un envoltorio, una caja y otro envoltorio.

La principal idea del decorador es extender las funciones y características del objeto base o decorado. Sin embargo esto no suele hacerse con una cadena de herencia ya que esto conlleva muchos problemas, como por ejemplo muchos lenguajes no soportan la herencia múltiple. Uno de los métodos que se utilizan para la implementación es la composición, mediante el cual el objeto decorado, mantiene una lista con todos los objetos decoradores y los va llamando seguidamente para que lo decoren.

Otro método con el que se puede implementar la decoración es más dinámico y se implementa como una cadena de llamadas a diferentes decoradores, pasando al último decorador el objeto base.

Las principales partes que componen este patrón son :

- **Component** : Objetos que pueden tener responsabilidades añadidas.
- **Decorator** : Añade responsabilidades al componente.

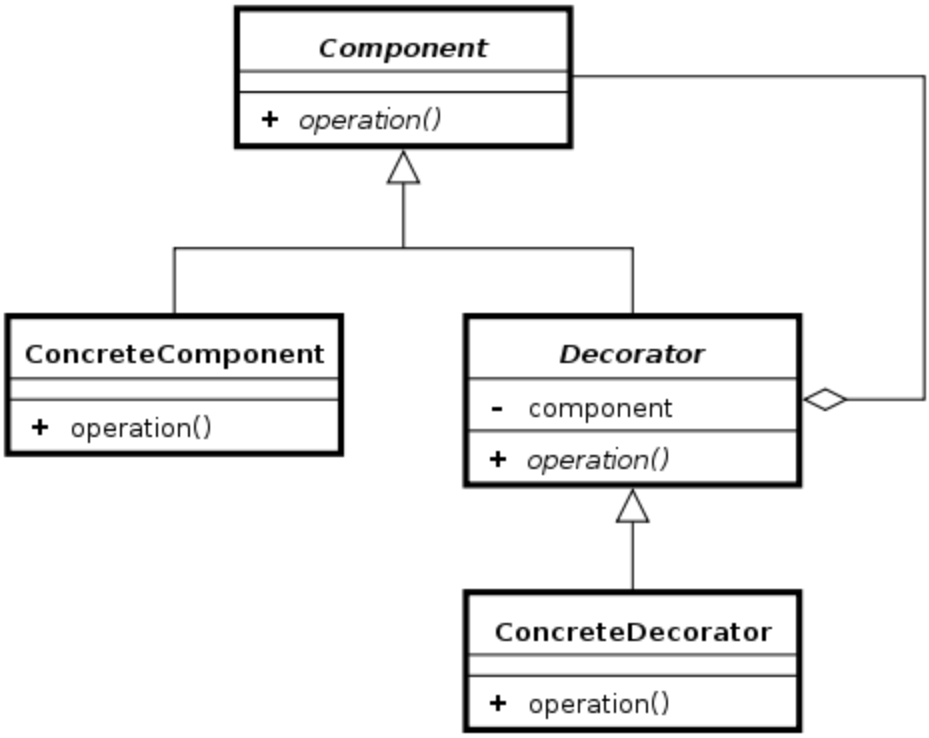


Diagrama UML del patrón Decorator

Algunos de los casos en los que es deseable modelar un sistema con este patrón son :

- Añadir de forma dinámica y transparente objetos que cambien o modifiquen el comportamiento de un objeto.
- Cuando algunas responsabilidades de un objeto pueden ser sacadas del mismo y puestas en objetos decoradores.
- Cuando la herencia múltiple no está disponible en el lenguaje que estemos usando.
- Existe la posibilidad de reutilizar un decorador para extender más de un objeto base.
- Es interesante crear largas y flexibles extensiones de objetos.

En Cuore.js este patrón se utiliza extensamente en el sistema de rendering para poder decorar a los Renders. Para llevar a cabo esta decoración, todo render contiene una lista con todos los decoradores que se han registrado en él.

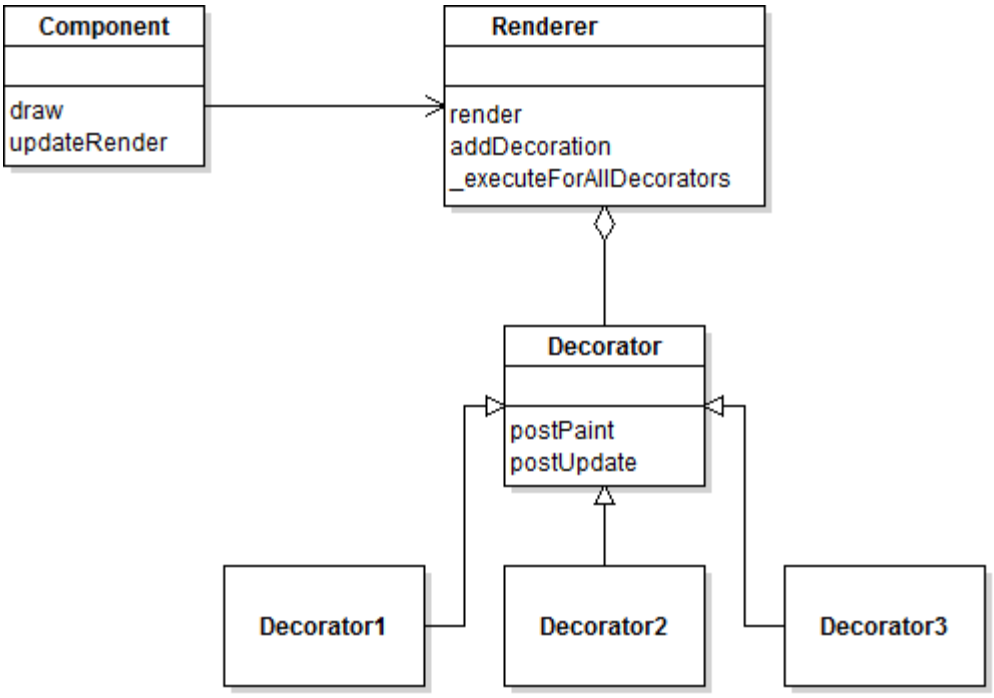


Diagrama UML de la implementación del patrón Decorator en Cuore.js

El funcionamiento es el siguiente :

1. El **Renderer** registra a los **Decorators** mediante el método “addDecoration”. Esto implica que el **Decorator** en concreto se añade a la lista interna que mantiene cada **Renderer**.

2. Después de terminar la ejecución del método render de la clase **Renderer**, se llama a todos los **Decorators** secuencialmente, es decir, uno detrás de otro.

3. Cada **Decorator** ejecuta su código.

Al utilizar este patrón de diseño en el sistema de rendering de Cuore.js se ha permitido tener más flexibilidad que el simple uso de la herencia. Los ejemplos de esto son muchos, como añadir y eliminar decoradores de componentes en tiempo de ejecución, herencia múltiple en los Renders, etc. La flexibilidad se mejora también por la gran cantidad de decoradores pequeños que se pueden reutilizar para varios renders. Por último, las clases Render son más sencillas y delegan parte de la responsabilidad a los decoradores, lo cual contribuye a que estas clases no sean muy complejas.

5 Desarrollo guiado por pruebas (TDD)

Las pruebas se han realizado conforme se ha ido haciendo la implementación y no después, como es común en la metodología de desarrollo de software en cascada.

Se ha empleado una metodología ágil y más en concreto la metodología TDD (Test-Driven Development). Por eso las etapas de Implementación y Prueba se han fusionado en una y se han desarrollado conjuntamente.

El desarrollo basado en pruebas (TDD) es un proceso de desarrollo de software que se basa en la repetición de un ciclo de desarrollo muy corto:

1. El desarrollador escribe un caso de prueba automatizada que define una mejora deseada o una nueva función del código. Como el código que satisfaga ese caso de prueba aún no está implementado, este caso de prueba dará error.

2. A continuación, se produce la cantidad mínima de código para pasar esa prueba.

3. Finalmente se refactorizar el código nuevo escrito para pasar la prueba a los estándares aceptables del proyecto.

Kent Beck, a quien se atribuye haber desarrollado o 'redescubierto' la técnica, declaró en 2003 que TDD fomenta diseños de programas simples e inspira confianza en el código producido.

El ciclo de desarrollo basado en pruebas, que se presenta a continuación está basado en el libro “Test-Driven Development by Example” escrito por Kent Beck.

1. **Agregar una prueba** : En desarrollo basado en pruebas, cada nueva función comienza con la escritura de una prueba o test. Esta prueba debe fallar porque está escrita antes de que la función se ha implementado. Si no falla, entonces o bien la "nueva" característica propuesto ya existe o la prueba es defectuosa. Para escribir una prueba, el desarrollador debe entender claramente las especificaciones y requisitos de las mismas. El desarrollador puede lograr esto a través de casos de uso para cubrir los requisitos y condiciones de excepción. Esta es una característica diferenciadora del desarrollo basado en pruebas frente a escribir las pruebas unitarias después de escribir el código: hace que el desarrollador se centre en los requisitos antes de escribir el código, una diferencia sutil pero importante.

2. **Ejecutar todas las pruebas** : Al ejecutar todas las pruebas, hay que ver si la nueva prueba añadida anteriormente falla. Esto valida que la batería de pruebas está funcionando correctamente y que la nueva prueba no pasa por error, sin la necesidad

de ningún código nuevo. En el caso normal la nueva prueba falla por la razón esperada. Esto aumenta la confianza (aunque no garantiza) que la prueba está probando lo que debe, y sólo pasa en los casos previstos.

3. **Escribir algo de código :** El siguiente paso es escribir el código que hace que la prueba anterior pase. El nuevo código escrito en esta etapa no es perfecto, y puede, por ejemplo, pasar la prueba de una manera poco elegante. Eso es aceptable, ya que los pasos posteriores se dedicaran a mejorar y perfeccionar este código. En este punto, el único propósito del código escrito es pasar la prueba.

4. **Volver a ejecutar todas las pruebas :** Si ahora todos los casos de prueba pasan, el programador puede estar seguro de que el código cumple con todos los requisitos que se están probando. Este es un buen punto de partida para comenzar el paso final del ciclo.

5. **Refactorizar el nuevo código** : Una vez que tenemos el nuevo código implementado y con la funcionalidad deseada y validada con las pruebas, es hora de refactorizar y mejorar este código. Mover el código de donde era conveniente para pasar la prueba a donde pertenece lógicamente. Retirar cualquier duplicación que pueda encontrar. Asegurarse de que los nombres de variables y métodos representan su uso actual. Aclarar cualquier construcción que podrían ser malinterpretados. Después de la refactorización hay que volver a ejecutar los casos de prueba, el desarrollador puede estar seguro de que la refactorización de código no está dañando alguna funcionalidad previamente existente.

6. **Repetir** : Se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requerimientos de diseño, por ejemplo que una funcionalidad esté desacoplada de otra. Después se escoge otro requisito y se empieza una nueva prueba, repitiendo el ciclo anterior.

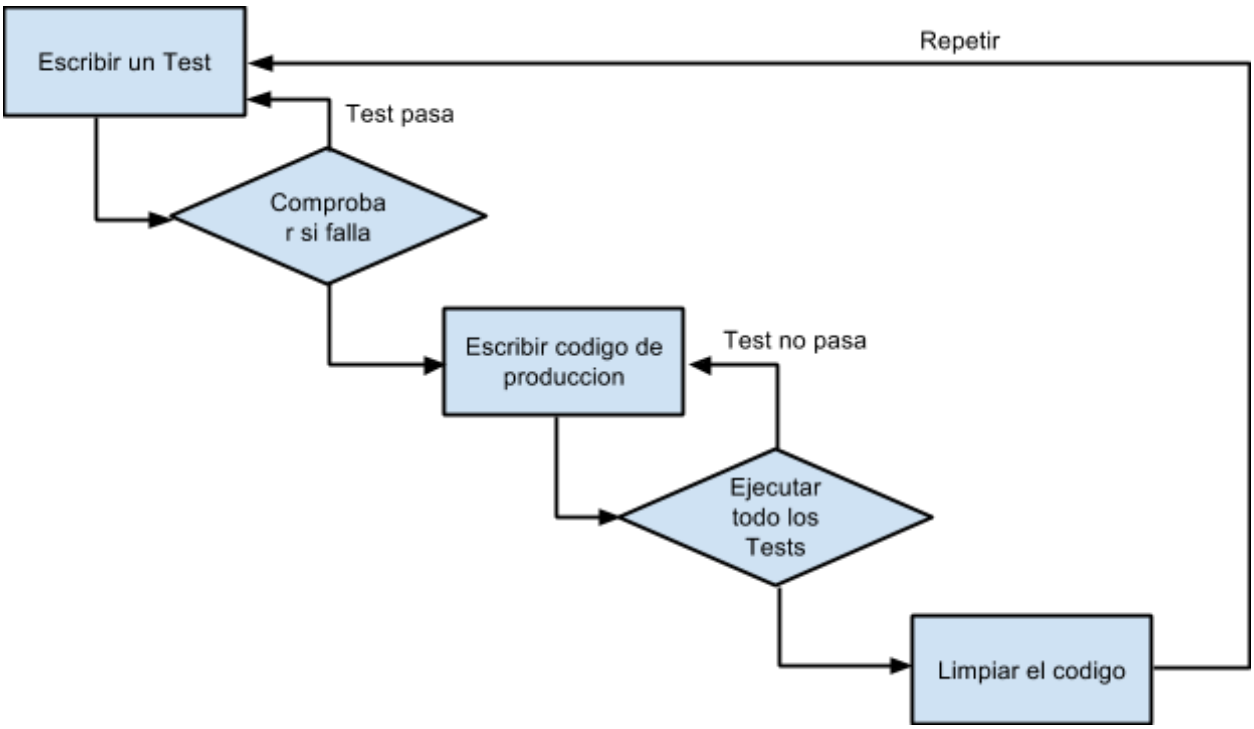


Diagrama del ciclo del desarrollo dirigido por pruebas

Tener un único repositorio universal de pruebas facilita complementar el Desarrollo guiado por pruebas con otra práctica recomendada por los procesos ágiles de desarrollo, la "Integración Frecuente". La integración frecuente de nuestro trabajo con el del resto del equipo de desarrollo permite ejecutar toda batería de pruebas y así descubrir si nuestra última versión es compatible con el resto del sistema. Es recomendable y menos costoso corregir pequeños problemas cada pocas horas que enfrentarse a problemas enormes cerca de la fecha de entrega fijada.

Las ventajas que se han obtenido al aplicar esta metodología son muchas. Una de las más importantes es que las pruebas no han sido una simple validación del cumplimiento de los requisitos, sino que también han guiado el diseño del framework. Al centrarse primero en los casos de prueba, uno debe imaginar cómo los clientes van a utilizar una determinada funcionalidad. Por lo tanto al programador solo le importa la interfaz y no la implementación. Esto resulta en que la estructura de las clases sea muy simple y clara, la comunicación o la API pública de las clases está bien definida y consistente y en general que la arquitectura de la aplicación sea flexible.

Por otro lado al aplicar TDD, se ha conseguido tener una batería de pruebas muy completa y cada clase tiene extensos casos de pruebas. Esto ayuda a la documentación de la propia clase y a la descripción de cómo usar esta clase en el contexto del framework. Sin embargo, aplazando la escritura de los casos de prueba al final del ciclo del desarrollo sería muy difícil llegar al

número y calidad de las pruebas que se obtuvieron al hacer la implementación y las pruebas a la vez.

5.1 Desarrollo dirigido por comportamiento (Behavior-driven development)

Behavior-driven development (BDD) es un proceso de desarrollo de software que se basa en el desarrollo guiado por pruebas (TDD). BDD combina las técnicas generales y principios de TDD con ideas de diseño dirigido por dominio y análisis orientado a objetos.

En su esencia, el BDD es una versión especializada de desarrollo basado en pruebas que se centra en la especificación del comportamiento de las unidades de software.

BDD elige utilizar un formato semi-formal para la especificación de comportamiento que está tomado del campo de las historias de usuario desde el campo del análisis y diseño orientado a objetos.

La estructura que debe tener este tipo de historias de usuarios o especificaciones es la siguiente :

- **Un título** que sea claro y explícito.
- **Una sección introductoria** corta que describe quien es el principal interesado en esta especificación y quienes son los afectados por ella.
- **Un conjunto de escenarios** que describen a cada caso por el que está compuesta la especificación. Un escenario tiene:
    - Una condición inicial que se asume es verdad al inicio de este escenario. Puede estar formada por una o varias cláusulas.
    - Después se precisa el evento que despierta o ejecuta el escenario.
    - Por último se indica el resultado esperado al terminar la ejecución del escenario.

BDD no tiene ningún requisito formal que especifique exactamente cómo deben de escribirse estas historias de usuario, pero insiste en que cada equipo utilizando BDD con un formato simple y estandarizado que incluye los elementos enumerados anteriormente.

Algunas de las principales herramientas que soportan el desarrollo dirigido por comportamiento son:

- **Cucumber,** herramienta escrita en Ruby.
- **RSpec,** otra herramienta escrita en Ruby.
- **JBehave,** framework escrito en Java
- **Jasmine.js,** es una librería escrita en JavaScript.

5.2 Herramienta utilizada : Jasmine.js

Se ha optado por el uso de Jasmine.js porque es una librería pequeña y simple. Está dirigida desde el principio a JavaScript y ofrece una API muy fácil de utilizar. No tiene dependencias externas y tampoco requiere DOM para ejecutarse.

El núcleo de la sintaxis de Jasmine.js consiste en dos palabras clave, ”**describe**” y “**it**”.

Un conjunto de pruebas comienza con una llamada a la función global Jasmine **describe** que acepta dos parámetros: una cadena y una función. La cadena es el nombre o el título de la suite de especificaciones - por lo general lo que está bajo prueba. La función es un bloque de código que implementa la suite.

Las especificaciones (Specs) se definen llamando a la función Jasmine “**it”**, la que, al igual que **describe** acepta como parámetros una cadena y una función. La cadena es el título de esta especificación y la función es la especificación, o prueba. Una especificación contiene uno o más expectativas que ponen a prueba el estado del código en el test.

Un ejemplo de esto sería :

```
describe("Una suite", function() {
  it("contiene especificaciones con expectativas", function() {
    expect(true).toBe(true);
  });
});
```

Con las simples primitivas anteriores se pueden hacer muchas suites complejas. Esto es así porque la directiva **describe** puede contener dentro de ella otras directivas **describe** y directivas **it**. Esto es posible gracias a que tanto describe, como it aceptan como segundo parámetro una función JavaScript, y tanto describe como it son funciones JavaScript normales.

Aparte de estas dos primitivas, Jasmine.js ofrece otras dos primitivas muy útiles. Se denominan **beforeEach** y **afterEach.** Como su nombre indica se llaman antes y después de cada especificación en un bloque **describe**.

Otra de las funcionalidades que ofrece Jasmine.js es un concepto denominado **Spy.** Este es un concepto que en otras librerías de pruebas se suele denominar "mock" u objeto simulado, es decir, un objeto que imitan el comportamiento de objetos reales de una forma controlada. De este modo Jasmine.js nos permite probar una clase que depende de otras clases para su funcionamiento, sin la necesidad de tener o referencias a estas otras clases.

Esto hace que las pruebas generadas utilizando **Spies** sean muy robustas y no fallen, aunque la clase que se esté imitando falle o ni siquiera esté implementada aun.

Para ejecutar todas las suites definidas, Jasmine.js nos ofrece una clase llamada **HTMLReporter.**

Primero paso es crear una página web donde se incluyen los ficheros que definen todas las suites y todo el código que las suites prueban. Después se llama a **HtmlReporter** y a **jasmineEnv.execute()** para ejecutar las pruebas.

Una vez se hayan ejecutado las pruebas, el resultado de las mismas se mostrará por pantalla en la página web.

6 Implementación

Para la implementación de Cuore.js se ha optado por el lenguaje JavaScript. Esto se debe a que JavaScript es el lenguaje de scripting más extendido y un estándar en internet. Además una de las razones más importantes es que JavaScript es soportado por todos los navegadores existentes, tiene una comunidad de usuarios muy grandes, y un gran conjunto de librerías y frameworks disponibles.

6.1 Class.js

El sistema de prototípico de JavaScript es bastante flexible, pero también un poco complejo, por eso hemos creado una herramienta que simplifique el trabajo con objetos y clases.

La clase Class.js representa el sistema de clases en Cuore.js. Es un sistema donde se permite la herencia y la sobre escritura de métodos con una apariencia y sintaxis más cercanos a otros lenguajes OOP.

El fichero que implementa la funcionalidad del Class se encuentre en “root/src/Cuore.Class.js”.

Class(padre, hijo)

```
CUORE.Class = function(Parent, props) {
```

Esta clase define una función, el constructor, que acepta como argumento a una clase padre y una clase hija y devuelve a una clase que tiene como contenido la clase “props” (clase hija) y tiene como padre la clase “Parent”.

```
    Parent || (Parent = Object);
    var hasOwn = Object.prototype.hasOwnProperty;
```

Primero se comprueba que se ha pasado una clase padre no nula. Si no es así, se le asigna como padre el objeto Object. Object es un objeto singleton en JavaScript y es el objeto padre del que descienden todos los demás objetos.

Por otro lado se crea una variable, que guarda la referencia a la función “hasOwnProperty”. Esta función devuelve un booleano dependiendo de si el objeto tiene una propiedad específica (variable o función).

```
    var Child = function() {
        Child.prototype.init.apply(this, arguments);
    };
```

Se define una clase “Child”, que consiste en una función en la que se llama al método “init” del propio prototipo. En esta función la llamada al método “init” representa el constructor del hijo. Se utiliza como contexto de “this”, el “this” del hijo y se le pasan los argumentos que se han pasado cuando se ha llamado a esta función.

Por tanto el método “init” se define por cada objeto y representa la lógica de su constructor. Esto puede sonar complicado ya que por ahora no tenemos ningún método “init” y el objeto Child consiste en una función.

Básicamente Child define una función constructora. Dentro de esa función constructora se llama al método “init” del propio objeto Child, y este método “init”, tiene como “this” la propia clase “Child”.

Con todo esto, tenemos la arquitectura para la creación del constructor de nuestro sistema de objetos.

```
    var F = function() {};
    F.prototype = Parent.prototype;
```

Se crea una clase “F” que almacena una función vacía, es decir, es un objeto vacío. Después al prototipo de “F” se le asigna el prototipo del padre. En este caso la variable “F” representa las propiedades del padre, que heredará el hijo.

```
    Child.prototype = new F();
    Child.parent = Parent.prototype;
    Child.prototype.constructor = Child;
```

Al prototipo de Child se le asigna un nuevo objeto del tipo “F”, es decir, un objeto que es vacío y tiene los métodos de la clase padre. A la propiedad “parent” de la clase Child se le asigna el prototipo del padre. También se asigna como constructor del prototipo de Child, la función que estaba asignada previamente a Child, es decir, el propio constructor.

En esencia esto representa la herencia de las propiedades del padre y la asignación del constructor a la clase hijo.

```
    for (var i in props) {
        if (hasOwn.call(props, i)) {
            Child.prototype[i] = props[i];
        };
    }
```

```
    return Child;
};
```

Se recorren todas las propiedades del objeto “props” y por cada propiedad “i” se comprueba si forma parte del objeto “props” o si es heredada. Si es verdad que forma parte, se añade a la clase “Child”. Estos son los métodos propios de la clase Child. En el objeto “props” se encuentra definida la función “init” utilizada en el constructor. Al final se devuelve la nueva clase Child, creada previamente.

6.2 Bus.js

El Bus representa una de las principales herramienta de comunicación en el framework Cuore.js. Su principal uso es permitir la comunicación entre los objetos Componente y los Servicios. Esta comunicación es tan desacoplada que ninguno de los dos tenga referencias al otro o sepa ni siquiera existencia.

El Bus es un objeto singleton, es decir, hay sólo una instancia de él. Por eso no utiliza el sistema de clases de Cuore.js y se define directamente en JavaScript.

El fichero que implementa la funcionalidad del Bus se encuentre en “root/src/Cuore.Bus.js”.

Clase Bus

```
CUORE.Bus = (function(undefined) {
    var subscriptions = [];
    var debugModeON = false;
```

Para la construcción de esta clase se utiliza un patrón muy extendido en el lenguaje JavaScript, llamado Módulo. Este patrón consiste devolver un objeto al ejecutar una función. De ese modo, si el output de esa función se asigna a una variable, al ejecutar esta función, a la variable se le asignará el objeto que ha devuelto la función.

En nuestro caso a la propiedad “Bus” del objeto CUORE, se le asigna el objeto devuelto por esta función anónima.

Se definen dos variables privadas de la clase Bus :

- “subscriptions” representa una lista con las suscripciones que existen en el sistema. El formato de cada suscripción es una tupla formada por el objeto que se ha suscrito y el nombre del evento al que está suscrito dicho objeto ([objetoSuscritor, NombreEvento]).

- “debugModeOn” indica de si queremos que salga en la consola JavaScript todos los mensajes que se mandan por el Bus. Se utiliza para fines de depuración, como su nombre indica

método subscribe(suscriptor, nombreEvento)

```
    var subscribe = function(subscriber, eventName) {
            if (!_validSubscriber(subscriber)) throw new Error("Not a subscriber (lacks
eventDispatch function)");
            if (!_subscriptionExists(subscriber, eventName)) {
                subscriptions.push([subscriber, eventName]);
            }
        };
```

Esta primera función “suscribe” hace lo que su nombre indica, suscribe a un objeto “subscriber” a un evento “eventName”.

Para hacer esta suscripción, primero se comprueba si el suscriptor es válido, es decir, el objeto suscriptor tiene definido un método “eventDispatch”.

Después se comprueba si es suscriptor esta dado de alta para ese tipo de evento, en cuyo caso no se hace nada más.

Si por el contrario el suscriptor aún no está dado de alta para ese evento, se almacena la tupla en la lista de suscripciones y se acaba la función.

método unsubscribe(suscriptor, eventos)

```
    var unsubscribe = function(subscriber, events) {

            if (typeof events == "string") {
                _removeSubscription([subscriber, events]);
                return;
            }

            for (var i = 0, len = events.length; i < len; i++) {
                var theSubscription = [subscriber, events[i]];
                _removeSubscription(theSubscription);
            }
        };
```

La función “unsubscribe” hace justo lo contrario que la función anterior. Elimina la suscripción de un objeto a un evento.

Este método acepta dos argumentos, el objeto suscriptor y un nombre de evento. Sin embargo también acepta una lista de eventos como segundo parámetro.

Dentro del método, lo primero se comprueba si se ha pasado solo un evento o una lista de eventos. Si es solo un evento se elimina de la lista de suscripciones llamando al método privado "_removeSubscription". Sin embargo, si es una lista de eventos, se recorre esta lista y se elimina cada suscripción llamando también al método "_removeSubscription".

método hasSubscriptions()

```
    var hasSubscriptions = function() {
            return (subscriptions.length > 0);
        };
```

El método "hasSubscriptions", comprueba que la lista de suscripciones no está vacía.

método subscribers(nombreEvento)

```
    var subscribers = function(theEvent) {
            var selectedSubscribers = [];
            for (var i = 0, len = subscriptions.length; i < len; i++) {
                var subscription = subscriptions[i];
                if (subscription[1] === theEvent) {
                    selectedSubscribers.push(subscription[0]);
                }
            }
            return selectedSubscribers;
        };
```

Este método se utiliza para devolver una lista, con todos los objetos suscritos a un evento determinado. Como argumento acepta el nombre del evento cuyos suscriptores queremos saber.

El modo para obtener esto es muy fácil. Primero se crea una lista con suscriptores seleccionados. Después Se recorre toda la lista de suscriptores y nos fijamos a qué evento está suscrito cada objeto. Si es el evento que nos interesa, añadimos el objeto suscriptor a nuestra lista de suscriptores seleccionados. Al final devolvemos esta lista, que puede o no estar vacía.

método emit(nombreEvento, parámetros)

```
    var emit = function(eventName, params) {
            var subscribersList = this.subscribers(eventName);

            debug("Bus.emit (event, params)");
            debug(eventName);
            debug(params);
            debug("------------");
```

```
            for (var i = 0, len = subscribersList.length; i < len; i++) {
                subscribersList[i].eventDispatch(eventName, params);
            }
        };
```

Esta es una función muy importante ya que se encarga de difundir los mensajes en el Bus. Recibe dos parámetros, el primero es el nombre del evento y el segundo es el mensaje que hay que difundir.

Para llevar a cabo la función de este método, lo primero que se hace es obtener una lista con todos los objetos que están suscritos al evento que hemos especificado. Esto se hace mediante la función definida anteriormente, llamada "subscribers".

Después se recorre esta lista de objetos y de cada objeto se llama a su método "eventDispatch" pasándole el nombre del evento y el mensaje.

Si el modo debug está activado, también se muestra en la consola JavaScript la información del evento y el mensaje transmitido.

método _subscriptionExists(suscriptor, nombreEvento)

```
    var _subscriptionExists = function(subscriber, eventName) {
            var result = false;
            var theSubscription = [subscriber, eventName];

            for (var i = 0, len = subscriptions.length; i < len; i++) {
                var subscription = subscriptions[i];
                var sameSubscriber = (subscription[0] === theSubscription[0]);
                var sameEvent = (subscription[1] === theSubscription[1]);
                if (sameSubscriber && sameEvent) {
                    result = true;
                    break;
                }
            }
            return result;
        }
```

Este método tiene un guión bajo, "_", al principio del nombre para indicar que es una función privada. Su funcionalidad consiste en comprobar que una tupla de suscripción existe en el sistema. Si la tupla existe se devuelve true y si por el contrario no existe se devuelve false. El método acepta a un objeto suscriptor y a un nombre de evento. Estos dos argumentos sirven para crear la tupla de suscripción llamada "theSubscription".

Para comprobar si dicha tupla existe, se recorre la lista de todas las suscripciones y se comprueba que existe una tupla igual que la que hemos creado anteriormente. Al encontrar que existe, se sale del bucle que recorre la lista y se devuelve true, en caso contrario se devuelve false.

método _removeSubscription(suscripción)

```
var _removeSubscription = function(theSubscription) {

        for (var i = 0, subscription; subscription = subscriptions[i]; i++) {
            var sameSubscriber = (subscription[0] === theSubscription[0]);
            var sameEvent = (subscription[1] === theSubscription[1]);
            if (sameSubscriber && sameEvent) {
                subscriptions.splice(i, 1);
            }
        }

    };
```

Este es otro método privado, que elimina una suscripción de la lista de suscripciones. Se pasa como argumento una tupla que contiene el objeto suscriptor y el nombre del evento, es decir, una tupla del tipo suscriptor.

Después se recorre toda la lista de suscriptores y si se encuentra una suscripción que sea igual que la que se ha pasado como argumento, se elimina de la lista.

Este método es utilizado internamente por el método “unsubscribe”.

método _validSubscriber(suscriptor)

```
var _validSubscriber = function(subscriber) {
        return subscriber.eventDispatch;
    };
```

Como podemos ver, el método “_validSubscriber” es un método privado que recibe como argumento a un objeto suscriptor. La sintaxis de JavaScript nos permite comprobar si un objeto define un método o no. La forma de hacer eso es llamando al método del objeto sin paréntesis. Es nuestro caso se comprueba que el objeto suscriptor tiene definido el método “eventDispatch”, que se utiliza para pasarle mensajes de los eventos que está suscrito.

métodos debug(objeto) , enableDebug() , disableDebug()

```
var debug = function(object) {
        if (debugModeON) {
            console.log(object);
        }
    };

var enableDebug = function() {
        debugModeON = true;
    }

var disableDebug = function() {
        debugModeON = false;
    }
```

Las tres funciones más abajo se ocupan de la depuración de esta clase.

Por un lado, la función “debug” acepta un objeto (preferentemente un String) que vuelca por la consola.

Por otro lado los métodos “enable Debug” y “disableDebug” hacen justamente lo que sus nombres indican, habilitan y deshabilitan la depuración del Bus.

interfaz pública de la clase Bus

```
    return {
        subscribe: subscribe,
        unsubscribe: unsubscribe,
        hasSubscriptions: hasSubscriptions,
        subscribers: subscribers,
        emit: emit,
        enableDebug: enableDebug,
        disableDebug: disableDebug
    };

})();
```

Se devuelve un objeto, con todos los métodos públicos. El lado izquierdo son los nombres de los métodos y el lado derecho consiste en los nombres de las funciones definidas anteriormente.

Todos los métodos privados también forman parte del objeto, pero no se incluyen en su API pública y no se pueden llamar externamente. Esto es otro de los beneficios del uso del patrón Module en JavaScript.

6.3 Handler.js

Esta clase representa el Controlador del MVC en Cuore.js. Se encarga de reaccionar ante determinados eventos. Puede desde cambiar el estado del Componente hasta hacer peticiones remotas o lanzar servicios mediante el Directorio de servicios. Se puede afirmar que es la lógica de la clase Component y lo que define su comportamiento.

El código fuente de esta clase se puede encontrar en “root/src/Cuore.Handler.js”.

definición de clase Handler

```
CUORE.Handler = CUORE.Class(null, {
```

```
    init: function() {
        this.owner = null;
    },
```

La clase se define utilizando el modelo de clases de Cuore.js. Como se puede ver en la primera línea no tiene padre, es decir, su padre el null. Después se declara el método constructor de la clase “init”, que crea a la propiedad “owner” con un valor nulo.

Normalmente el propietario de nuestro Handler, será un Component.

método handle()

```
    handle: function(params) {},
```

Este es el método más importante de esta clase. Es el método que se llamara cuando se invoque a este manejador. En este caso es vacío, pero en los manejadores que crearemos para nuestra aplicación heredarán de esta clase y lo sobrescribirán con su propio comportamiento.

método setOwner(objetoPropietario)

```
    setOwner: function(anObject) {
        this.owner = anObject;
    },
```

Este método sirve para cambiar el propietario de nuestro Handler. Es importante conocer qué Componente es nuestro propietario para poder interactuar con él y cambiar su estado si hace falta.

método getOwner()

```
    getOwner: function() {
        return this.owner;
    }
});
```

Este es el último método de la clase Handler.js. Su función es devolver el propietario actual del manejador.

6.4 HandlerSet.js

La clase HandlerSet se encarga de almacenar y organizar de todos los Handlers o manejadores de un Component. Una vez están organizados, se encarga de ejecutar los handlers específicos

para responder a un evento. Se puede entender como el punto de interacción entre el componente y sus manejadores.

Su implementación se puede encontrar en el fichero “root/src/CUORE.HandlerSet.js”.

definición de clase HandlerSet

```
CUORE.HandlerSet = CUORE.Class(null, {

    init: function() {
        this.eventNames = [];
        this.handlersForEvent = {};
    },
```

La clase HandlerSet se define en el sistema de clases de Cuore.js.

Por tanto el método “init” representa su constructor. En él se definen dos propiedades:

- ”eventNames” es una lista con todos los nombres de eventos para los que existen manejadores
- “handlersForEvent” representa un diccionario, donde las claves son los diferentes nombres de eventos y los valores son listas de manejadores que responden a dichos eventos.

método register(nombreEvento, manejador)

```
    register: function(eventName, aHandler) {
        if (!this._contains(eventName)) {
            this.eventNames.push(eventName);
        }
        var handlersForEvent = this.handlersForEvent[eventName] ||
(this.handlersForEvent[eventName] = []);

        handlersForEvent.push(aHandler);
    },
```

Este método, como su nombre indica, registra a un handler para ser llamado cuando es recibido un evento. Acepta dos argumentos, el nombre del evento, y el objeto handler que será llamado cuando se reciba dicho evento.

El funcionamiento de este método consiste en :

1. Se comprueba que el nombre del evento no está registrado, si es así se añade a la lista “eventNames”.
2. Se comprueba que el diccionario “handlersForEvent” tiene una entrada para el evento, si no es así se crea.

3. Por último se añade el objeto manejador a la lista del evento en el diccionario anterior.

método getManagedEvents()

```
getManagedEvents: function() {
    return this.eventNames;
},
```

Este método sirve para consultar qué eventos en total estamos manejando. Su implementación es muy sencilla y consiste en devolver la lista de los nombres de los eventos que tiene por lo menos un manejador asignado.

método notifyHandlers(nombreEvento, datosEvento)

```
notifyHandlers: function(eventName, eventData) {
    var handlersToNotify = this.handlersForEvent[eventName];
    if (handlersToNotify) this._notify(handlersToNotify,eventData);
},
```

El método “notifyHandlers” es llamado para avisar a todos los manejadores de la ocurrencia de un evento. Este método es público y forma parte de la API de esta clase.

Acepta dos argumentos, el primero es el nombre del evento y el segundo es un objeto que contiene los datos relacionados con este evento (puede ser un objeto nulo).

La implementación consiste en obtener la lista de manejadores para el evento y almacenarla en la variable “handlersToNotify”. Después se comprueba de que esta lista de manejadores no está vacía, y se proceda a llamar al método “_notify” con esa lista y los datos del evento.

métodos _notify(manejadores, datosEvento) , _safeNotification(manejador, datosEvento)

```
_notify: function(handlersToNotify, eventData) {
    for (var i = 0, len = handlersToNotify.length; i < len; i++) {
        this._safeNotification(handlersToNotify[i], eventData);
    }
},
_safeNotification: function(handler, eventData) {
    handler.handle(eventData);
},
```

El método privado “_notify” recorre a la lista de manejadores y por cada manejador llama a la función “_safeNotification” pasándole el propio manejador y los datos del evento.

En cuanto al método “_safeNotification”, su función es llamar al método “handle” del

manejador que ha recibido como argumento de la función anterior. A este método “handle” se le pasa como argumento los datos del evento.

Es en esta función “handle” del objeto manejador donde se hace el procesamiento del evento.

método _contains(nombreEvento)

```
    _contains: function(eventName) {
        var result = false;

        for (var i = 0, len = this.eventNames.length; i < len; i++) {
            if (this.eventNames[i] === eventName) {
                result = true;
                break;
            }
        }
        return result;
    }
});
```

Esta es una función auxiliar que sirve para comprobar que el evento que se le ha pasado como argumento esta contenido la lista de evento “eventNames”.

Para llevar a cabo esta acción, simplemente se recorre la lista de nombres de eventos y se comprueba que el nombre pasado como parámetro a la función está contenido dentro de esta lista.

6.5 Service.js

La clase de Service.js como su propio nombre indica implementa un servicio genérico y está pensada para ser heredada para la implementación de servicios concretos.

En Cuore.js, los Servicios son entidades compartidas por todos los componentes y se pueden reutilizar. Su función es amplia y pueden servir tanto para comunicación con servidores remotos, como para la implementación de sistemas de cache u otras acciones.

La implementación completa se encuentra en el fichero “root/src/CUORE.Service.js”.

definición de la clase Service

```
CUORE.Service = CUORE.Class(null, {
```

```
    init: function () {
        this.name = 'ABSTRACT';
        this.executionPrefix = 'EXECUTED';
        this.SEPARATOR = '_';
        this.baseURL = '';
    },
```

La clase Servicio se define mediante el sistema de clases de Cuore.js, lo que permite su herencia y la creación de servicios específicos.

Su método constructor define cuatro variables que sirven para localizar el servicio y para construir el evento que se emitirá en el Bus si el servicio necesita comunicar algo.

- "name", es el nombre del servicio
- "executionPrefix", es el prefijo que se añade al nombre del evento una vez se ha ejecutado el servicio. Se utiliza en la construcción del evento que se emitirá por el Bus.
- "SEPARATOR", es un carácter que sirve para separar diferentes partes del nombre del evento que se emitirá por el Bus.
- "baseURL", representa la dirección base del servidor con el que vamos a comunicar. Normalmente es la dirección base desde donde hemos descargado la aplicación por primera vez.

métodos execute(procedimiento, parámetros) , getEventNameForExecution(procedimiento)

```
    execute: function (procedure, params) {
        var eventName = this.getEventNameForExecution(procedure);
        this[procedure](params, eventName);
    },
    getEventNameForExecution: function (procedure) {
        return this.getName() + this.SEPARATOR + procedure + this.SEPARATOR +
this.executionPrefix;
    },
```

Este método es llamado por el Directory y es el punto de entrada en el servicio. Se ocupa de ejecutar el método del servicio que se ha solicitado.

Lo primero que se hace es llamar a la función "getEventNameForExecution" para poder construir el nombre completo del evento a partir del nombre del método y de propiedades de esta clase. Este nombre se guarda en la variable "eventName".

Después se llama al método de esta clase cuyo nombre es "procedure" y se le pasan como argumentos los parámetros recibidos por esta función y el nombre del evento creado anteriormente.

Por otro lado el método "getEventNameForExecution" devuelve un String representando el

nombre del evento que este servicio emitirá por el Bus cuando se termine de ejecutar la función “procedure”.

métodos request(url, parámetros, NombreEvento) , wrapRequestParams(parámetros)

```
    request: function (url, params, eventName) {
        var paramsData = this.wrapRequestParams(params)

        var callback = this._responseCallback(eventName);
        this._doRequest(url, paramsData, callback);
    },

    wrapRequestParams: function(params){
        var theMessage = new CUORE.Message();
        theMessage.putMapOnQuery(params);

        return theMessage.asJson();
    },
```

El método “request” es el encargado de hacer una petición a un servidor remoto y recibir su respuesta. Una vez que se recibe la respuesta del servidor remoto se emite un mensaje por el Bus de Cuore.js con la posible respuesta.

El método acepta tres parámetros. El primer parámetro es la url del servidor remoto donde se encuentra el servicio remoto, el segundo parámetros es un objeto del tipo clave-valor que contiene diferentes parámetros que enviaremos con la petición. El último parámetro representa el nombre del evento que se emitirá por el Bus.

Lo primero que hacemos en la implementación es “envolver” el objeto de “params” es un objeto del tipo JSON. Esto se hace con una llamada a la función “wrapRequestParams”.

Después se crea una función callback, que será llamada una vez que el servicio remoto devuelve el resultado de la petición. Esta función callback se construye llamando al método “_responseCallback” y se guarda en la variable “callback”.

Por último se llama a la función privada “_doRequest” que se encarga de hacer la petición.

método _doRequest(url, parámetros, callback)

```
    _doRequest: function (url, paramsData, callback)
    {
        CUORE.Core.request(url, paramsData, callback);
    },
```

El método “_doRequest” es un método muy simple y sirve de wrapper al método auxiliar “request” del objeto singleton Core. A este método se le pasa la url de la petición, los parámetros en formato JSON y una función callback que se llama una vez se finaliza la

petición.

métodos emit(nombreEvento, respuesta) , wrapResponse(respuesta)

```
    emit: function (eventName, response) {
        var theMessage = this.wrapResponse(response);
        CUORE.Bus.emit(eventName, theMessage);
    },

    wrapResponse: function(response){
        return new CUORE.Message(response);
    },
```

El método “emit” se ocupa de emitir un mensaje por el Bus. Acepta dos parámetros, el primero es el nombre del evento y el segundo es un objeto de datos.

Lo primero que se hace es “envolver” el objeto de datos en un objeto tipo Message. Después ese objeto Message junto con el nombre del evento se envían por el Bus a todos los interesados.

métodos getName() , getBaseURL() , setBaseURL(url)

```
    getName: function () {
        return this.name;
    },

    getBaseURL: function () {
        return this.baseURL;
    },

    setBaseURL: function (baseURL) {
        this.baseURL = baseURL;
    },
```

Estas tres funciones son muy simples y hacen exactamente lo que se nombre indica.

método _responseCallback(nombreEvento)

```
    _responseCallback: function(eventName) {
        var emit = this.emit;

        var callback= function(response) {
            this.emit(eventName, response);
        }
        return CUORE.Core.bind(this,callback);;
    }
});
```

Este método privado se utiliza por el método público “request” para construir una función

callback, que será llamada una vez se complete la petición al servidor remoto.

Lo primero que se hace es definir una función llamada “callback” que acepta un objeto de datos y lo emite por el Bus junto con el nombre del evento. Por las propiedades del lenguaje JavaScript esta función puede acceder a la variable “eventName”, porque se guarda el contexto junto con la función.

Después se llama a un método auxiliar “bind” de objeto singleton Core. Esta función devuelve una función nueva que esta enlazada a esta clase, aunque se use fuera de ella.

De ese modo se permite llamar a métodos de esta clase desde la callback, aunque esta se use fuera del contexto de la clase Service.

6.6 Directory.js

Esta clase implementa el patrón clase de directorio encontrado en muchas arquitecturas SOA. Consiste en un índice o almacén donde se guardan todos los servicios del sistema. De este modo cualquier componente de Cuore.js puede llamar a cualquier servicio sin tener que poseer una referencia específica a él. Con tener una referencia del objeto Directory puede acceder a todos los servicios que este proporciona. De ese modo consigue un desacoplamiento entre los servicios y los componentes que los utilizan. El único punto en común entre los dos es el directorio de servicios.

La implementación de esta clase se encuentra en el fichero “root/src/CUORE.Directory.js”.

definición de la clase Directory

```
CUORE.Directory = CUORE.Class(null, {

    init: function(baseURL) {
        this.listing = [];
        this.services = {};

        this.setBaseURL(baseURL);
        this._addBuiltinServices();
    },
```

La clase Directory es creada con el sistema de clases de Cuore.js.

Define dos variables básicas:

- “listening”, es una lista con nombres de servicios que se proporcionan por el

directorio.

- “services”, es un diccionario que relaciona el nombre del servicio con el propio objeto de tipo Service que ofrece la funcionalidad.

Después se establece la url base de todos los servicios, mediante la llamada a la función “setBaseURL”.

Por último se definen una serie de servicios integrados en el framework Cuore.js

método add(servicio)

```
add: function(aService) {
    var serviceName = aService.getName();
    this.listing.push(serviceName);

    aService.setBaseURL(this.baseURL);
    this.services[serviceName] = aService;
},
```

El método “add” se utiliza para añadir un nuevo servicio a este directorio. Acepta un objeto de tipo Service como único parámetro.

El primer paso de la implementación consiste en obtener el nombre del servicio y almacenar ese nombre en la lista de nombres “listing”. Después se llama al método “setBaseURL” del servicio para configurar su url base. Una vez la url base del servicio es la misma que la url base del directorio, se añade el servicio y su nombre al diccionario “services”.

método execute(nombreServicio, nombreProcedimiento, parámetros)

```
execute: function(serviceName, procedureName, params) {
    this.getService(serviceName).execute(procedureName, params);
},
```

Este método es llamado por cualquier componente que quiere invocar un servicio.
Se aceptan tres parámetros, el primero es el nombre del servicio que se invoca, el segundo es el método del servicio en el cual estamos interesados y el tercer parámetro representa cualquier otro dato que queramos pasar como argumento al método del servicio.

La implementación consiste primero en encontrar el servicio mediante el método “getService” y una vez encontrado se ejecuta su método “execute” pasándole como parámetros el nombre de la función del servicio en la que estamos interesados ejecutar y cualquier otro parámetros que se necesite por ese método.

métodos getService(nombreServicio) , _findService(nombreServicio)

```
    getService: function(serviceName) {
        var service = this._findService(serviceName);
        return service || new CUORE.Services.Null();
    },
    _findService: function(serviceName) {
        return this.services[serviceName];
    },
```

Este es un método auxiliar que busca un servicio mediante el nombre “serviceName” y si no se encuentra se devuelve un servicio nulo.

La búsqueda del servicio se lleva a cabo mediante la función privada “_findService”. Esta función simplemente devuelve el valor que corresponde a la clave “serviceName” en el diccionario de servicios.

método setBaseURL(url)

```
    setBaseURL: function(baseURL) {
        this.baseURL = baseURL || '';
        var serviceNames = this.listing;
        var numberOfServices = serviceNames.length;

        for (var i = 0; i < numberOfServices; i++) {
            this._findService(serviceNames[i]).setBaseURL(this.baseURL);
        }
    },
```

El método “setBaseUrl” hace principalmente dos funciones. La primera función es almacenar la url base en la propiedad “baseURL” de la clase Directory para su uso en nuevos servicios. Por otro lado recorre todos los servicios del directorio y llama a sus métodos “setBaseURL” para cambiar su url base.

método _addBuiltinServices()

```
    _addBuiltinServices: function() {
        this.add(new CUORE.Services.Label());
        this.add(new CUORE.Services.Button());
    }
});
```

Por último este método añade unos servicios básicos que ya vienen implementados en Cuore.js

6.7 Component.js

Component es una de las clases centrales en Cuore.js. Representa a un componente de la aplicación web y normalmente está relacionado con algún tipo de componente gráfico, un botón, un campo, un enlace, etc. pero también puede representar otras cosas invisibles en la aplicación web.

Por otra parte se encarga de almacenar el estado del componente y por lo tanto es el Modelo en el patrón MVC.

Otra de las funciones que desempeña la clase Component es la unión entre los Servicios y los Handlers mediante el Bus. Esta clase se encarga de ejecutar diferentes servicios, después recibir su respuesta mediante el Bus y ejecutar los manejadores específicos. Una vez ejecutados los handlers y si se ha producido un cambio de estado, el componente ejecuta el render para actualizar la interfaz de la aplicación web.

La implementación de esta clase se encuentra en el fichero “root/src/CUORE.Component.js”

definición de la clase Component

```
CUORE.Component = CUORE.Class(null, {

    init: function() {
        this.setHandlerSet(new CUORE.HandlerSet());
        this.name = this._generateUUID();
        this.procedure = 'nullProcedure';
        this.SEPARATOR = '_';
        this.labels = {};
        this.renderer = new CUORE.Renderer();
        this.enabled = true;
        this.behaviour = CUORE.Behaviours.APPEND;
    },
```

Dado que la clase Component está pensada para ser heredado y especializada, se crea como parte del sistema de objetos de Cuore.js.

En el constructor “init” se definen varias propiedades :

- Se crea un nuevo HandlerSet y se llama el método “`setHandlerSet`” para asignar ese objeto a la propiedad “handlerSet”.
- “name”, es el identificador de este componente y se obtiene llamando a la función privada “_generateUUID”, que genera un UUID (Universally unique identifier).
- “labels”, es un diccionario de clave-valor, que almacena diferentes etiquetas del componente.

- “render ”, es un objeto tipo Render que se utilizará para mostrar esta clase en la página web.
- “enable”, es un flag que representa si el componente está habilitado o no.
- “behaviour”, especifica el comportamiento del componente, es decir, si se “secuestra” un elemento del DOM o se crea uno nuevo.

métodos setHandlerSet(handlerSet) , setDirectory(directory) , setRenderer(renderer)

```
setHandlerSet: function(handlerSet) {
    this.handlerSet = handlerSet;
},

setDirectory: function(directory) {
    this.services = directory;
    this.requestLabelText();
},
setRenderer: function(renderer) {
    this.renderer = renderer;
},
```

Los siguientes tres métodos asignan diferentes objetos de infraestructura a la clase.

El primer método llamado “setHandlerSet” asigna un objeto tipo HandlerSet a la propiedad “handlerSet”. Este representa el repositorio de manejadores que tiene el componente y normalmente es privado para cada instancia de Component.

“setDirectory” es un método que asigna un directorio de servicios a la propiedad “services” y llama al método “requestLabelText” para obtener las etiquetas de este componente.

“setRenderer”, como su nombre indica, es un método que asigna a la propiedad “renderer” un objeto de tipo Renderer.

métodos behave(comportamiento) , doYouReplace() , doYouHijack()

```
behave: function(behaviour) {
    this.behaviour = behaviour;
},

doYouReplace: function() {
    return this.behaviour === CUORE.Behaviours.REPLACE;
},

doYouHijack: function() {
    return this.behaviour === CUORE.Behaviours.HIJACK;
},
```

Un componente tiene tres tipos de comportamiento con respecto a los elementos DOM :

- APPEND, añadir más subelementos al elemento DOM donde se encuentra este componente.
- REPLACE, reemplazar el elemento DOM actual.
- HIJACK, “secuestrar” el elemento DOM, es decir, poder añadir, reemplazar o cualquier otra acción.

El comportamiento de un componente es muy importante porque se utilizar por el Renderer para saber cómo “pintar” el componente en la página web.

EL primer método asigna un comportamiento que recibe como argumento a la propiedad “behaviour”.

“doYouReplace” y “doYouHijack” son métodos que comprueban, cómo sus nombres indicar, si el comportamiento de este objeto es reemplazar o “secuestrar” un elemento DOM.

métodos draw() , updateRender()

```
draw: function() {
    this.renderer.render(this);
},

updateRender: function() {
    this.renderer.update(this);
},
```

El método “draw” se utiliza para dibujar el componente en la pantalla. Consiste en llamar al método “render” de la propiedad “renderer”, pasando como argumento al propio componente.

El segundo método “update”, se utiliza para actualizar el estado del Renderer, pero sin actualizar el componente en la pantalla.

método destroy()

```
destroy: function() {
    this.renderer.erase();
    CUORE.Bus.unsubscribe(this, this.getManagedEvents());
},
```

Esta función se llama antes de destruir o eliminar una instancia de la clase Component.

Lo primero que se hace es quitar el componente del DOM mediante el método “erase” del objeto render.

Después se eliminan las suscripciones de este componente en el Bus. Para eso se llama al método “unscubscribe” del objeto Bus. Se le pasan como primer argumento el propio objeto componente y como segundo argumento una lista de todos los eventos a los que está suscrito el componente, mediante la función “getManagedEvents”.

método execute(nombreServicio, nombreProcedimiento, parámetros, asíncrono)

```
    execute: function(theService, theProcedure, params, asynchronous) {
        if (!this.services) throw new Error("Cannot call service. A service directory is not
configured");
        this.services.execute(theService, theProcedure, params, asynchronous);
    },
```

“Execute” es el método de la clase Component que se encarga de invocar los servicios necesarios. Consiste en un wrapper alrededor de la propiedad “services”, que en realidad es un objeto del tipo Directorio.

La función acepta cuatro argumentos:

- “theService”, representa el nombre del servicio que estamos buscando.
- “theProcedure”, es el nombre del procedimiento que queremos invocar. Hay que señalar que el método debe estar declarado como un método público del servicio anterior.
- “params” es un objeto que pasamos como argumento al procedimiento anterior.
- “asynchronous” es un flag que representa si queremos invocar el servicio síncrona o asíncronamente.

Una vez que hemos comprobado que la propiedad “services” no es nula, es decir tenemos un directorio de servicios, ejecutamos el método “execute” de este directorio pasándole los argumentos recibidos por esta función.

método eventDispatch(nombreEvento, parámetros)

```
    eventDispatch: function(eventName, params) {
        this.handlerSet.notifyHandlers(eventName, params);
    },
```

Este método es llamado cuando el Bus tiene un evento en el que este componente está suscrito. Los objetos tipo Component, son los principales suscriptores en el Bus de datos. Sin embargo la clase Bus requiere que cualquier objeto que quiere suscribirse a un evento y recibir notificaciones tiene que implementar un método llamado “eventDispatch”.

Una vez que el Bus llama a este método, el componente notifica a sus manejadores,

pasándoles el nombre del evento y cualquier dato que acompañe al evento.

La notificación a los manejadores se hace por medio de la propiedad “handlerSet” y más en concreto por el método “notifyHandlers”.

métodos addHandler(nombreEvento, manejador) , addExecHandler(nombreEvento, manejador)

```
    addHandler: function(eventName, handler) {
        handler.setOwner(this);
        this.handlerSet.register(eventName, handler);
        CUORE.Bus.subscribe(this, eventName);
    },

    addExecHandler: function(eventName, handler) {
        this.addHandler(eventName, new CUORE.Handlers.Executor(handler));
    },
```

“AddHandler”, como su nombre indica, es un método que añade un nuevo manejador al componente. Acepta el nombre del evento para el que se invocara el manejador y el propio objeto manejador.

La primer instrucción de la implementación pone como propietario del manejador, este componente. Una vez inicializado, el handler se registra en el HandlerSet junto a su evento.
Por último, se suscribe el componente al Bus, pasándole como argumentos el propio componente y el nombre del evento para el que se ha registrado el manejador.

Por otro lado, el método “addExecHandler” es un método wrapper alrededor del método “addHandler” y se utiliza para añadir un tipo específico de handlers llamados Executors.

métodos addClass(claseCSS) , removeClass(claseCSS)

```
    addClass: function(aClass) {
        this.renderer.addClass(aClass);
    },

    removeClass: function(aClass) {
        this.renderer.removeClass(aClass);
    },
```

Estos dos métodos sirven para añadir/eliminar clases CSS al elemento DOM del componente. Su acción se lleva llamando al objeto renderer y pasándole el nombre de la clase CSS que queremos añadir o eliminar.

métodos getText(clave) ,setText(clave, valor)

```
getText: function(key) {
    if(!key) return null;

    return this.labels[key];
},
setText: function(aKey, aText) {
    this.labels[aKey] = aText;
    this.updateRender();
},
```

El primer método obtiene una etiqueta identificada por la clave “key” del diccionario “labels”. Si no existe una etiqueta que tenga esa clave se devuelve null. El segundo método almacena una clave y un valor en el diccionario de etiquetas.

Después se actualiza el renderer mediante el método “updateRender” para reflejar los cambios de estado en el componente.

métodos getName() , setName(nombre)

```
getName: function() {
    return this.name;
},

setName: function(aName) {
    this.name = aName;
},
```

Estos dos procedimientos, como sus respectivos nombres indican, sirven para obtener y cambiar el nombre del componente.

método setContainer(contenedor)

```
setContainer: function(container) {
    if (this.doYouHijack()) this.setName(container);
    this.renderer.setContainer(container);
},
```

El Container representa el elemento DOM que contiene al componente actual, es decir su padre en la jerarquía DOM. En esta función cambiamos el Container actual por uno nuevo. Si el componente tiene un comportamiento de “secuestro”, lo único que hacemos es cambiar el nombre del componente por el nombre del contenedor, de los contrario llamamos al método “setContainer” del objeto renderer.

método getManagedEvents()

```
getManagedEvents: function() {
    return this.handlerSet.getManagedEvents();
},
```

Este método se utiliza para obtener una lista con los nombres de todos los eventos a los que está suscrito este componente y por lo tanto para los que tiene manejadores.

Para obtener dicha lista se llama al método “getManagedEvents” de la propiedad handlerSet.

método setI18NKey(clave)

```
setI18NKey: function(key) {
    if (!key) return;

    this.setText(key, key);

    this.addHandler('LABELS_getLabel_EXECUTED_' + key, new CUORE.Handlers.SetText());
    this.requestLabelText(key);
},
```

Los componentes Cuore.js aceptan internacionalización (i18n) por defecto y este método se encarga de ello.

Lo primero que hacemos es reservar la clave en el diccionario de etiquetas, mediante la función “setText”. Después añadimos un manejador especial incluido en Cuore.js llamado “SetText”, que será llamado cuando tengamos el valor de nuestra etiqueta.

Por último llamamos a la función “requestLabelText” para obtener el valor de la etiqueta (posiblemente de un servidor remoto).

método requestLabelText(clave)

```
requestLabelText: function(aKey) {

    if(!aKey){
        for(var key in this.labels){
            this._executeLabelsService(key);
        }
    }
    else{
       this._executeLabelsService(aKey);
    }
},
```

El procedimiento “requestLabelText” se ocupa de obtener una etiqueta identificada por “aKey” desde un servicio. Si no se pasa una clave como argumento, se obtienen las etiquetas de todas las claves en el diccionario de etiquetas (esto puede tardar bastante, dependiendo

del número de etiquetas). En cualquiera de los dos casos, este método llama al método privado "_requestLabelsService" con una clave, para obtener la etiqueta.

método _executeLabelService(clave)

```
_executeLabelsService:function(aKey){
    if (!this.services) return;
     this.services.execute("LABELS", 'getLabel', {
            key: aKey
       }, true);
},
```

Este método encapsula una llamada al servicio de etiquetas "LABELS", y en concreto, su procedimiento "getLabel". Para hacer la invocación de este servicio se utiliza el directorio de servicios "services" y se pasa como clave el argumento recibido por esta función.

métodos isEnabled() , enable() , disable()

```
isEnabled: function() {
    return this.enabled;
},

enable: function() {
    this.enabled = true;
    this.updateRender();
},

disable: function() {
    this.enabled = false;
    this.updateRender();
},
```

Los métodos "enable"/"disable" tiene la función de habilitar/deshabilitar el componente mediante la actualización de la propiedad "enabled". Después de cualquier cambio del estado del componente se llama al método "updateRender" para actualizar el componente en pantalla.

Por otra parte, el método "isEnabled" es un método de consulta y obtiene el estado actual del componente.

método addDecoration(decorador)

```
addDecoration: function(decoration) {
    if (decoration instanceof CUORE.Decoration) {
        this.renderer.addDecoration(decoration);
    }
},
```

“addDecoration” sirve para añadir un decorador al render actual. El decorador puede cambiar el output de render y últimamente cambiar la apariencia del componente en la página.

Primero se comprueba que el objeto pasado es una instancia de la clase Decoration. Si esto es cierto se añade al render.

métodos onEnvironmantUp() , _generateUUID()

```
    onEnvironmentUp: function() {},

    _generateUUID: function() {
        return (((1 + Math.random()) * 0x10000) | 0).toString(16).substring(1);
    }
});
```

“onEnvironmentUp” es un método de inicialización, que es llamado para llevar a cabo acciones que necesitan de que el framework y el DOM estén completamente cargados. Está pensado para casos en los que el uso del constructor no es adecuado.

Por último el método privado “_generateUUID”, como su nombre indica, genera un UUID. Se utiliza en el nombramiento de los componentes ya que un UUID nunca se puede repetir.

7 Conclusión

Cuore.js fue creado con la misión de hacer una implementación del paradigma SOFEA. Se puede decir con total seguridad que esta misión fue cumplida. Cuore.js permite la creación de aplicaciones Web de una manera rápida y sencilla siguiendo la ideología SOFEA, es decir, una interfaz con arquitectura de servicios.

Mediante la utilización de varios patrones de diseño Cuore.js tiene una arquitectura muy modular que permite una clara distinción de las partes en aplicaciones más grandes. Esto permite que su utilización en aplicaciones complejas y escalables sea muy oportuna. Al tener su propio sistema de objeto, los desarrolladores que utilizan Cuore.js se pueden beneficiar de todas las metodologías y buenas prácticas OOP ya existentes.

Por otra parte, la comunicación estructurada con servidores remotos permite escalar muy fácilmente la aplicación Web, manteniendo el código de la aplicación modular y fácil de mantener y cambiar.

La implementación del framework se ha mantenido lo más sencilla posible y rara vez el código fuente contiene más de 100 líneas por fichero.

Por último, la utilización de técnicas como TDD en el desarrollo de Cuore.js lo convierte en un framework muy robusto y capaz de soportar cargas muy grandes.

7.1 Posibles mejoras en Cuore.js

Cuore.js es un framework en continuo desarrollo. La versión que se ha discutido en este PFC es la primera versión estable de Cuore.js. Esta versión ha sido utilizada en muchas aplicaciones web de producción, con un intenso tráfico de datos. Por eso en su presente forma está preparado para soportar la mayoría casos de uso clásicos en las aplicaciones web. Sin embargo esto no quiere decir que Cuore.js no debería mejorar.

Durante el desarrollo de varias aplicaciones web se han notado varios aspectos del framework donde el actual workflow podría ser mejorado.

Una mejora al framework es no usar Renderers sino capturar un elemento del DOM que ya está en la página. La idea de los Renders parecía muy acertada en el momento de creación de Cuore.js. Sin embargo conforme se han ido desarrollando proyectos con él, se ha visto que los

beneficios, que proporciona la capa de complejidad de los Renderers y Decorators, no son muchos en la práctica. Se podría mantener la flexibilidad si exponemos directamente el HTML en el framework y aplicamos plantillas o directamente manipulamos el propio HTML.

Un lugar perfecto para exponer ese elemento HTML seria la clase Handler, ya que es la clase que se ocupa del cambio de estado. De esa manera mantendremos todo el cambio de estado y su actualización en la pantalla juntos y no haría falta una comunicación complicada como es el caso actual. Un inconveniente de este cambio sería que el framework requeriría demás convenio.

Otra mejora es hacer más visible el wiring (enlazado) o la conexión entre objetos, por ejemplo como unos objetos escuchan a otros, que unos objetos dependen de otros, etc.

Eso podría solucionarse mediante un DSL y en vez de crear una subclase del objeto Page, hay un Builder (objeto estático) que produce la página configurada a través del DSL. Es un ejemplo clásico de DSL que construye un modelo semántico y lo entrega. De esta manera el usuario de Cuore.js podría ver mejor los enlaces entre los diferentes objetos y su interacción. Además esto ayudaría mucho al uso del framework y la creación de aplicaciones web.

7.2 El futuro de Cuore.js

El futuro del desarrollo de aplicaciones web es un futuro muy prometedor pero a la vez muy difícil de predecir.

El gran crecimiento en la venta de dispositivos móviles y el aumento de usuarios de internet provoca un gran aumento en aplicaciones web. Estas aplicaciones web cada vez más tienen un comportamiento y funcionalidades parecidas a las aplicaciones nativas. Un ejemplo de aplicaciones web con comportamiento nativo sería Google Docs, que permite crear documentos ofimáticos y es un competidor directo a las aplicaciones nativas de Microsoft.

En este contexto de gran expansión de las aplicaciones web, se necesitan herramientas cada vez más flexibles y optimizadas. Por eso actualmente se está desarrollando la segunda versión de Cuore.js, que intenta solucionar los problemas que existen en la versión actual, así como añadir nueva funcionalidad que facilita la creación de aplicaciones web cada vez más complejas.

Uno de pilares que se consideran para la versión 2 de Cuore.js es la simplificación de la API y su uso, pero manteniendo la flexibilidad y robustez del framework.

Otra característica importante que se implementará en Cuore.js 2 es un paquete más grande de manejadores y servicios ya hechos y listos para ser usados. Eso ayudará a una creación de aplicaciones más rápida.

Por otra parte, el proyecto Cuore.js es un proyecto Open Source y por eso se espera un cierto apoyo de la comunidad. Por eso con la versión 2 existen planes de hacer un sitio web donde la discusión entre usuarios sea más intuitivo y rápido. Eso ayudaría a la involucración de los mismos y por último el crecimiento y expansión de Cuore.js.

8 Bibliografía y referencias

1. Repositorio de código oficial del framework Cuore.js ,alojado en en el servicio Github *https://github.com/beCodeMyFriend/Cuore.js*

2. The Single Page Interface Manifesto (SPA Manifesto) ,según los creadores de ItsNat *http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php*

3. Ganesh Prasad, Rajat Taneja, Vikrant Todankar (2007) *Life above the Service Tier* (paper)

4. The "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994) *Design Patterns: Elements of Reusable Object-Oriented Software* (págs. 326 - 337) Addison-Wesley

5. Addy Osmani (2014) *Learning JavaScript Design Patterns* (págs. 26 - 39) O'Reilly Media http://addyosmani.com/resources/essentialjsdesignpatterns/book/

6. John Resig, JQuery, http://jquery.com/

7. Jeremy Ashkenas, Backbone.js, http://backbonejs.org

8. Robert Cecil Martin (2002) *Agile Software Development, Principles, Patterns, and Practices* (págs. 155 - 231) Prentice Hall

9. Kent Beck, (2002) *Test Driven Development: By Example* (págs. 102 - 129) Addison-Wesley Professional

10. Cucumber, Aslak Hellesøy, Joseph Wilk, Matt Wynne, Gregory Hnatiuk, Mike Sassak, http://cukes.info/

11. Addy Osmani, (2012) *Learning JavaScript Design Patterns (*Module Pattern*)*, O'Reilly Media, http://addyosmani.com/resources/essentialjsdesignpatterns/book/

12. Marijn Haverbeke, (2011) *Eloquent JavaScript: A Modern Introduction to Programming* (cap. 4 Data structures: Objects and Arrays), No Starch Press, http://eloquentjavascript.net/

13. Douglas Crockford, (2008) *JavaScript: The Good Parts* (págs 20 - 58), O'Reilly Media

14. Mark Pilgrim, (2010) *HTML5: Up and Running* (págs 120-165), O'Reilly Media

15. Cal Henderson, (2006) *Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications* (cap. 2 Web Application Architecture), O'Reilly Media

16. Ethan Marcotte,(2011) *Responsive Web Design (Brief Books for People Who Make Websites, No. 4)*, A Book Apart

17. Donald A. Norman, (2002) *The Design of Everyday Things*, Basic Books