MODELO DE ADECUACIÓN Y ARQUITECTURA DE APLICACIONES UTILIZANDO PATRONES ARQUITECTONICOS MODELO VISTA CONTROLADOR EN EL FRONTEND

HÉCTOR MARTÍN DE LOS RÍOS SÁIZ

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería de Computadores

Madrid, 22 de junio de 2015

Director: José Luis Vázquez-Poletti

Autorización de Difusión

HÉCTOR MARTÍN DE LOS RÍOS SÁIZ

Madrid, 22 de junio de 2015

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "MODELO DE ADECUACIÓN Y ARQUITECTURA DE APLICACIONES UTILIZANDO PATRONES ARQUITECTONICOS MODELO VISTA CONTROLADOR EN EL FRONTEND", realizado durante el curso académico 2014-2015 bajo la dirección de Jose Luis Vazquez-Poletti en el Departamento de Arquitectura de Computadores, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

Con el avance de los navegadores modernos, la posibilidad de construir aplicaciones web completas en JavaScript se ha vuelto cada vez más popular, y con ello, las aplicaciones en el cliente no dejan de aumentar su complejidad. Cada vez más y más lógica termina siendo ejecutada en el navegador, por lo que escribir un código reusable y fácil de mantener es crucial en esta nueva era de la web. Para resolver este problema, los desarrolladores se han ido trasladando a diferentes propuestas MVC que prometen incrementar la productividad y la facilidad de mantenimiento del código.

Este proyecto surge dentro del framework Dashboard de la sección "Infraestructura de Monitorización" del grupo "Soporte a la Computación Distribuida" del departamento IT del CERN. En él, se encuentran multitud de interfaces de usuario altamente interactivas, las cuales están enormemente basadas en JavaScript, y se quiere evaluar las distintas tecnologías actuales para ver si es posible obtener un beneficio en términos de reducir la carga de trabajo en el desarrollo y soporte de las aplicaciones.

Uno de los objetivos de este trabajo es establecer unas métricas lo más atemporales posibles que permitan diferenciarlas y valorarlas, y que ayuden a determinar y elegir la más adecuada acorde a las necesidades de un proyecto.

Asimismo, se demuestra el beneficio de estas soluciones para aplicaciones web JavaScript altamente escalables examinando el diseño y la arquitectura de la aplicación "Site Availability Monitoring" (SAM) dentro del Experimento Dashboard del CERN, mostrando el desarrollo de su arquitectura con una de estas tecnologías previamente seleccionada con el modelo propuesto y midiendo los resultados obtenidos al haber trabajado con la solución correcta.

Palabras clave

Arquitecturas, JavaScript, MVC, AngularJs, EmberJs, CERN, aplicaciones escalables.

Abstract

With the progress of modern browsers, the possibility of building web applications using only JavaScript has became extremely popular, and with it, the client-side applications are much more complex than before. Application development requires collaboration from multiple developers and writing maintainable and reusable code is crucial in the new web app era.

Due to this issue, developers have moved to different MVC technologies in order to increase their productivity and the maintainability of their code. This project starts within the Dashboard framework at CERN, where there are many highly interactive user interfaces that are heavily based on JavaScript and with the goal of evaluating these new technologies to see if we can benefit from them in terms of reducing our workload for developing and supporting our applications.

We present timeless metrics that allow us to differentiate and rank the technologies, and help us to choose the right one for a specific project.

At the same time, we analyze the structure of the large-scale JavaScript application SAM3 and show its development with one of these technologies.

Keywords

Architectures, JavaScript, MVC, Angularjs, EmberJs, CERN, scalable applications.

Glosario

Término	Definición
AJAX	"Asynchronous JavaScript and XML". Técnica de desarrollo web para crear aplicaciones interactivas utilizada para permitir a los clientes realizar peticiones a APIs asincronamente una vez que la página ha cargado.
API	"Application Programming Interface". Conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
CDN	"Content Delivery Network". Red de ordenadores que sirven contenido con una alta disponibilidad y rendimiento.
DOM	"Document Object Model". Representación en árbol de los objetos en una página web.
HTML	"Hypertext Markup Language". Lenguaje de marcado para la elaboración de páginas web.
HTTP	"Hypertext Transfer Protocol". Protocolo utilizado en cada transacción de la World Wide Web.
JSON	"JavaScript Object Notation". Formato ligero para el intercambio de datos.
MySQL	Sistema de gestión de bases de datos relacional, multihilo y multiusuario.
MVC	"Modelo Vista Controlador". Patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones.
REST	"REpresentational State Transfer". Estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.
SOAP	"Simple Object Access Protocol" Protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.
SPA	"Single-Page Application". Aplicación web que inserta todo el contenido en una única página.

URL	"Uniform Resource Locator" Cadena de caracteres que designa recursos de una red.

Índice de contenidos

Índice de figuras

Índice de tablas

CAPÍTULO 1: INTRODUCCIÓN

1.1. Motivación.

El desarrollo web ha evolucionado drásticamente durante la última década. La demanda de aplicaciones de alta calidad se ha incrementado y al mismo tiempo, diferentes patrones y arquitecturas han emergido motivadas por el incremento de la complejidad de estos servicios.

Con el objetivo de reducir el ancho de banda utilizado por los usuarios, y de desacoplar los datos de un único sitio web, están aumentando el número de APIs JSON que son expuestas con el simple propósito de proveer datos, en lugar de enviar toda una página preparada para ser interpretada en el navegador. Como consecuencia directa de ello, el número de clientes JavaScript en aplicaciones o navegadores comunicándose con APIs RESTful no para de crecer.

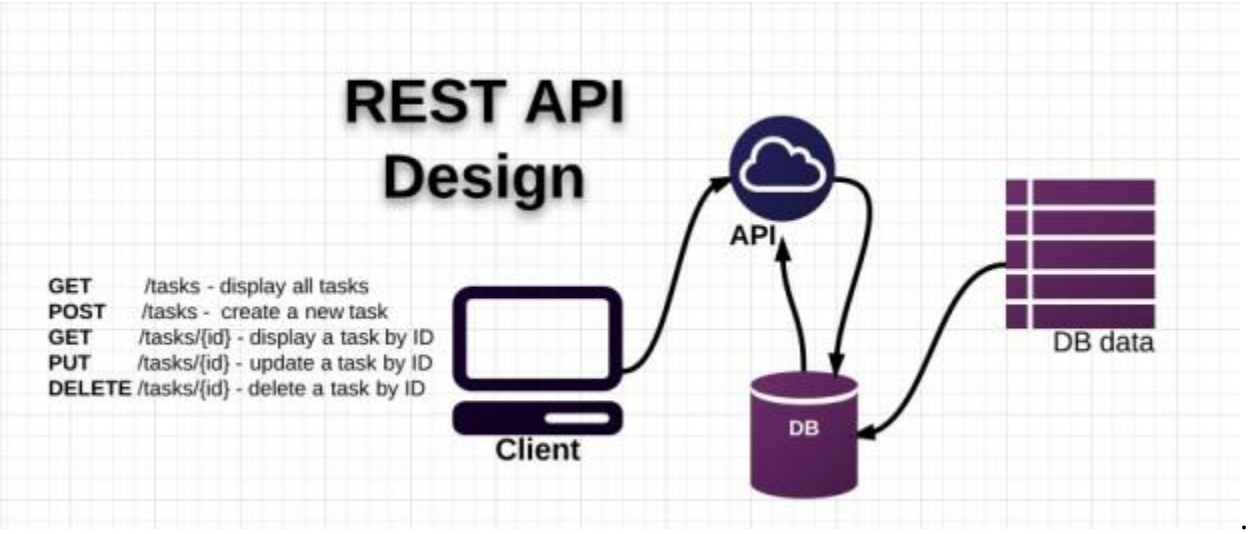

Figura 1.1 RESTful API

Con el avance de los navegadores modernos, la posibilidad de construir aplicaciones web completas en JavaScript se ha vuelto cada vez más popular, y con ello, las aplicaciones en el cliente no dejan de aumentar su complejidad. Cada vez más y más lógica termina siendo

ejecutada en el navegador, por lo que escribir un código reusable y fácil de mantener es crucial en esta nueva era de la web. Para resolver este problema, los desarrolladores se han ido trasladando a diferentes propuestas MVC que prometen incrementar la productividad y la facilidad de mantenimiento del código. Sin embargo, a la hora de elegir una de estas propuestas para estructurar y organizar nuestras aplicaciones, la lista de nuevas y estables soluciones no para de crecer.

Figura 1.2 Algunas de las tecnologías MVC más populares para el cliente del 2015.

¿Hay demasiadas opciones? Parece que sí. El número de soluciones estables no para de crecer. Y los desarrolladores se encuentran abrumados ante tal variedad de oferta.

En la vida hay que enfrentarse a infinidad de decisiones en las que se han de valorar múltiples opciones y escenarios, pero normalmente es posible establecer unas restricciones fácilmente. Por ejemplo, si surgiese la decisión de determinar que coche debería comprar un sujeto en particular, podría establecer una restricción de presupuesto, y/o una de eficiencia.

Sin embargo, a la hora de escoger una de estas tecnologías, establecer restricciones no es tan sencillo: todas son gratuitas, todas tienen una documentación muy bien escrita, y a simple vista,

podría parecer que todas van a cumplir la función necesaria, y que todas podrían ser igualmente validas para nuestro proyecto.

Este proyecto surge dentro del framework Dashboard [1] de la sección "Infraestructura de Monitorización" del grupo "Soporte a la Computación Distribuida" del departamento IT del CERN [2]. En él, se encuentran multitud de interfaces de usuario altamente interactivas, las cuales están enormemente basadas en JavaScript, y se quiere evaluar las distintas tecnologías actuales para ver si es posible obtener un beneficio en términos de reducir la carga de trabajo en el desarrollo y soporte de las aplicaciones.

Uno de los objetivos de este trabajo es establecer unas métricas lo más atemporales posibles que permitan diferenciarlas y valorarlas, y que ayuden a determinar y elegir la más adecuada acorde a las necesidades de un proyecto. El objetivo es conseguir maximizar el beneficio que puede proporcionar y nunca conformarse con la primera solución que se encuentre y que cubra las necesidades momentáneas.

1.2. Público objetivo.

Este trabajo está enfocado a una audiencia que ya se encuentre familiarizada con JavaScript y el patrón MVC. Sin embargo, también se aspira a proveer de información valiosa a desarrolladores que estén empezando a familiarizarse con JavaScript y que se encuentren interesados en explorar tanto la arquitectura como las posibilidades de las nuevas tecnologías MVC.

1.3. Antecedentes.

Vivimos en un mundo cada vez mas informatizado. Tan sólo en el último año, productos como Pebble [3] (un reloj programable) y Raspberry Pi [4] (una placa de ordenador del tamaño de una tarjeta de crédito) aparecieron en el mercado. Pebble recolectó más de 10.266.845 USD en el popular sitio web de "crowdfunding" Kickstarter [5], y entre las empresas inversoras se

encuentra Y-Combinator [6], la aceleradora de start-ups famosa por invertir en empresas enormemente exitosas como Dropbox, Scribd, Disqus y Airbnb entre otras.
Raspberry Pi no para de incrementar su popularidad, y recientemente ha ganado el respaldo de Google, quien está entregando más de 15.000 microordenadores a estudiantes alrededor del Reino Unido. Considerando que Code.org apenas acaba de hacer su aparición en línea hace un par de años, respaldado por personajes de la talla del actual Presidente de los Estados Unidos Barack Obama, el ex presidente de la misma nación, Bill Clinton, y el fundador de Facebook, Mark Zuckerberg, asegurando que "cualquier estudiante de cualquier escuela debe tener la oportunidad de aprender a programar", la programación nunca ha tenido más relevancia que en estos momentos.

Cuando los primeros ordenadores se introdujeron al mercado, interactuar con ellos sin saber cómo programar o poseer un avanzado conocimiento de los lenguajes de ciertos dominios era prácticamente imposible. Con el tiempo, los ordenadores se han vuelto más fáciles de utilizar, y en el proceso, el código se ha escondido prácticamente en su totalidad al usuario común. La mayoría de ordenadores personales vienen equipados con navegadores web capaces de procesar JavaScript. A diferencia de lenguajes como Java o C, que utilizan una sintaxis bastante compleja, JavaScript ofrece una sintaxis más sencilla, lo que disminuye en gran medida la curva de aprendizaje para los nuevos usuarios.
Durante muchos años, JavaScript fue mayoritariamente relegado a pequeños scripts que realizaban las funciones más sencillas de una página. Era lento, y los desarrolladores web tenían que recurrir a flash para realizar tareas más complejas. Hacer clic en una etiqueta web ("tab"), como una imagen, requería una completa actualización de la página en cuestión, y el código de JavaScript tendía a ser en general repetitivo y poco orientado a objetos.

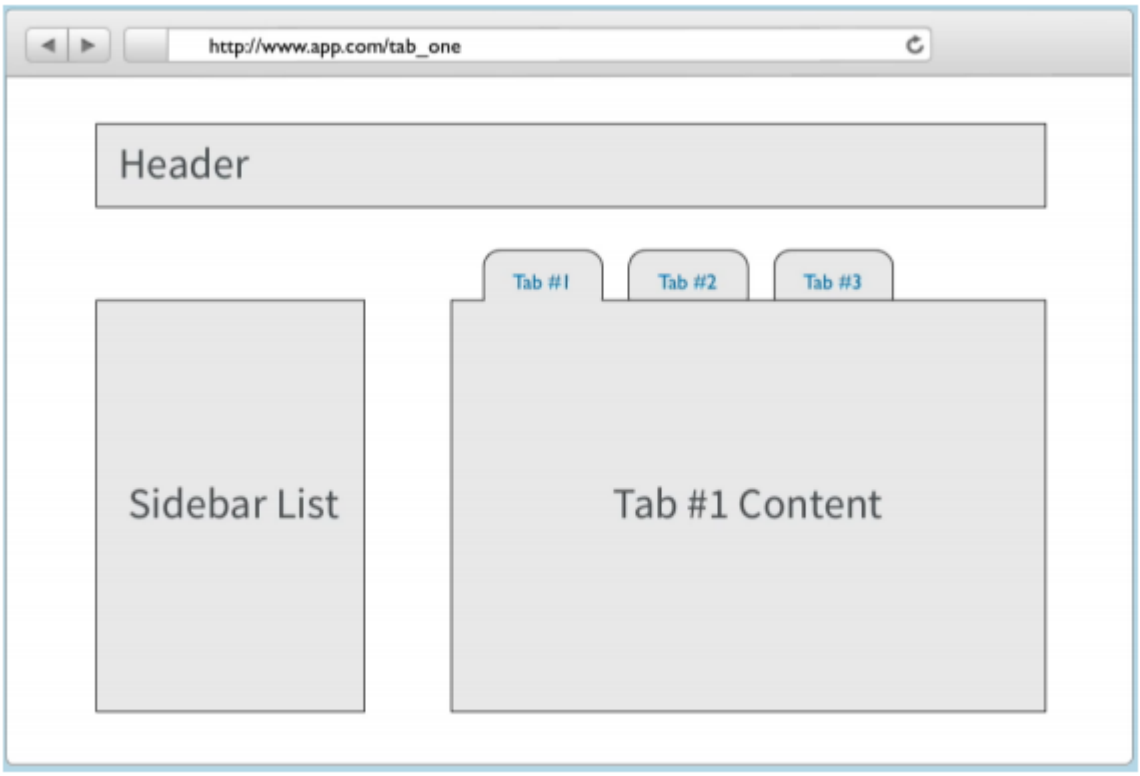

Figura 1.3 Ventana con etiquetas (“tabs”) en una aplicación web.

Indudablemente, un gran punto de inflexión fue la popularización de AJAX, la cual vino de la mano del lanzamiento de Gmail en el 2004. Con Gmail, Google demostró que se podía ofrecer una buena experiencia al usuario sin refrescar la página utilizando para ello únicamente JavaScript.

El lanzamiento del framework de JavaScript, jQuery, en Agosto del 2006, trajo consigo una especie de renacimiento para el lenguaje. Librerías como jQuery son famosas por “haber ayudado a abstraer inconsistencias entre navegadores y proveer una API de alto nivel para realizar consultas a AJAX y cambios del DOM”. En el ejemplo anterior, la popularización de jQuery significaba que las pestañas podían trabajar sin tener que refrescar la pagina, en algunos casos por medio de llamadas AJAX y en otros simplemente escondiendo y mostrando partes de la misma página.

Cabe recalcar que en los tiempos en los que jQuery fue introducido, las aplicaciones web rara vez actualizaban el estado de la pestaña activa al usuario (lo que ahora ocurre con “hashes” y “push-states”), por lo que la pestaña activa se perdía en cuanto la pagina era refrescada y al

usuario no le era posible navegar hacia adelante o hacia atrás entre pestañas con los botones estándares del navegador.
Un buen ejemplo es el carrito de la compra en una aplicación de comercio electrónico. Antes de la introducción de Ajax, añadir elementos al carrito requería refrescar la pagina, pero con el uso de jQuery, de llamadas a Ajax y de respuestas a estas llamadas, los desarrolladores fueron capaces tanto de tener una información persistente como de mantener la interfaz del usuario sincronizada. El resultado de esto era a menudo una base de código que ni estaba bien estructurada ni era fácil de mantener.

Según el sitio web oficial de jQuery, "jQuery es una librería JavaScript rápida, pequeña, y repleta de funciones. Hace que funciones con documentos HTML tales como manipular, animar, manejar eventos y AJAX, sean mucho más sencillas, y con una API fácil de utilizar entre multitud de navegadores compatibles. Con una combinación de versatilidad y extensibilidad, jQuery ha cambiado la manera de utilizar JavaScript para millones de personas".

Con la ayuda de jQuery normalmente se puede hacer mucho más con mucho menos código:
JQuery:

```
$('#container');
```

JavaScript:

```
var container = document.getElementById('container');
```

Figura 1.4 Google Trends de JavaScript y jQuery.

Para representar la importancia de jQuery, el grafico de arriba ilustra el hecho de que en los últimos años la búsqueda del término “jQuery” se ha vuelto igual de popular que la búsqueda del término “JavaScript”.

Estadísticas similares [7] muestra la popular web de preguntas y respuestas sobre desarrollo “Stack Overflow” que demuestran cómo el incremento en el interés sobre jQuery y JavaScript van de la mano.
JavaScript es así mismo, de lejos, el lenguaje más popular en proyectos alojados en Github [8].

Las bases de datos modernas que no se encuentran basadas en SQL, como por ejemplo MongoDB [9], también utilizan JavaScript para algunas funciones como por ejemplo, la reducción de imagen. MongoDB tiene incluso un “escudo” basado en este lenguaje. Por otro lado, como dato anecdótico, el lenguaje de programación Clojure posee un compilador llamado ClojureScript, cuyo objetivo es parecido al de JavaScript. Con la introducción de Node.js en el año 2009, el cual facilita tanto la programación en JavaScript de aplicaciones escalables como de

servidores web, así como con el apoyo de compañías tan grandes como Microsoft y Linkedin, JavaScript goza de una popularidad inmensa. El experto en Backbone.js, Brian Mann, lo resume con las siguientes frases: “JavaScript ha venido para quedarse”, “JavaScript se ha convertido en la piedra angular del desarrollo web moderno”.

1.4. Aplicaciones del lado del cliente.

Con la creciente popularidad de JavaScript y el aumento de las aplicaciones del lado del cliente, las aplicaciones web modernas han aumentado su complejidad hasta tal punto, que todo el núcleo lógico termina hecho completamente en el lado del cliente y la obtención de datos y el almacenamiento se realizan en segundo plano utilizando para ello AJAX. Páginas como Pandora.com o Rdio.com son ejemplos conocidos de lo que se denomina aplicaciones de página única (SPA, siglas en inglés), dado que recogen todos los datos sin necesidad de recargar la página y procesan toda la lógica en el lado del cliente. El beneficio de hacer esto es la velocidad, ya que el cliente no necesita solicitar grandes volúmenes de información del servidor, ni costosas interacciones con la base de datos. La mayor parte de la información se carga cuando lo hace la página inicial y el resto se transmite asincrónicamente en formato JSON, el cual requiere poco procesamiento del servidor y da lugar a un menor número de peticiones HTTP.

El proceso de una interacción normal entre el cliente y un servidor web consiste en los siguientes pasos [10]:

1. El usuario escribe o hace clic en un enlace en su cliente (en la mayoría de los casos un navegador web).
2. El navegador realiza una solicitud HTTP al servidor que contiene una cabecera y un cuerpo.
3. El servidor procesa la solicitud de acuerdo a lo que se pide y a los parámetros potenciales.
4. El servidor manipula o crea información en la base de datos en el caso de una página dinámica.

5. El servidor envía una respuesta HTTP con una Cabecera (por ejemplo: 200 OK) y un Cuerpo (por ejemplo: el modelo modificado) que contiene información en un formato que puede entender el cliente tales como HTML o JSON.
6. El navegador recibe la respuesta HTTP.
7. El navegador genera la respuesta que visualiza el usuario.
8. Esta repuesta puede en muchas ocasiones realizar más solicitudes HTTP al servidor.

Si el usuario hace clic en otro enlace se repite el ciclo.

En algunos casos el proceso anterior puede ser evitado porque todos los datos ya están en el lado del cliente. En vez de enviar una solicitud HTTP y cargar una nueva página cuando el usuario hace clic en un enlace, los datos obtenidos originalmente simplemente tienen que ser representados de una manera diferente. En lugar de que el servidor entregue páginas HTML que contengan imágenes, hojas de estilos y referencias a scripts que dan inicio a una nueva solicitud HTTP, toda la interacción se realiza usando JSON, y la única información que se entrega es aquella que cambia en el modelo.

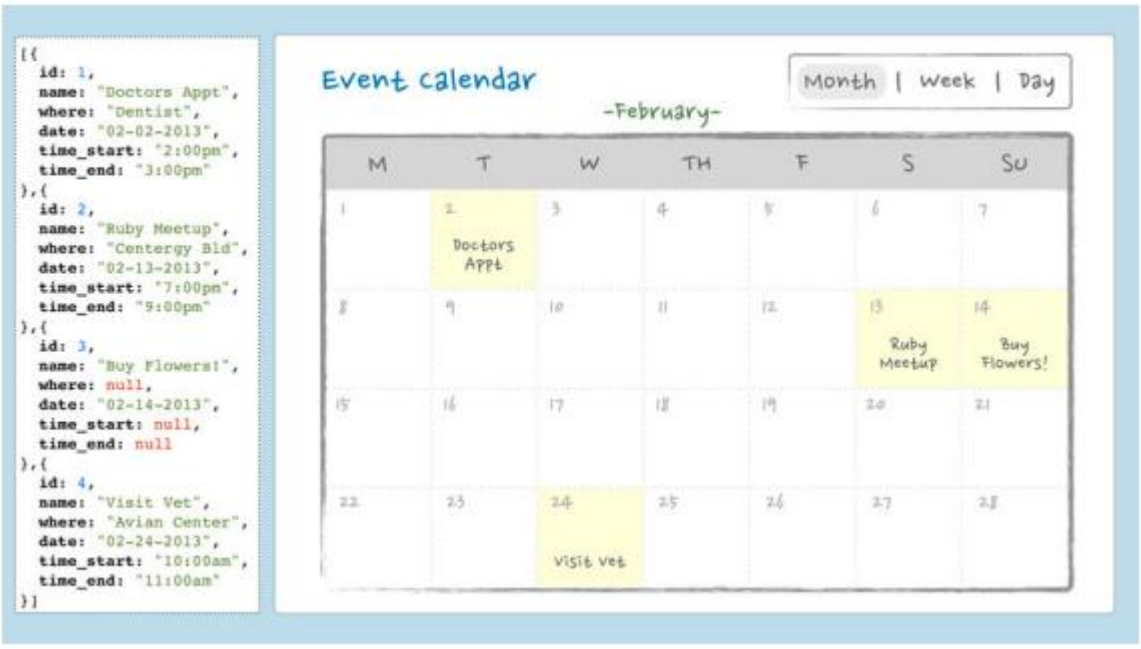

Figura 1.5 Calendario mostrando los eventos por mes.

Tómese por ejemplo una aplicación calendario, que en su forma tradicional, al cambiar la visualización de mes a semana requeriría la recarga de la página o una solicitud al servidor por AJAX que obtenga la vista de la nueva región. Como podemos ver en la Figura 1.5, una aplicación del lado del cliente almacena toda la información necesaria para mostrar el calendario con diferentes granularidades con los datos que ya existen en el cliente (puede ser incluso en un almacenamiento local HTML5), por lo tanto, todo lo que se necesita modificar es la plantilla que muestra la información. Si los datos de un modelo cambian, la vista automáticamente reflejará esto por medio de las plantillas.

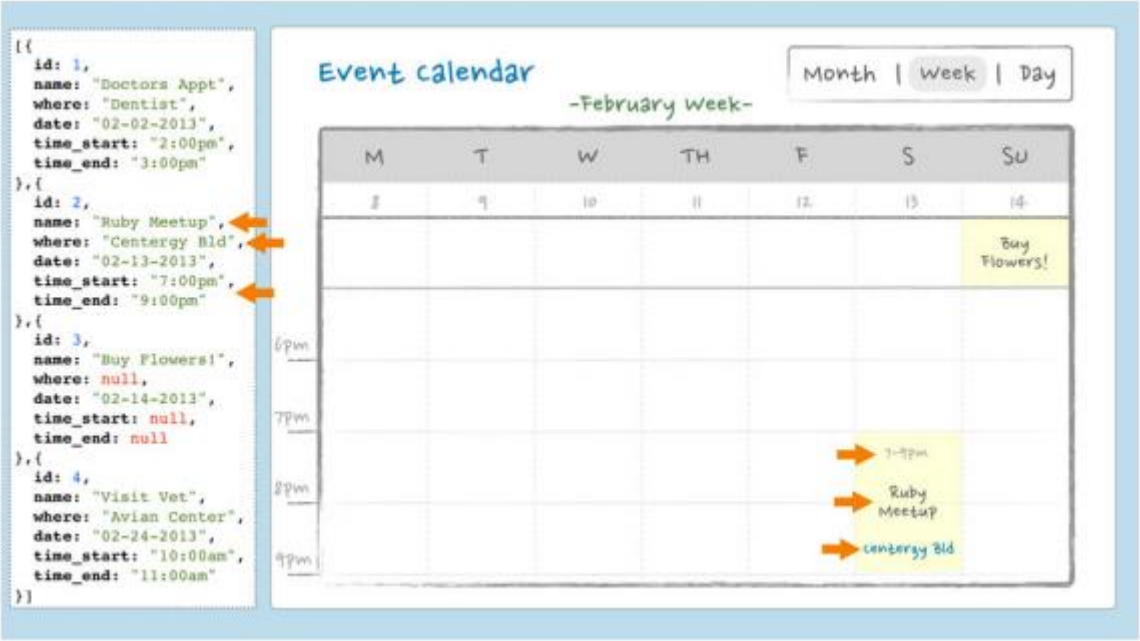

Figura 1.6 Calendario mostrando los eventos por semana.

Es computacionalmente caro y una mala práctica crear código HTML con JavaScript usando la concatenación de cadenas, que es el motivo por el cual se introdujeron las plantillas JavaScript. Las variables en las plantillas son definidas por el uso de una sintaxis específica, como: {{ejemplo}}, en Handlebars.js y fijadas por la inserción de código JSON en la plantilla.

Las aplicaciones del lado del cliente también permiten a los desarrolladores implementar fácilmente versiones sin conexión (offline) de sitios web mediante el uso de tecnologías web modernas como el almacenamiento local, algo que adquiere una importancia tremenda en

nuestro uso siempre creciente de la información en plataformas móviles. Tener un servidor en segundo plano que envía código JSON abre muchas posibilidades para la fácil integración con aplicaciones móviles. Las solicitudes JSON necesitan muchos menos datos para ser transferidos, lo que se traduce en grandes beneficios, sobre todo cuando se trabaja sobre el ancho de banda, a menudo limitado, de las redes móviles.

JSON es descrito de la siguiente manera en json.org:

> Notación de Objeto JavaScript (JavaScript Object Notation) o JSON, es un formato ligero de intercambio de datos. Es sencillo para los humanos leer y escribir en este formato. Es sencillo para las máquinas analizarlo y generarlo. Está basado en un subconjunto del estándar ECMA-262 3ra Edición de diciembre de 1993 del Lenguaje de Programación JavaScript. JSON es un formato de texto independiente del lenguaje de programación usado pero que utiliza elementos que son familiares para los programadores de los lenguajes de la familia C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Esto hace que JSON sea un lenguaje ideal para el intercambio de datos.

El uso de JSON ha crecido enormemente con la popularización del uso de las APIs, y es el modelo estándar de almacenamiento que usan muchos de los servidores de bases de datos NoSQL (que no usan lenguaje de preguntas estructurado), como por ejemplo, CouchDB y Riak. Contrario al almacenamiento tabular de datos que utiliza SQL, JSON hace fácil el almacenamiento de datos utilizando un anidamiento profundo, algo así como hijos de hijos.

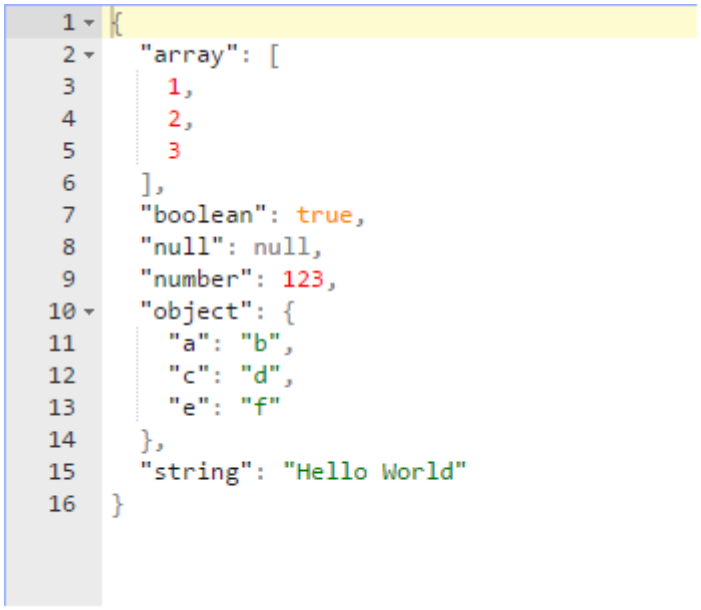

Figura 1.7 Ejemplo de JSON propuesto por https://www.jsoneditoronline.org/

Una de las mayores desventajas de las aplicaciones del lado del cliente y páginas únicas es la difícil indexación de sus contenidos por los motores de búsqueda, a menos que se degrade útilmente hasta una aplicación normal. Tradicionalmente, en el desarrollo web se buscaba realizar la degradación útil de la funcionalidad de JavaScript, de manera que aquellos dispositivos que tuviesen deshabilitado JavaScript no estuvieran en desventaja, pero según un estudio realizado por Yahoo en 2010, el número de dispositivos con JavaScript deshabilitado está alrededor del 1.3%. Por esa razón uno debería preguntarse si la degradación útil aún persiste como un requerimiento para la accesibilidad [11].

1.5. Modelo – Vista – Controlador.

Modelo – vista – controlador (MVC) es una arquitectura de software diseñada por Trygve Reenskaug mientras trabajaba en 1979 en Smalltalk-80, pero sólo ganó popularidad después de ser descrito en profundidad en **Patrones de Diseño: Elementos reusables de software orientado a objetos** en 1994. MVC divide las partes de una aplicación en tres tipos de elementos con el objetivo de crear grandes proyectos manejables a través de la abstracción y para crear una estructura unificada entre diferentes proyectos. Esto permite a una aplicación JavaScript

permanecer manejable y escalable, y previene de tener códigos base llenos de un sinnúmero de funciones de respuesta automática (“callbacks”) de AJAX. Separando las vistas y los modelos también se simplifica la creación de unidades de pruebas [12].

MVC no se aplica solamente a JavaScript, sino que puede ser encontrado en muchas de las tecnologías más utilizadas para el desarrollo web, como por ejemplo Djando o Ruby on Rails.

Los tres componentes de la arquitectura MVC tradicional son los siguientes:

- **Controlador:**

  Procesa y responde a los eventos y maneja el modelo y la vista. Es la lógica de programación de la aplicación.

- **Modelo:**

  Los Modelos contienen la representación de los datos que son usados por la aplicación. Los modelos notifican a las Vistas cuando su estado cambia de tal manera que estas puedan adecuarse a los datos del mismo.

- **Vistas:**

  Las vistas presentan los datos del Modelo de manera que puedan ser adecuados para la interacción con el usuario. Por ejemplo el código HTML en una aplicación web. En un MVC, la lógica de programación de la aplicación debe separarse el máximo posible de la Vista.

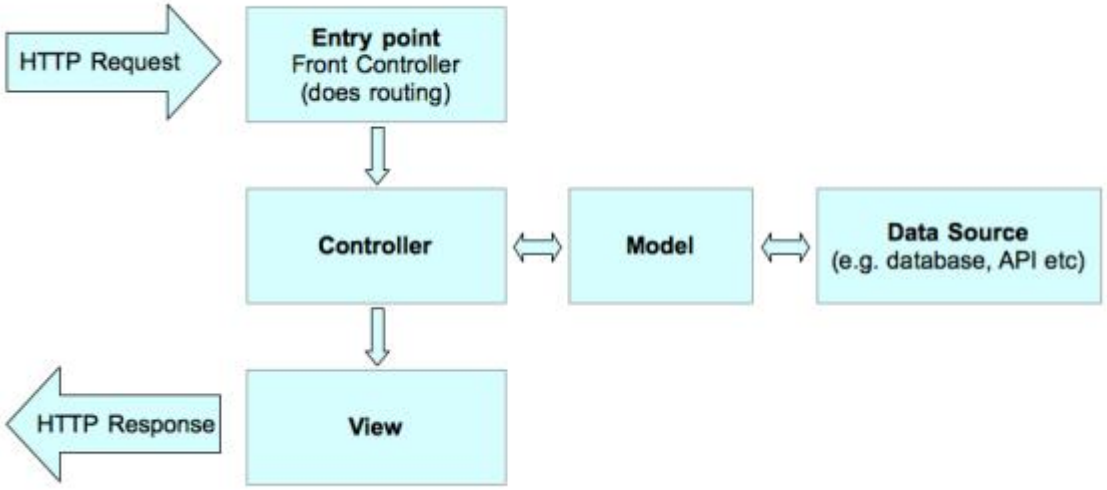

Figura 1.8 Ciclo de vida de la solicitud/respuesta HTTP para la arquitectura MVC del lado del servidor.

Aunque en cada proyecto la implementación se ajusta para dar respuesta a necesidades específicas, las siguientes características pueden ser vistas como un estándar:

- **Modelo:**

- Validación de atributos.
- Preserva los datos del modelo en una base de datos o en el almacenamiento local del navegador.
- Son monitorizados por las vistas para reflejar los cambios de los modelos de cara al usuario.
- A menudo son agrupados en colecciones de tal modo que la parte lógica pueda ser aplicada a diversos modelos al mismo tiempo.

- **Vista:**

- Muestra una interfaz al usuario.
- Adecua y presenta el contenido del modelo (o colección).
- Se actualiza cuando el modelo cambia.

- Gestiona las plantillas, por medio del uso de bibliotecas como Handlebars.js, Eco (Embedded CoffeeScript) o EJS (Embedded JavaScript).

Plantilla EJS:

```
<table>
  <tr>
    <th>Titulo</th>
    <th>Creado</th>
  </tr>
  <% articulos.each(function(model) { %>
    <tr>
      <td><%= model.escape(titulo) %></td>
      <td><%= model.escape('creado_por') %></td>
    </tr>
  <% }); %>
</table>
```

Plantilla Handlebars.js:

```
<table>
  <tr>
    <th>Titulo</th>
    <th>Creado</th>
  </tr>
  {{#each articulo}}
    <tr>
      <td>{{titulo}}</td>
      <td>{{creado_por}}</td>
    </tr>
  {{/each}}
</table>
```

- **Controlador:**

- Maneja los cambios en la vista y actualiza el modelo.
- Los controladores son los componentes de MVC que tienen más diversidad y es muy difícil describir una funcionalidad común a todos ellos.

En una aplicación JavaScript de página única, en la que se utiliza MVC en el cliente y en el servidor, las vistas generadas por el **servidor** MVC no tienen una gran función, pero proveen un contenedor para ser llenado por medio de JavaScript (cliente) MVC (*Figura 1.7*). Los modelos JavaScript (cliente) deben almacenar los datos acorde a la lógica del lado del servidor, mientras

que las vistas deben actualizarse automáticamente para representar cualquier cambio de datos. Los Controladores deciden qué modelo deber ser usado y que vista será visualizada. El servidor y el cliente interactúan entre sí normalmente a través de JSON sobre interfaces REST (Transferencia de Representación de Estado) o SOAP (Protocolo Simple de Acceso a Objeto).

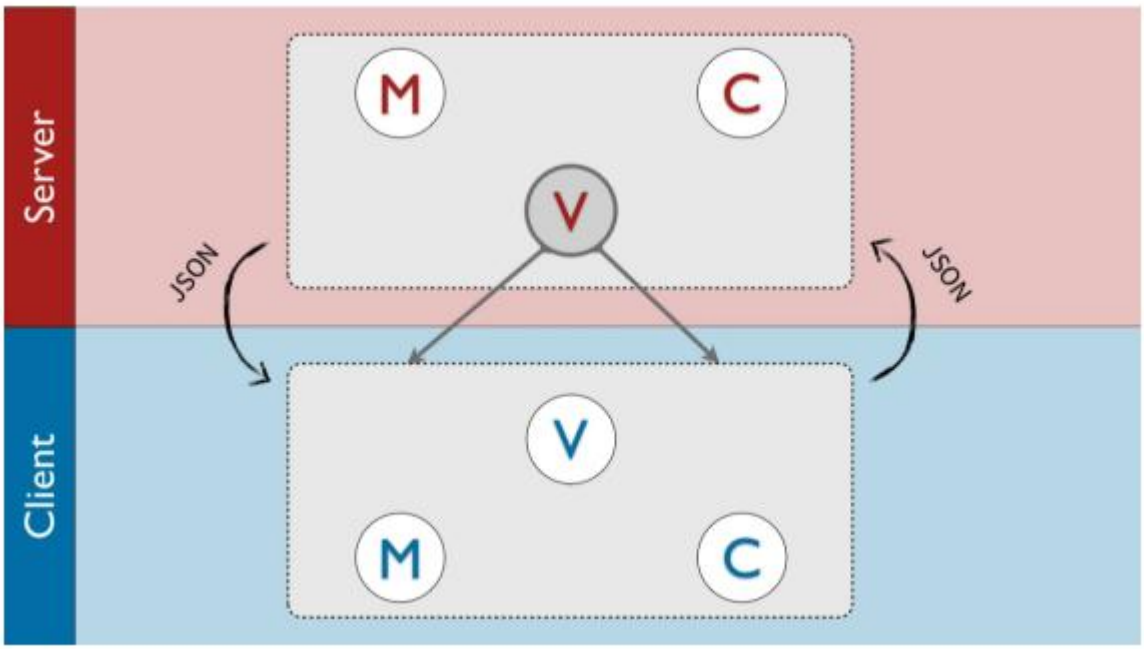

Figura 1.9 Modelo de configuración de una aplicación de página única.

El que REST haya remplazado a SOAP puede ser atribuido a su estructura más simple y legible y que REST usa los métodos estándares de HTTP como GET, POST, DELETE y PUT. En la Figura 1.8 se muestra un ejemplo de una aplicación CRUD (Create – Read – Remove – Delete) (Alta-Baja-Modificar-Consulta) implementada en Ruby on Rails.

```
      articles GET    (/:locale)/articles(.:format)            articles#index {:locale=>/en|de/}
               POST   (/:locale)/articles(.:format)            articles#create {:locale=>/en|de/}
   new_article GET    (/:locale)/articles/new(.:format)        articles#new {:locale=>/en|de/}
  edit_article GET    (/:locale)/articles/:id/edit(.:format)   articles#edit {:locale=>/en|de/}
       article GET    (/:locale)/articles/:id(.:format)        articles#show {:locale=>/en|de/}
               PUT    (/:locale)/articles/:id(.:format)        articles#update {:locale=>/en|de/}
               DELETE (/:locale)/articles/:id(.:format)        articles#destroy {:locale=>/en|de/}
```

Figura 1.10 Implementación de una aplicación Alta-Baja-Modificar-Consulta en Ruby on Rails.

Previo al antes mencionado renacimiento de JavaScript existieron muchas buenas prácticas, recomendaciones y frameworks para estructurar el código del lado del servidor (código que se ejecuta en el servidor) pero poco o nada para organizar el código del lado del cliente (código que se ejecuta en el cliente, por ejemplo un navegador web). Aunque jQuery es de mucha utilidad para el desarrollo moderno de sitios web, al mismo tiempo carece de una estructura para organizar el código. Grandes aplicaciones del lado del cliente a las que les falta una estructura organizativa modular y desacoplada frecuentemente devienen en lo que se conoce popularmente bajo el término "código espagueti":

> "El **código espagueti** es un término peyorativo para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible. Su nombre deriva del hecho de que este tipo de código parece asemejarse a un plato de espaguetis, es decir, un montón de hilos intrincados y anudados.
> Tradicionalmente suele asociarse este estilo de programación con lenguajes básicos y antiguos, donde el flujo se controlaba mediante sentencias de control muy primitivas como goto y utilizando números de línea." – *Wikipedia, 20 de Enero del 2015.*

A raíz de este problema, surgen multitud de tecnologías JavaScript MVC con el objetivo de dotar de cierto orden y estructura a grandes aplicaciones JavaScript. Mientras que muchos de estos proyectos se centran en el patrón Modelo – Vista – Controlador, existen otros (por ejemplo KnockoutJS) que incorporan una estructura diferente como Modelo – Vista – Presentador o Modelo – Vista – VistaModelo. Algunos le asignan la responsabilidad del Controlador a la Vista (por ejemplo Backbone.js) mientras que otros añaden sus propios componentes en una mezcla que ellos entienden es más efectiva. Por ejemplo, Backbone.js, técnicamente hablando no es MVC, lo cual reconocen renombrando sus "Controladores" como "Encaminadores". Dado que algunos proyectos JavaScript MVC difieren mucho en su estructura de lo que tradicionalmente se considera MVC, ellos se refieren con frecuencia a los mismos usando el patrón MV* debido a que la mayoría posee al menos un Modelo y una Vista.

> "Los frameworks y bibliotecas JavaScript modernas pueden traer estructura y organización a tus proyectos, estableciendo un fundamento sostenible justo desde el comienzo." – *Addy Osmani, autor de Developing Backbone.js applications.*

Los proyectos JavaScript MVC también ayudan a mantener libre el código HTML de almacenar muchos datos en los atributos "rel" o "data-*", algo que puede considerarse como muy bueno porque le permite a la vista quedarse con la capa de presentación sin insertar lógica de la aplicación o datos, y permite hacer el código fuente modular y más fácil de mantener.

1.6. Ámbito de la contribución.

En este documento se presenta un análisis de las soluciones actuales para resolver la complejidad de las aplicaciones en el cliente y se propone un modelo de evaluación y comparación de las mismas con el objetivo de encontrar de una manera rápida la que mejor se adapte a un proyecto en particular. Asimismo, se demuestra el beneficio de estas soluciones para aplicaciones web JavaScript altamente escalables examinando el diseño y la arquitectura de la aplicación "Site Availability Monitoring" (SAM) dentro del Experimento Dashboard del CERN del grupo "Soporte a la Computación Distribuida", mostrando el desarrollo de su arquitectura con una de estas tecnologías previamente seleccionada con el modelo propuesto y midiendo los resultados obtenidos al haber trabajado con la solución correcta.

CAPÍTULO 2: MODELO DE ADECUACIÓN

2.1. Introducción.

Como ya se ha comentado anteriormente, la oferta de soluciones MVC estables disponibles para su ejecución en el cliente no para de crecer. Tomar una decisión a la hora de empezar un proyecto no es nada sencillo.

Como consecuencia del exceso de información sin un formato homogéneo, y de la publicidad que se recibe de las múltiples funcionalidades de estas tecnologías, en numerosas ocasiones, los desarrolladores se encuentran perdidos y terminan decidiéndose por la primera solución que encuentran por ellos mismos y que parece satisfacer las necesidades momentáneas, pasando por alto algunas características tan importantes como la escalabilidad o la comunidad que da soporte al código, y que pueden llevar a un proyecto al completo fracaso, o en su mejor caso, a una ardua tarea de mantenimiento y muchas horas de trabajo innecesarias.

Por lo tanto, parece muy importante definir unas métricas que permitan tomar una decisión que maximice el rendimiento en base a unos requisitos de proyecto.

Es de gran importancia además, proveer unas métricas que puedan extenderse en un futuro, en lugar de realizar únicamente una comparación, ya que estas tecnologías están en constante desarrollo y pueden variar rápidamente dejando de ser válidas algunas premisas.

Por lo tanto, cada vez que se inicie un nuevo proyecto, será necesario evaluar de nuevo todas las opciones, partiendo de estas métricas como base, y si fuera necesario, añadir las nuevas métricas que correspondan.

Este trabajo aspira, de esta manera, a proveer de un **modelo de adecuación** lo más **atemporal** posible.

Por último, el peso de las métricas, y el hecho de que puedan excluir alguna de las tecnologías, dependerán en gran medida de los requisitos del proyecto a los que es aplicado el modelo, como

se verá en el apartado 3.4 cuando se aplique en el ámbito del acelerador de partículas de Suiza (el CERN).

2.2. Definición de métricas de evaluación generales.

2.2.1. Métrica 1. Popularidad.

La primera métrica que se debe observar es la popularidad. Un buen indicador de la buena salud de un proyecto de código libre es la cantidad de usuarios que utilizan la tecnología, la dimensión del grupo de desarrolladores que hay detrás, y la comunidad de soporte que se encuentra.

Una gran comunidad significa un mayor número de respuestas a preguntas, más módulos de librerías externas, más tutoriales de Youtube, etc.

No se querrá tomar una decisión basada únicamente en este dato, pero ciertamente da una primera impresión de qué tecnologías están siendo utilizadas actualmente en el mercado.

Para medirla, se utilizarán estadísticas como la tendencia en Google Trends, el número de seguidores en Stack Overflow, el número de usuarios utilizando el repositorio de Github, el número de preguntas en Stack Overflow, el número de resultados en videos de Youtube, o la cantidad de usuarios utilizando la extensión de Google Chrome o de Mozilla Firefox.

2.2.2. Métrica 2. Curva de aprendizaje.

Una curva de aprendizaje describe el grado de éxito obtenido durante el aprendizaje en el transcurso del tiempo. Es un diagrama en que el eje horizontal representa el tiempo transcurrido y el eje vertical el número de éxitos alcanzados en ese tiempo.

A menudo se cometen muchos errores al comenzar una nueva tarea. En las fases posteriores disminuyen los errores, pero también las materias nuevas aprendidas, hasta llegar a una llanura.

Dependiendo del proyecto y de la situación, podrá permitirse un aprendizaje más lento, si ello se traduce en unos importantes beneficios futuros, y para ello, será necesario estudiar la curva de aprendizaje de cada tecnología.

Para medirla, se estudia la documentación y se intenta desarrollar un pequeño proyecto de prueba con cada una las tecnologías más prometedoras que se hayan encontrado hasta el momento. Por ejemplo, un lector de noticias de Google News comunicándose con la API RESTful en formato JSON y con un parámetro para filtrar un rango temporal.

La curva de aprendizaje puede variar con rapidez en proyectos poco maduros o poco estables, a la vez que cambia su API, su documentación y sus guías de toma de contacto.

2.2.3. Métrica 3. Tamaño.

Es muy importante entender como de grande es el conjunto de librerías que conforman las tecnologías y qué es exactamente lo que se está consiguiendo a cambio de un tamaño extra en la aplicación. El tamaño afecta al rendimiento y al tiempo de carga de la aplicación, lo cual es crucial para el éxito del sitio web, pero también es un indicador de cuán ambiciosa es una tecnología y de cuánto tiempo y esfuerzo puede llevar dominarla, así como dar una pista para entender de qué manera está tratando de ayudar al desarrollador a construir una aplicación (por ejemplo cuántas características soporta y cómo de robustas son). Cuanto más ambiciosa y llena de características esté, normalmente, será más difícil de integrar con otras tecnologías en el mismo sitio o aplicación. Las tecnologías más ligeras suelen ser tipo librerías, siendo más fáciles de incluir en proyectos ya existentes.

Para medir estos tamaños, se acude directamente a los datos de la página web oficial de la tecnología ó se inspecciona el tamaño del archivo descargado. Las librerías se encuentran “minificadas” y comprimidas, por lo que se comparará el tamaño de las versiones en este estado.

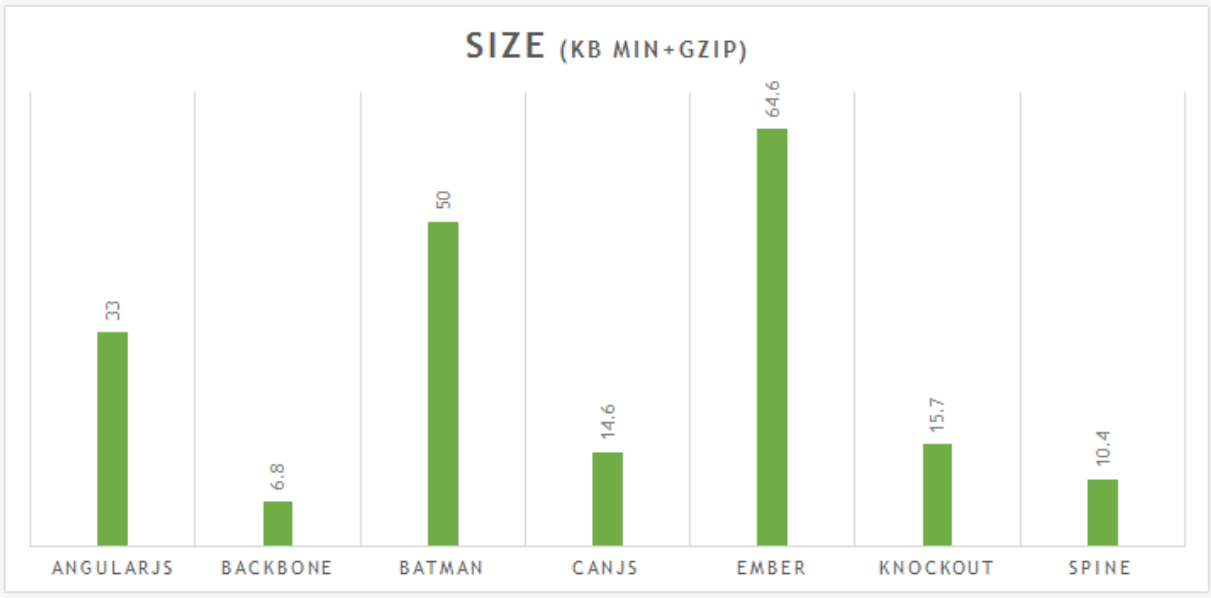

Figura 2.1 Tamaño de algunas de las propuestas evaluadas en el momento de la redacción de este trabajo.

Algunos de estos proyectos como Backbone y Spine se enorgullecen de cuán pequeños son y se consideran a ellos mismos más como una librería. A menudo, estos pequeños proyectos dejan al desarrollador la decisión de escoger otras librerías para completar sus características, como por ejemplo para las plantillas de visualización o el enrutamiento. Hablaremos de esto con más detalle cuando analicemos las características de cada uno.

Otros proyectos en cambio, como Emberjs o AngularJs son más ambiciosos y se encuentran cómodos siendo llamados frameworks. Normalmente tienen más características nativas y dependen mucho menos de librerías externas.

A continuación, se presenta una lista de ejemplo mostrando que proyectos se consideran más una librería frente a cuales se consideran más un framework:

Librerías	frameworks
Backbone	**Ember**
Knockout	**AngularJs**
Spine	**Batman**
CanJs	**Meteor**

Tabla 2.1 Librería contra framework.

2.2.4. Métrica 4. Dependencias.

¿Qué otras librerías se necesitan para construir una aplicación en el mundo real con estos proyectos?
El tamaño y número de dependencias es un factor a tener en cuenta.
En la práctica, la mayoría de los proyectos actuales van a utilizar jQuery junto con estas tecnologías para realizar manipulaciones del DOM en la aplicación web, y al mismo tiempo, para dar soporte a las animaciones y a AJAX. En las aplicaciones para móviles es común utilizar Zepto.js, la cual es una librería mucho más ligera para manipular el DOM pero no soporta Internet Explorer, lo que no suele ser un problema en las aplicaciones móviles.

Se ha recopilado una pequeña tabla con estos datos, los cuales pueden necesitar ser actualizados para el próximo proyecto. Se muestra una columna para móviles en la que se asume que se utiliza Zepto.js y una para aplicaciones web en la que se utiliza jQuery.

AngularJs ya incorpora una pequeña versión de jQuery, jQuery lite, que puede ser sobrescrita por jQuery si se incorpora completamente. El equipo de AngularJS anima a los desarrolladores a no incluirlo a no ser que sea estrictamente necesario.

APLICACION WEB	
	TAMAÑO (kb min+gzip)
AngularJS	**33**
angular.js	33
Backbone	**44.8**
underscore.js	5.4
backbone.js	6.8
jquery.js	32.6
Batman	**50**
batman.js	50
CanJS	**47.2**
canjs.js	14.6
jquery.js	32.6
Ember	**111.4**
ember.js	64.6
handlebars.js	14.2
jquery.js	32.6
Knockout	**58.7**
sammy.js	6.6
knockout.js	15.7
knockout.mapping.js	3.8
jquery.js	32.6
Spine	**48.4**
underscore.js	5.4
spine.js	10.4
jquery.js	32.6

APLICACION MOVIL	
	TAMAÑO (kb min+gzip)
AngularJS	**33**
angular.js	33
Backbone	**22.4**
underscore.js	5.4
zepto.js	10.2
backbone.js	6.8
Batman	**50**
batman.js	50
CanJS	**24.8**
zepto.js	10.2
canjs.js	14.6
Ember	**78.8**
ember.js	64.6
handlebars.js	14.2
Knockout	**36.3**
sammy.js	6.6
knockout.js	15.7
knockout.mapping.js	3.8
zepto.js	10.2
Spine	**26**
underscore.js	5.4
zepto.js	10.2
spine.js	10.4

Figura 2.2 Tamaño de las dependencias de algunas de las propuestas evaluadas en el momento de la redacción de este trabajo.

Para medirlo, se analizan las dependencias y su tamaño que se pueden encontrar en cdnjs [13] por ejemplo.

2.2.5. Métrica 5. Interoperabilidad.

Esta métrica trata de analizar si cada tecnología está diseñada para controlar la página completa y si se puede utilizar como una pequeña parte de una página ya existente al mismo tiempo que se trabaja lentamente en la incorporación en todo el proyecto. Es decir, la dificultad de integrarla en un proyecto ya existente.

El debate anterior entre librería y framework determina por lo general cuán interoperable es cada uno de estos proyectos. Mientras que las librerías son más fáciles de integrar en proyectos ya existentes, los frameworks aportan más valor al desarrollador pero no trabajan tan bien en equipo.

2.2.6. Métrica 6. Madurez.

Considerar la madurez de cada proyecto ayuda a entender cuán grande es el riego que se está corriendo al usar estas nuevas tecnologías. Los frameworks nuevos y sin probar pueden tener problemas con la documentación, escalabilidad, estabilidad (cambios en la API) y el soporte (encontrar desarrolladores que conozcan el framework para mantener el código), lo cual podría convertir una buena decisión, en todo lo contrario.

Para medirlo, debemos hacernos las siguientes preguntas:

¿Cuántas aplicaciones del mundo real usan estas tecnologías en su desarrollo y cuántos usuarios tienen estas aplicaciones? ¿Cuán buena es la documentación y cuantos ejemplos/tutoriales están disponibles? ¿Están actualizados los ejemplos? ¿Cuán estable es la API? ¿Hay otros desarrolladores que conocen o que están empezando a familiarizarse con esta tecnología?

En el momento de redactar este trabajo se intenta dar una respuesta actualizada a las preguntas anteriores para las tecnologías más populares del momento. Hay que volver a realizar el proceso cada vez que se plantee empezar un proyecto nuevo con una tecnología diferente.

Backbone (el más maduro).

- Aplicaciones utilizándolo durante más de tres años que incluyen por ejemplo a: GroupOn, FourSquare, USAToday y DocumentCloud.
- Buena documentación.
- Buenos ejemplos, aunque muchos están desactualizados.
- Una API muy estable.

- Muchos observadores en GitHub.

AngularJS (maduro).

- Google utilizándolo.
- Buena documentación, cada día mejor.
- Un gran número de ejemplos.
- Muchos observadores en GitHub.

Knockout (maduro).

- En uso desde hace al menos 2 años.
- Buena documentación incluyendo bastantes ejemplos del tipo "Jsfiddle".
- Una API estable.
- Un gran número de ejemplos.
- Muchos observadores en GitHub.

Ember.js (maduro).

- Primera versión estable (v1.0) el 30 de agosto de 2013 tras dos años de desarrollo.
- Documentación mejorando aunque la API ha estado intencionadamente inestable hasta la liberación de la versión 1.0 por lo que existe un buen número de ejemplos pero algunos están desactualizados debido a estos cambios en la API.
- Muchos observadores en GitHub.

Meteor.

- Aun se encuentra en una etapa inicial de su desarrollo, se usa mayormente en aplicaciones de ejemplo.
- Buena documentación pero en progreso.
- API evolucionando.
- Algunos ejemplos.

- Muchos observadores en GitHub.

CanJS.

- Lleva apenas dos años pero fue extraído de un framework con seis años de uso.
- Aplicaciones en uso para Walmart, Cars.com, Apple (store.apple.com), Sams Club mobile app, Wanderfly.

2.2.7. Métrica 7. Liderazgo, inspiración y filosofía.

El entender a las personas, su trasfondo y los problemas que estaban tratando de solucionar cuando crearon una determinada tecnología, nos ayuda a valorar más sus decisiones y motivaciones en cuanto al diseño. Por ejemplo, David Heinemeier Hanson, creador del popular framework web Ruby on Rails, trabajaba como desarrollador contratado en la realización de proyectos de diseño para 37signals y solo disponía de 10 horas a la semana para trabajar en dicho framework. De hecho, Ruby on Rails se obtuvo de uno de sus primeros trabajos de contrato con 37signals. Este trasfondo te ayuda a comprender que el framework debía ser en gran manera productivo para el desarrollador, lo que quiere decir, muchas convenciones (decisiones ya tomadas) y estructura. A continuación, se conocerá a los creadores de algunos de los proyectos MVC JavaScript más populares en el momento de realización de este trabajo para que de manera similar se pueda llegar a apreciar su trabajo.

Para medirlo: La tarea consiste en investigar sobre los diferentes líderes de cada una de las tecnologías en consideración.

A continuación se muestra un pequeño resumen de algunos de ellos.

Jeremy Ashkenas (Backbone)

Figura 2.3 Jeremy Ashkenas.

Jeremy Ashkenas es el creador del lenguaje de programación CoffeeScript, del framework JavaScript Backbone.js y de la biblioteca de utilidades JavaScript Underscore.js.

Miško Hevery y Adam Abrons (AngularJS)

Figura 2.4 Miško Hevery y Adam Abrons.

AngularJS fue desarrollado originalmente en Google en el 2009 por Miško Hevery y Adam Abrons como el software detrás de un servicio en línea de almacenamiento JSON. Abrons dejó el proyecto, pero Hevery, que trabaja en Google, continúa su desarrollo y mantiene la biblioteca junto a sus colegas de Google Igor Minár y Vojta Jína.

Steve Sanderson (Knockout).

Figura 2.5 Steve Sanderson.

Steve Sanderson es el creador original de Knockout. Él trabaja actualmente como desarrollador para Microsoft en el equipo que nos proporciona la tecnología ASP.NET, e IIS entre otras. Anteriormente, desarrolló el software .NET en calidad de contratista/asesor para clientes en Bristol y además escribió algunos libros para Apress como Pro ASP.NET MVC framework.

Yehuda Kats y Tom Dale (EmberJs).

Figura 2.6 Yehuda Kats y Tom Dale.

Yehuda Katz es miembro de los equipos Ember.js, Ruby on Rails y jQuery Core. Además, es coautor de los libros de gran éxito editorial *jQuery in Action y Rails 3 in Action*.

Tom Dale estuvo previamente en el equipo SproutCore. Él es un ex-ingeniero de software de Apple que ganó experiencia en el desarrollo de interfaces (front-end) JavaScript mientras trabajaba en las aplicaciones MobileMe e iCloud.

Distintos desarrolladores (Meteor).

El grupo de desarrollo de Meteor acaba de acumular $11.2 millones por lo que pueden trabajar a tiempo completo y tienen un equipo de 12 desarrolladores con currículos impresionantes. El equipo tiene metas ambiciosas que van más allá del alcance de la mayoría de los frameworks MVC JavaScript que se enfocan en organizar el código del lado del cliente. Meteor es un framework completo que además incluye una arquitectura en el servidor web y en la base de datos.

Equipi Bitovi (CanJs).

CanJS fue creado hace apenas 2 años por Justin Mayer y su equipo en Bitovi (una empresa web de asesoramiento sobre aplicaciones). CanJS se extrajo del framework original de la empresa JavaScriptMVC que para la fecha en que se escribe este trabajo ya lleva siete años de existencia. El negocio fundamental de Bitovi es crear aplicaciones con el framework CanJS.

Una de las preguntas favoritas de los periodistas al entrevistar a un músico es: "¿Qué artistas escuchabas según crecías o quién fue tu inspiración?" Esto a menudo conduce a comprender o da pistas a sus lectores de qué sonido pueden esperar de ese músico. Muchas de las ideas en estas tecnologías no son nuevas en el desarrollo de software sino que vienen de otras en las que sus creadores trabajaron en el pasado y lo disfrutaron.
Esta métrica, está en gran parte relacionada con la anterior, y la información se puede obtener de entrevistas con los creadores de estos proyectos acerca de dónde encontraron su inspiración.

A continuación se muestra un ejemplo de resumen de la información que se ha podido obtener durante la escritura de este trabajo sobre algunos de ellos (deberá ser actualizado incluyendo todas tecnologías más populares en el momento de desarrollo de nuestro próximo proyecto):

AngularJs

El trabajo de los creadores de AngularJS estuvo influenciado en gran medida por las tecnologías de los lenguajes de programación declarativos, como HTML, y las tecnologías de aplicaciones enriquecidas de Internet (RIAs) tales como Flex de Adobe, Windows Presentation Foundation

(WPF) y Silverlight de Microsoft. La vinculación de datos bidireccional de las vistas a los objetos del modelo es un ejemplo de este estilo de programación declarativo en acción. También la inyección de dependencias y los contenedores de inversión de control (IOC), en particular Juice, que es usado enormemente en los códigos Java del lado del servidor por los ingenieros de Google, constituye una inspiración declarada para los creadores, dado que ellos valoran las pruebas de unidades y necesitan un framework que esté diseñado para permitirle inyectar las dependencias de manera que las pruebas puedan ser aisladas de otras capas de la aplicación y se ejecuten muy rápido.

EmberJs

Tom Dale hizo un gran trabajo al describir en *Quora* los elementos que influyeron en el desarrollo de Ember:

«Con Ember.js, dedicamos mucho tiempo a tomar prestados conceptos introducidos por frameworks para aplicaciones nativas como Cocoa. Cuando sentimos que esos conceptos eran más un obstáculo que una ayuda – o que no concordaban con las restricciones de la web- dirigimos la mirada a otros proyectos populares de código abierto como Ruby on Rails o Backbone.js para encontrar nuestra inspiración. Por lo tanto, Ember.js, es una síntesis de poderosas herramientas de nuestros antepasados con las sensibilidades de la web moderna. – Tom Dale en Quora»

Asimismo, es importante entender que Ember.js es una evolución de la biblioteca JavaScript SproutCore y que se convirtió en Ember cuando SproutCore dejó de parecerse a Cocoa para ser mucho más influenciado por jQuery.

Knockout

Esencialmente, el patrón de diseño Modelo-Vista-Vista-Modelo (MVVM) y las tecnologías declarativas como WPF de Microsoft (Windows Presentation Foundation) y Silverlight fueron las mayores fuentes inspiración. Uno se puede haber dado cuenta de que la vinculación de datos bidireccional declarativa, que es la mejor característica de Knockout, es muy similar a AngularJS, esto es debido a que ambas tuvieron algunas influencias similares.

CanJs

Según Justin Meyer, Rails tuvo una gran influencia, en particular en cuanto al nombre y la API. La evolución del framework, en especial las características añadidas en la versión 2.0, ha sido influenciada por otros frameworks MVC JavaScript, más específicamente, la vinculación de datos bidireccionales y directivas de AngularJS (elementos habituales en CanJS).

Los periódicos generalmente se esfuerzan para lograr ser imparciales a la hora de comunicar las noticias. La única excepción es la sección Editorial donde se promueven las opiniones y los escritores a menudo asumen una posición bastante inamovible con respecto a un tema dado. En ocasiones ambos tipos de escritos no son ni completamente imparciales ni posiciones radicales, sino que se encuentran en algún punto intermedio. Los frameworks tecnológicos tienen una división similar, pues tienden a ser mucho más inflexibles o no tan inflexibles. Por ejemplo, Ruby on Rails le da más valor a la convención que a la configuración y toma muchas decisiones en nombre del desarrollador como por ejemplo la estructura de ficheros y el acceso a los datos.
Es por esto que se le considerada en gran manera inflexible. Otros frameworks del lado del servidor como Sinatra son más pequeños y flexibles sobre la estructura de los ficheros y el acceso a los datos, por lo que son considerados no tan inflexibles. Así como estos frameworks del lado del servidor tienen sus filosofías, los frameworks MVC JavaScript del lado del cliente que se han estado estudiando pueden ser examinados bajo la misma luz de inflexible a flexible. A continuación se ve cada uno de estos proyectos y se analizan sus filosofías.

Backbone: flexible

Es el proyecto de menta más abierta, extremadamente flexible, permitiendo así a los desarrolladores tomar sus propias decisiones, algunas veces al punto de que las cosas son hechas lo suficientemente distintas como para que el código sea menos sostenible. La única excepción a esta postura es la forma en la que Backbone asume el servicio REST en el servidor, lo cual se analiza más adelante en la sección de características. Esta atribución puede ser obviada sobrescribiendo el método de sincronismo en el modelo.

AngularJS: inflexible

Bastante inflexible, en particular su énfasis en la comprobación y la inyección de dependencias. También, la idea de que la programación declarativa como HTML, es impresionante, impregna este framework.

Ember: inflexible

Persigue la meta de que el desarrollador tome solo decisiones acerca de lo que es diferente para su aplicación y tome el control del resto mediante las convenciones y la estructura. Esta filosofía es similar a la de Ruby on Rails y jQuery y en el sitio web de emberjs.com se expresa como en ningún otro lugar:

«No pierdas tiempo tomando decisiones triviales. Ember.js incorpora idiomas comunes de manera que puedas enfocarte en lo que hace a tu aplicación especial y no en reinventar la rueda.»

Ember estandariza las estructuras de ficheros y URLs pero te permite sobrescribir estos elementos si es necesario. Por lo tanto, tendrás que tomar menos decisiones insignificantes porque el framework ya ha escogido valores razonables por defecto y podrás concentrarte en construir las partes que hacen a tu aplicación única.

Knockout: flexible

Deja el enrutamiento y el almacenamiento de datos a la elección del desarrollador. No dicta la estructura de ficheros o URL. Incluso permite la sustitución de las plantillas basadas en declaraciones DOM por plantillas basadas en cadenas.

2.3. Definición de métricas de evaluación técnicas.

Pensemos en estas diversas tecnologías MVC JavaScript como un conjunto de características comunes que ayudan a los desarrolladores a construir aplicaciones de página única. Es crucial entender la manera en que cada proyecto implementa estas características o escoge no implementarlas y en lugar de ello propone una librería complementaria que puede ser insertada para completar la tecnología.

2.3.1. Métrica 8. Vinculación de datos.

Esta es la característica más divulgada de estos proyectos. Al cambiar los datos en una página HTML, el objeto JavaScript enlazado con esa página es inmediatamente actualizado, al igual que cualquier otro elemento de la interfaz que esté vinculado a esa misma propiedad. En muchos de los proyectos, funciona también a la inversa, si cambias el objeto JavaScript el código HTML se actualizará automáticamente. Es la vinculación de datos bidireccional en la web lo que siempre se ha experimentado en proyectos de aplicaciones clientes enriquecidas como Flex, Windows Forms o Windows Presentation Foundation (WPF).

En el momento de escribir este trabajo, AngularJs y EmberJs poseen una vinculación de datos bidireccional, al igual que casi todos los proyectos. Aunque no es así en todos, algunos podrían argumentar que Backbone y Spine tienen cierto soporte para la vinculación de datos pero estas le dejan tanto trabajo por hacer al desarrollador que se cree que se puede afirmar que no es una característica de esas bibliotecas.

2.3.2. Métrica 9. Enrutamiento.

Mapear las diferentes rutas URL a funciones JavaScript permite el uso del botón Atrás en los navegadores. Una de las principales desventajas de las aplicaciones de página única es que como la página no se recarga, no se añaden entradas al historial del navegador y por lo tanto el botón

Atrás con frecuencia no lleva al usuario al estado anterior de la página sin que el desarrollador tenga que hacer un trabajo extra durante esos cambios de estado e implementar un mecanismo de rastreo de estados a través del uso de un código "hash" añadido a la URL o usando las funcionalidades de "push" y "pop" de estados en los navegadores modernos. En resumen, la mayoría de los proyectos proveen una funcionalidad muy básica y rudimentaria aunque extremadamente útil en esta área. Knockout permite simplemente que insertes otra biblioteca de código abierto.

Una excepción en el enrutamiento, al parecer lo constituye Ember, el cual en un punto dado durante su proyecto escuchó a su comunidad e implemento esta característica justo antes de estabilizar su versión 1.0.0.

CanJS también tiene un enrutador más elaborado que hace más que enviar las diferentes rutas a las funciones y puede mantener "estados" más elaborados en una aplicación.

En el momento de escribir este trabajo, AngularJs no provee esta característica por defecto, pero tiene un proyecto independiente llamado Angular-route para ello.

2.3.3. Métrica 10. Plantillas de visualización.

Los datos del modelo JavaScript del lado del cliente necesitan ser combinados con el HTML y estos proyectos toman uno de los siguientes dos caminos para resolver este problema:

Las **plantillas basadas en cadenas**, de las cuales actualmente la más popular es handlebars.js, toman una cadena o plantilla de texto y remplazan la parte dinámica con la información del modelo. Una de las ventajas más frecuentemente referidas pero también discutibles de las plantillas basadas en cadenas es el rendimiento. La desventaja al parecer es que se dificulta el flujo de depuración de errores en la lógica de control.

Las **plantillas basadas en DOM** adoptan el uso declarativo natural de las etiquetas y crean un código HTML potenciado donde el HTML es acotado con atributos adicionales para describir los enlaces y los eventos que se necesitan. Estas bibliotecas requieren mucho menos código pero parece que hacen sustancialmente más magia de cara al desarrollador.

2.3.4. Métrica 11. Seguimiento de cambios en el modelo.

Algunos proyectos (Backbone, Spine) se centran más en el modelo y le indican al desarrollador que extienda sus clases de modelo JavaScript a partir de un tipo de modelo base y que accedan a todas las propiedades a través de los métodos ".get()" y ".set()" de modo que los cambios puedan ser rastreados y los eventos puedan ser activados cuando cambie el modelo. KnockoutJS requiere que el desarrollador aplique un encapsulamiento rastreable a sus objetos antiguos en JavaScript plano y entonces accede a las propiedades a través de la sintaxis "objeto.nombrePropiedad()".
Otras bibliotecas (AngularJS) hacen el trabajo sucio comprobando todos los vínculos de los elementos DOM en la página dado que no tienen los accesos estándar "get" y "set". Lo que nos lleva a una alerta sobre el rendimiento: el hecho de que estas bibliotecas tendrán problemas en páginas grandes. No solo estas bibliotecas requieren menos código para actualizar las plantillas, sino que no requieren tampoco el uso de métodos específicos "get" y "set" para modificar los datos, con lo que el modelo puede ser sencillamente un objeto antiguo de JavaScript plano. Esto resulta en mucha más productividad para el desarrollador, particularmente cuando se está comenzando a usar estos proyectos por primera vez.

2.3.5. Métrica 12. Almacenamiento de datos. Persistencia.

Estos proyectos almacenan datos en el servidor a través de:

- Sincronización automática mediante los servicios REST.
- Solicitud al desarrollador de que lo implemente él mismo a través de llamadas AJAX al servicio web y devolviendo JSON.
- Permitiendo cualquiera de las opciones anteriores

REST

Algunas de las tecnologías asumen por defecto que se posee un servicio extremadamente limpio de JSON REST en el servidor y que (por lo menos por defecto) te comunicas bien con ese servicio, actualizando datos de forma asíncrona en segundo plano mientras la interfaz de usuario

continúa respondiendo al usuario. Internamente estos frameworks usan jQuery o Zepto para enviar una solicitud AJAX adecuada al servidor. Así como los elementos HTML DOM de la interfaz de usuario están pendientes de los cambios del modelo en la aplicación, la implementación de la sincronización es notificada de los cambios de propiedades en el modelo y envía actualizaciones al servicio REST para mantener al modelo en sincronía con el servidor.

Conectado o desconectado

Backbone envía solicitudes por defecto antes de que los datos sean almacenados en el lado del cliente de manera que el servidor y el cliente se mantienen sincronizados con más facilidad. Spine, que es un framework muy similar a Backbone, toma un enfoque diferente, guardando los registros del lado del cliente antes de enviar una solicitud de forma asíncrona al servidor, lo que brinda una interfaz de usuario más sensible y es más tolerante a estados de desconexión, que son más frecuentes en las aplicaciones móviles. Si el proyecto requiere estados de desconexión, hay que asegurarse de conocer de qué manera la tecnología que se está usando da soporte a esa característica.

Hazlo tú mismo (DIY – Do-it-yourself)

Estos proyectos exigen al desarrollador el uso de "$.ajax" (jQuery) para hacer las llamadas al servicio web u a otra biblioteca complementaria de código libre que maneje las demandas de almacenamiento de datos.

Elementos de Almacenamiento de datos

Frameworks más elaborados, tales como Meteor, tienen implementaciones mucho más completas para el almacenamiento de datos pero requieren una base de datos MongoDB en el servidor.
Hacen esto en un esfuerzo por brindar una solución increíblemente más escalable por defecto y dar soporte el desarrollo con JavaScript de principio a fin.

CAPÍTULO 3: CASO DE USO. APLICACIÓN DEL MODELO AL GRUPO DEL CERN.

3.1. Planteamiento.

El siguiente caso de uso tiene lugar durante mi estancia en el CERN [12] y ha sido el precursor del modelo que se propone en este trabajo.

Dentro del framework Dashboard [13] de la sección Infraestructura de Monitorización del grupo de Soporte a la Computación Distribuida del departamento de IT del CERN, se encuentran multitud de interfaces de usuario altamente interactivas las cuales están enormemente basadas en JavaScript. Todas estas aplicaciones utilizan librerías de JavaScript de código abierto muy comunes, como JQuery, jQueryUI, Highcharts y Datatables entre ellas. También se está utilizando ya una tecnología JavaScript MVC de código abierto llamada Backbonejs y 2 frameworks internos: xbrowse [14] y hbrowse [15] para estructurar las aplicaciones en la parte del cliente.

Cuando se desarrollaron por primera vez los frameworks internos xbrowse y hbrowse, la única tecnología suficientemente madura era Backbonejs. En ese momento se probó Backbonejs para una aplicación [16], pero no se encontró útil para las demás, muchas de las cuales fueron desarrolladas con xbrowse [17] y hbrowse [18].

Hoy en día, sin embargo, existe un gran número de proyectos JavaScript MVC de código abierto que aportan al desarrollador una enorme funcionalidad. Por lo tanto, deberán ser evaluados y ver si es posible obtener un beneficio en términos de reducir la carga de trabajo en el desarrollo y soporte de las aplicaciones.

Para calcular el peso de las métricas utilizadas en el modelo de adecuación propuesto, es muy importante definir los requisitos previos que necesita el proyecto en cuestión.

A continuación, los **requisitos mínimos** que debería cumplir la tecnología MVC JavaScript seleccionada:

1. URL-driven MVC
   - i.e. La vista es rastreable y el historial de la URL funciona como se espera.

2. UI desacoplada del servidor
   - i.e. La UI se comunica con una API en el servidor RESTful utilizando AJAX/JSON.

3. Modular
   - i.e. El código para las distintas vistas está desacoplado y se puede trabajar en cada una de manera independiente.

4. Es sencillo integrar elementos UI populares
   - i.e. Highcharts, Datatables, etc.

5. Vistas declarativas
   - i.e. El código de la vista está definido en plantillas minimizando de esta manera el impacto de una manipulación directa del DOM utilizando JavaScript/jQuery

6. Buena documentación y soporte de la comunidad.

7. Posibilidad de hacerse cargo del control de toda la página.

Los cuatro primeros puntos de la lista los cumple el framework interno xbrowse.

Además, sabemos que las aplicaciones en las que se va a utilizar con complejas y debe ser ampliamente escalable y fácil de mantener por distintos desarrolladores ya que la estructura de trabajadores del CERN hace que cambien rápidamente y la persona encargada de mantener un proyecto no tiene porque ser la misma que la que lo desarrolló inicialmente.

3.2. Aplicación del modelo.

Elegir la tecnología correcta para nuestro proyecto puede tener un gran impacto en nuestra habilidad para desarrollar y extender las funcionalidades, así como en la facilidad de mantenimiento del código en el futuro. La web está evolucionando rápidamente y nuevas tecnologías emergen mientras que viejas metodologías rápidamente quedan obsoletas.

A la hora de elegir la tecnología que mejor se adecue al proyecto nos basaremos en el modelo de evaluación de métricas anteriormente expuesto, intentando de esta manera, maximizar los beneficios que se puedan obtener de uno de estos proyectos de código libre, y conseguir tomar una decisión en el menor tiempo posible.

Todos los proyectos que se van a tener en cuenta hoy tienen mucho en común: todos son de código libre, liberados bajo la permisiva licencia MIT, e intentan resolver el problema de crear Aplicaciones Web de una sola página utilizando el patrón arquitectónico de diseño MV*.

Los requisitos 1, 2 y 5 previamente planteados (URL-driven MVC, UI desacoplada del servidor y Vistas declarativas) los cumplen todos los proyectos JavaScript MVC modernos por lo que no vamos a entrar en más detalle sobre ellos. El requisito 6 coincide con la métrica 1 (popularidad) que examinaremos a continuación y el 7 con la métrica 5 (interoperabilidad). Y finalmente, los requisitos 3 y 4 (Modular y que sea sencillo integrar elementos UI populares) se examinarán mientras se estudia la documentación y se intenta desarrollar un pequeño proyecto de prueba para evaluar la curva de aprendizaje (Métrica 2).

3.2.1. Métrica 1. Popularidad.

Nuestra primera métrica coincide con el requisito 6.

Para medir la comunidad detrás de la tecnología, se utilizarán estadísticas como por ejemplo la tendencia en “Google Trends”, el número de seguidores en “Stack Overflow”, el número de usuarios utilizando el repositorio de “Github” o el número de preguntas en “Stack Overflow”.

Como punto de partida se toman los proyectos de código libre que se están utilizando actualmente por un número significativo de desarrolladores y que en el momento de redactar este trabajo son los siguientes:

Figura 3.1 Proyectos de código libre JavaScript MVC más utilizados.

Se quiere saber cómo de populares son actualmente, por lo que se analiza el número de seguidores que tienen en GitHub. Las graficas se han realizado con la librería HighCharts.

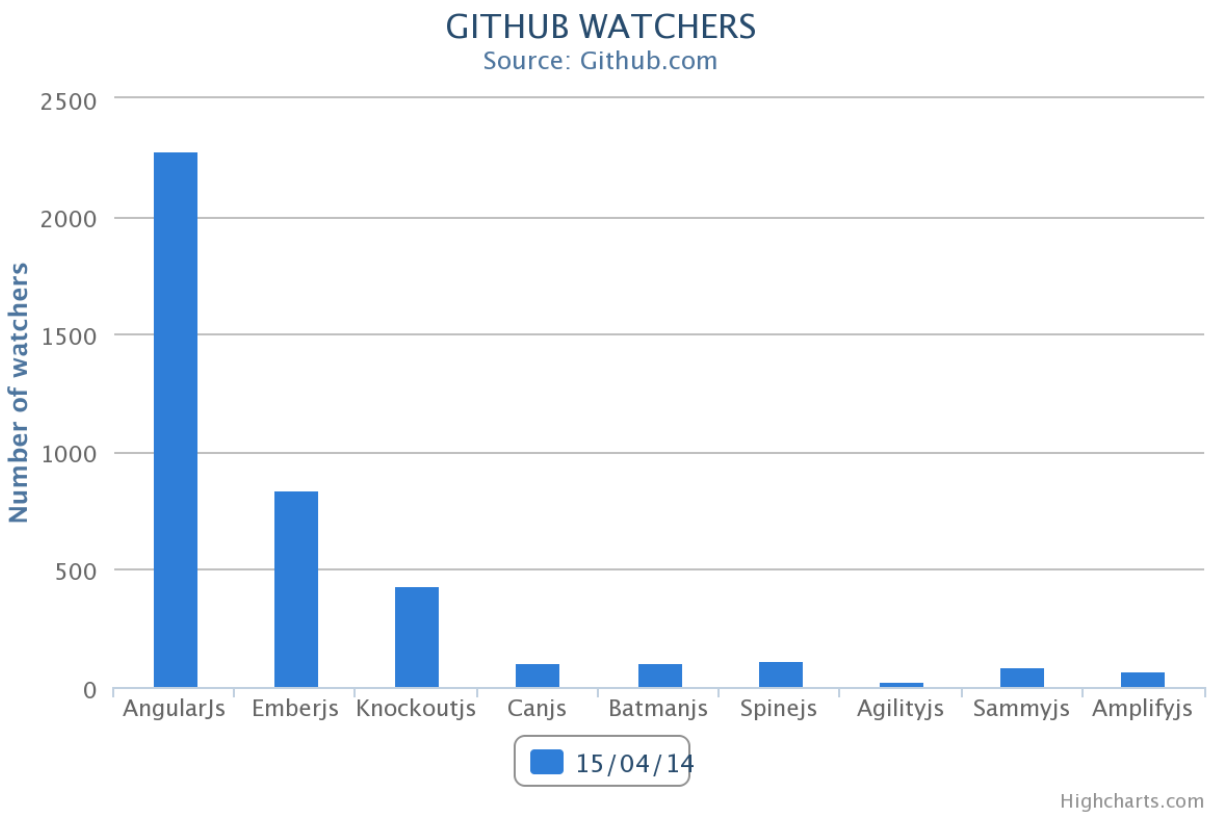

Figura 3.2 Número de seguidores en GitHub.

Los datos se puede conseguir directamente en la página de Github, y como se puede ver, en este momento, el más seguido es AngularJs, y seguidamente se tiene a EmberJs y a KnockoutJs.

A continuación se muestra lo que ocurre en la popular web de preguntas y respuestas "StackOverflow". Los datos están extraídos de la propia web y de nuevo montados con la librería HighCharts.

El número de seguidores:

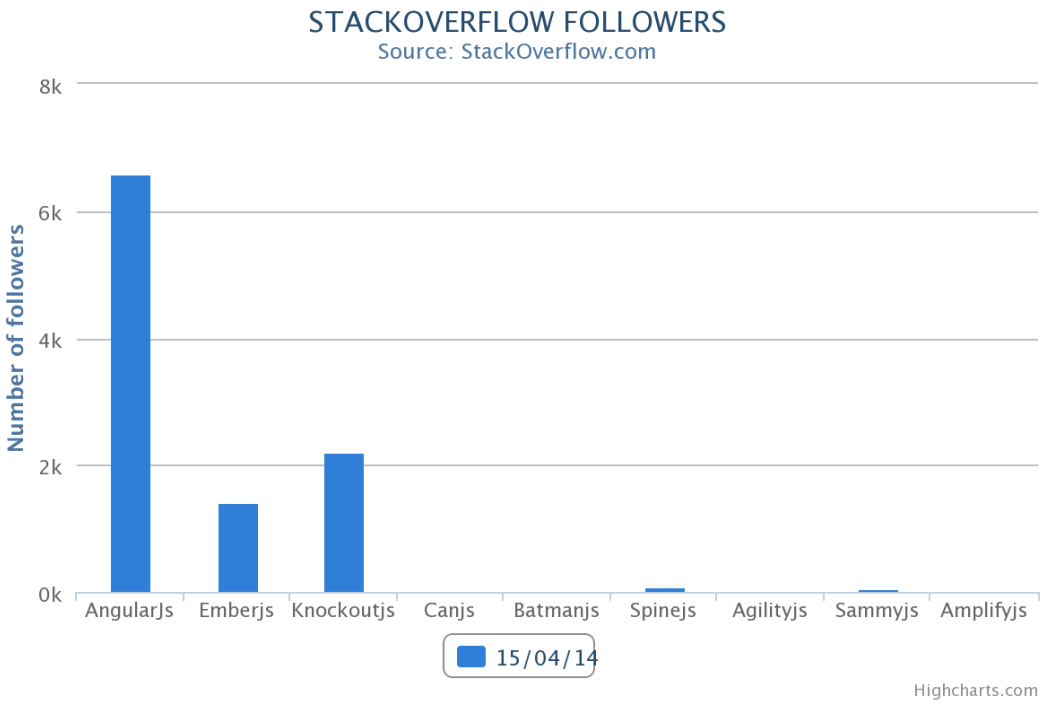

Figura 3.3 Número de seguidores en StackOverflow.

Y el número de preguntas:

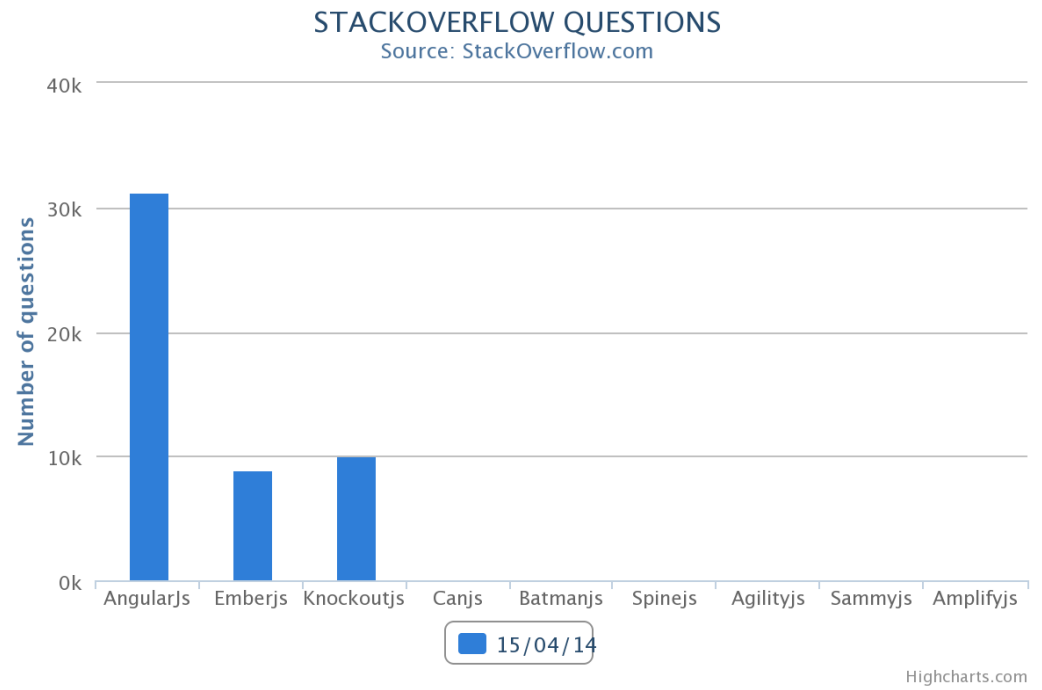

Figura 3.4 Número de preguntas en StackOverflow.

Se cree que ya se tienen datos más que suficientes por lo que no se continua, se podrían extraer más estadísticas, por ejemplo de Google Trends, o del resto de servicios de rastreo de popularidad que existen en la web, pero si el proyecto no cuenta con un número representativo de usuarios interesados en las dos webs de referencia de desarrollo de software GitHub y StackOverflow no conviene para nuestro caso, recordemos que era un requisito indispensable.

Por lo tanto, de manera muy sencilla y simplemente con la primera de las métricas, se ha limitado nuestro campo de búsqueda a tres tecnologías: AngularJs, EmberJs y KnockOut.Js
Pero aunque la popularidad sea un requisito para nuestros proyectos, no es lo único ni lo más importante, y aunque se diera el caso de que éstas tres tecnologías dieran por satisfechos los requisitos 3 y 4 (Modular y que sea sencillo integrar elementos UI populares) y pudiera parecer que cualquiera de ellas es válida para nuestro proyecto, se deberían seguir estudiando el resto de métricas hasta encontrar la que más se adecue a nuestro trabajo y la que permita maximizar su rendimiento. Así pues, se pasa a la métrica número 2.

3.2.2. Métrica 2. Curva de aprendizaje.

En este apartado, tal y como se comentaba en la definición de la métrica, se debe echar una ligero vistazo a la documentación y desarrollar una pequeña demo de cada una de las tecnologías más prometedoras que se hayan encontrado hasta el momento, en nuestro caso AngularJs, EmberJs y KnockoutJs.

Para ello, se ha decidido realizar un lector de noticias de Google News comunicándose con la API RESTful en formato JSON y con un parámetro para filtrar un rango temporal. Estas demos se encuentran anexadas a la memoria del trabajo.

Se descargan las librerías y las dependencias de cada una de las tecnologías de sus páginas web y se realizan los tutoriales propuestos por cada una de ellas. A continuación, con los conocimientos básicos adquiridos y la ayuda de la documentación, se desarrollan nuestros proyectos de prueba que se encuentran anexados a este trabajo.

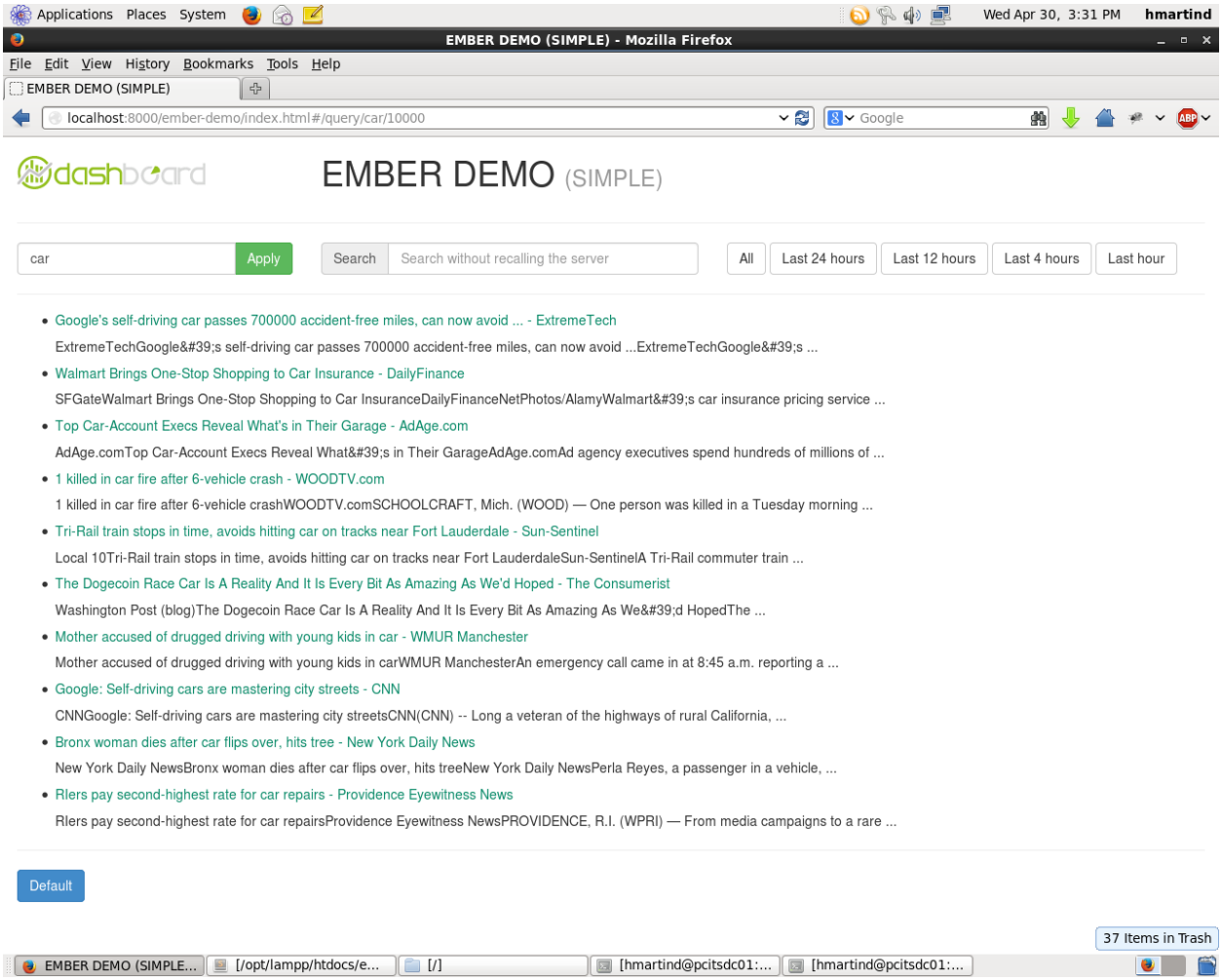

Figura 3.5 Proyecto de prueba realizado con EmberJs. Lector de noticias de Google News.

La realización del proyecto de prueba básico con AngularJs ha llevado 2 días. Mientras que con EmberJs se han necesitado 5. No se tenía ningún conocimiento previo de ninguna de las 2 tecnologías, pero sí de JavaScript y JQuery (aunque apenas ha sido necesario, únicamente para realizar las llamadas AJAX al servidor), de HTML (necesario para AngularJs ya que se basa en extender la sintaxis HTML), de CSS y del patrón arquitectónico MVC (necesario).

Aunque la documentación de EmberJs ha mejorado en gran medida desde la realización de este trabajo, el proyecto sigue siendo mucho más extenso que AngularJs y su curva de aprendizaje más empinada. Además, AngularJs también ha mejorado su documentación, poniendo un especial énfasis en la parte de toma de contacto y ofreciendo, además del tutorial de inicio, un curso gratuito junto con CodeSchool [19].

La pequeña curva de aprendizaje de AngularJs lo hace ideal para pequeños y medianos proyectos con unos requisitos muy básicos dentro del ámbito de los servicios web.

Durante el comienzo de la realización de la demo utilizando KnockOutJs y el estudio de su documentación, se ha llegado a la conclusión de que es una pequeña librería muy ligera y no el proyecto completo que se busca para que cumpla con nuestros requisitos, su tamaño de 15.7 KB y sus dependencias lo iban a advertir en la métrica correspondiente al tamaño. A KnockOutJs le falta mucha de la funcionalidad se requiere en este caso, por lo que se ha descartado.

Una de las diferencias que se han podido comprobar, es la tendencia de EmberJs a las convenciones, frente al "libre albedrio" de AngularJs.
Esta característica es muy poderosa en un entorno como el del CERN en donde el personal cambia muy a menudo y unos desarrolladores deben mantener el trabajo realizado por otros. Una clara separación de competencias te fuerza a escribir pequeños componentes donde cada uno tiene unas claras responsabilidades, lo que significa que los desarrolladores que leen el trabajo de otros desarrollares saben rápidamente donde buscar una funcionalidad especifica. Esto parece mucho más robusto y fácil de mantener, especialmente para proyectos muy grandes y escalables como en nuestro caso.

Figura 3.6 Convenciones en EmberJs. Robusto y fácil de mantener.

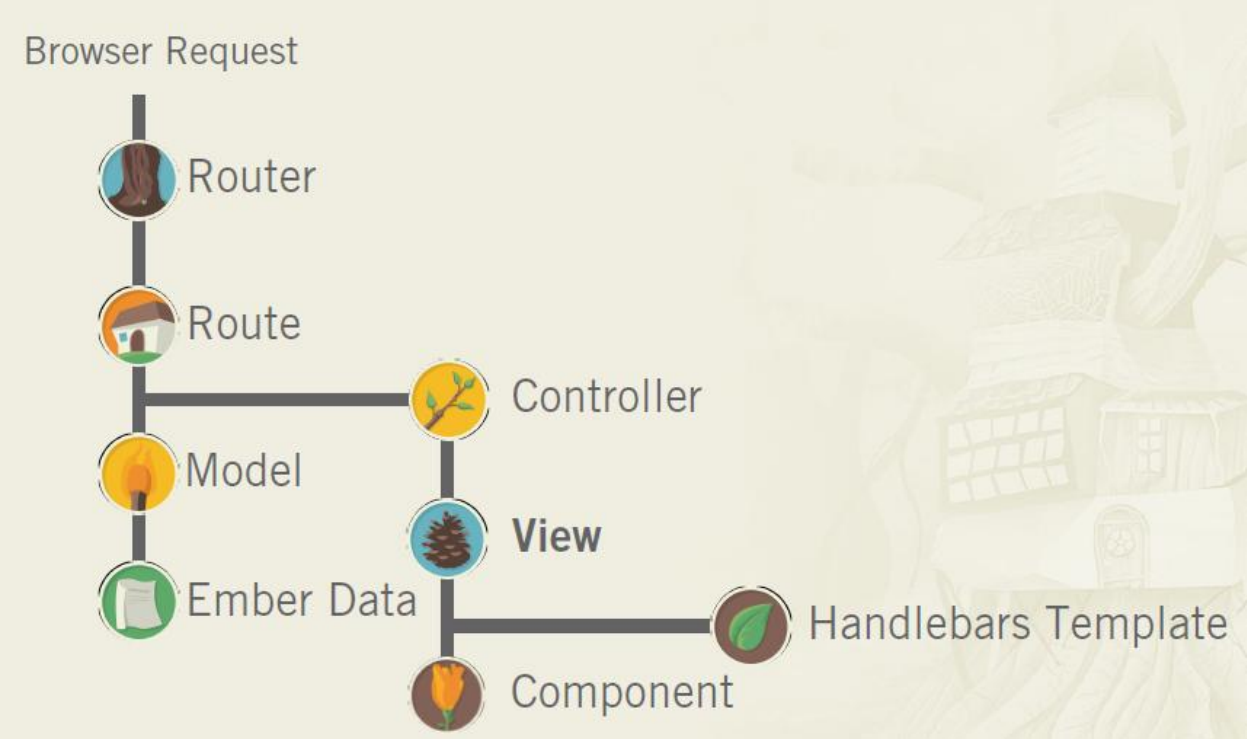

Figura 3.7 Estructura modular de EmberJs. Clara separación de competencias.

3.2.3. Métrica 3. Tamaño.

Para medir estos tamaños, se acude directamente a los datos de la página web oficial de la tecnología:

EmberJs: 64.6 KB (MIN + GZIP).
AngularJs: 33 KB (MIN+GZIP).

Además, con un rápido vistazo a la documentación se observa que la de EmberJs es bastante más amplia por lo que se puede intuir que el incremento de tamaño puede deberse a que ofrece más funcionalidades que AngularJs. Algo que en realidad ya se ha podido comprobar durante la realización del proyecto de prueba de la métrica 2.

Ambos se consideran a sí mismos "frameworks" más que "Librerías" y estamos completamente de acuerdo con esta afirmación por el tamaño de los mismos.

3.2.4. Métrica 4. Dependencias.

Para las dependencias, se han descargado las necesarias desde cdnjs.com y medido su tamaño.

EmberJs:

- Handlebarjs.js: 14.2 KB (MIN + GZIP).
- JQuery.js: 32.6 KB (MIN + GZIP).

AngularJs:

- Angular-ui-router.js 32 KB (MIN + GZIP). No es estrictamente necesaria pero se quiere para nuestro proyecto, y proporciona una funcionalidad que ya viene por defecto en EmberJs.

AngularJs es más ligera y menos dependiente de librerías externas. Sin embargo, también hay que tener en cuenta que en la práctica, casi todos los desarrolladores terminan utilizando jQuery para alguna pequeña característica e incluyendo la librería, por lo que al final el tamaño de las dependencias es similar.

3.2.5. Métrica 5. Interoperabilidad.

Como comentamos en la definición, esta métrica trata de analizar si cada tecnología está diseñada para controlar la página completa y si se puede utilizar como una parte pequeña de una página ya existente al mismo tiempo que se trabaja lentamente en la incorporación en todo el proyecto. Es decir, la dificultad de integrarla en un proyecto ya existente.

Ember.Js

Está destinado a controlar todo el sitio web durante el tiempo de ejecución, por lo que no es aconsejable utilizarlo únicamente en una parte de la página, aunque sería posible, limitando los estados de la URL en la que es utilizado, por ejemplo únicamente en "*site.com/apartadodeEmber*".

AngularJs

Muy parecido a Emberjs, es posible utilizarlo en una sola parte de la página, pero dificulta la utilización de test unitarios que fomenta la tecnología, ya que muchas librerías y "plugins" externos no han sido diseñados con la misma filosofía en mente. Para solucionarlo, se utilizan objetos "mock" [20].

Que la tecnología controle toda la página en principio no es ningún problema para nuestro caso, es más, lo que se está buscando es un proyecto grande de estas características que nos provea de mucha funcionalidad extra.

3.2.6. Métrica 6. Madurez.

Ambos proyectos son suficientemente maduros para nuestros requerimientos. Existen grandes sitios web estables que los utilizan:

Figura 3.8 Algunos proyectos que utilizan AngularJs

Figura 3.9 Algunos proyectos que utilizan EmberJs

3.2.7. Métrica 7. Liderazgo, inspiración y filosofía.

Analizada en la definición para estos dos proyectos.

3.2.8. Métrica 8. Vinculación de datos.

Se acude a la documentación de cada proyecto para informarse del soporte que proveen a esta funcionalidad, la cual es una de las principales y más importantes características técnicas de estas nuevas tecnologías MVC. Óptimamente, para nuestros proyectos, se está buscando una vinculación de datos bidireccional.

A fecha de redacción de este trabajo, estas son las características que se han encontrado de algunos de los proyectos, los dos primeros son los que nos interesan:

Proyecto	Vinculación de datos
EmberJs	Bidireccional
AngularJs	Bidireccional
Batman	Bidireccional
CanJs	Bidireccional
Knockout	Bidireccional
Meteor	Bidireccional
Backbone	No
Spine	No

Tabla 3.1 Vinculación de datos de los proyectos.

Ambas tecnologías tiene una vinculación de datos bidireccional, por lo que en lo que respecta a esta métrica, ambas son igualmente válidas para nuestros propósitos.

3.2.9. Métrica 9. Enrutamiento.

EmberJs tiene lo que se podría considerar como el mecanismo de enrutamiento incorporado por defecto más avanzado del mercado. Permite definir estados y rutas, y anidarlas entre sí.

El mecanismo de AngularJs solo es capaz de alcanzar al de EmberJs a través de Angular-route, un proyecto independiente que se debe incluir como dependencia para poder utilizarlo.

Por lo tanto, esta métrica se decanta hacia el lado de EmberJs.

3.2.10. Métrica 10. Plantillas de visualización.

Como se comentaba en la definición de la métrica, los datos del modelo JavaScript del lado del cliente necesitan ser combinados con el HTML y estos proyectos siguen uno de los siguientes dos caminos para resolver este problema:

- Plantillas basadas en cadenas.
- Plantillas basadas en DOM.

Para más información, ver el apartado ***2.3.3.*** de definición de la métrica.

No se cree que una de las 2 opciones sea mejor que la otra, pero el grupo de trabajo tiene una preferencia por las primeras, ya que ya están siendo utilizadas en otros proyectos y requeriría un menor esfuerzo de cara al aprendizaje por parte de los desarrolladores y una potencial ventaja a la hora de la reutilización de código.

A fecha de redacción de este trabajo, estas son las características que se han encontrado de algunos de los proyectos, de nuevo, los dos primeros son los que nos interesan:

Proyecto	**Plantillas de visualización**	**Normalmente utilizadas**
EmberJs	Basadas en cadenas	Handlebars.js
AngularJs	Basadas en DOM	-
Batman	Basadas en DOM	-
CanJs	Basadas en cadenas	Mustache.js
Knockout	Basadas en DOM	-
Meteor	Basadas en cadenas	-
Spine	Basadas en cadenas	Underscode.jr, Handlebars.js
Backbone	Basadas en cadenas	Underscode.jr, Handlebars.js

Tabla 3.2 Plantillas de visualización de los proyectos.

3.2.11. Métrica 11. Seguimiento de cambios en el modelo.

En este apartado EmberJs gana ventaja sobre AngularJs por la manera que tiene de comprobar los cambios en el modelo. Mientras que Angular utiliza lo que se conoce como “dirty checking”

(comprobar cada cierto tiempo si se ha producido un cambio) [21], EmberJs utiliza "change listeners" (se mantiene a la espera de ser notificado de un cambio) para actualizar la plantilla con los cambios del modelo.

La segunda opción es potencialmente mucho más escalable y una clara ventaja para interfaces complejas y con gran cantidad de datos referenciados como es nuestro caso.

```
App.Room = Ember.Object.extend({
  area: function() {
    return this.get('width') * this.get('height');
  }.property('width', 'height')
});
```

Figura 3.10 Objeto básico en Ember.Js

La propiedad área escucha cambios en "width" y "height" para ser recalculada en EmberJs.

```
<p>Room:</p>

<p>{{width}}</p>
<p>{{height}}</p>
<p>{{area}}</p>
```

Figura 3.11 Plantilla de visualización Handlebars mostrando las propiedades.

3.2.12. Métrica 12. Almacenamiento de datos. Persistencia.

AngularJs utiliza AJAX de manera manual o un cliente RESTful en este apartado, mientras que EmberJs tiene un sistema montado por defecto para que el desarrollador no tenga que realizar nada más que configurarlo. Además, cuenta con una librería externa de persistencia de datos llamada Ember Data muy útil de la que hablaremos más adelante.

Cabe destacar que el sistema de EmberJs se puede sobrescribir de acuerdo a las necesidades del proyecto y es muy modular, en nuestro caso, se ha tenido que sobrescribir un modulo para permitir el cacheo de unos datos que no podía funcionar con el sistema por defecto.

3.3. Conclusiones.

En este apartado se recopilan los resultados de las métricas analizadas.

Se ha tomado como punto de partida los proyectos de código libre que se están utilizando actualmente por un número significativo de desarrolladores, que son:

- AmplifyJs
- Spine
- AngularJs
- KnockOut
- Sammy.Js
- EmberJs
- Batman.Js
- CanJs
- AgilityJs

De la primera métrica, la popularidad y comunidad, se ha extraído una de las conclusiones que más ha limitado el campo de selección. Actualmente solo hay 3 proyectos con una comunidad tan activa y madura que satisfaga nuestro requisito en este aspecto:

- AngularJs
- KnockOut
- EmberJs

En la segunda métrica, la curva de aprendizaje, se ha desarrollado un pequeño proyecto de prueba con cada una de las tecnologías que nos quedaba y estudiado su documentación, entendiendo de esta manera como de empinada es su curva de aprendizaje, y algunas diferencias técnicas y de filosofía muy importantes. Cabe destacar que la apuesta de EmberJs por la convención frente a la configuración es muy atractiva para el entorno en el que se van a desarrollar los proyectos, el CERN. Además, se ha descartado KnockOut por no aportar la funcionalidad básica que se requiere para nuestros proyectos.

A partir de este momento, por las características de nuestros proyectos, la decisión ha quedado entre estas dos tecnologías:

- AngularJs
- EmberJs

De la métrica número 3 se ha concluido que los 2 se consideran a sí mismos “frameworks” más que “Librerías” y que EmberJs tiene un tamaño superior que puede ser debido al extra de funcionalidad que parece proveer respecto a AngularJs. No se descarta ninguno.

De la métrica número 4 se deduce que la diferencia de tamaño de las dependencias no es significativa. No se descarta ninguno.

La métrica 5 junto con los requisitos de nuestros proyectos no limita ninguna de las dos tecnologías, ambas funcionan mejor cuando tienen el control de toda la página, pero no supone un problema para nuestro caso. No se descarta ninguno.

La métrica 6 también da luz verde a nuestros requisitos. No se descarta ninguno.

Las métricas 7 es correcta para ambos proyectos. No se descarta ninguno.

Métrica 8, vinculación de datos, bidireccional en los dos casos. No se descarta ninguno.

Métrica 9, enrutamiento, es un claro requisito de nuestros proyectos y EmberJs saca ventaja. No se descarta ninguno.

Métrica 10, plantillas de visualización, ligera ventaja para EmberJs porque el grupo de trabajo tiene una preferencia Handlebars.Js, debido a que ya están siendo utilizadas en otros proyectos y requeriría un menor trabajo de cara al aprendizaje de los desarrolladores y una potencial ventaja a la hora de la reutilización de código. No se descarta ninguno.

En la métrica 11 EmberJs es claro vencedor para nuestras aplicaciones con una alta escalabilidad. No se descarta ninguno.

Y por último, en la métrica 12 los dos cumplen la función que se necesita.

Para los requisitos de nuestros proyectos, estas dos tecnologías podrían ser válidas, pero EmberJs cumple con los mismos de una manera más eficiente, mientras que no se encuentra nada que destaque por encima de él en AngularJs. Por lo tanto, aunque la curva de aprendizaje sea un poco más empinada al comienzo de la integración de EmberJs, se cree que se pueden conseguir unos rendimientos superiores con esta tecnología en el largo plazo.

Seleccionamos EmberJs.

CAPÍTULO 4: ARQUITECTURA DE EMBERJS.

En este capítulo se expone la estructura y el funcionamiento más básico de EmberJs.

4.1. Preparación.

"*A framework for creating ambitious web applications*" (Un framework para crear aplicaciones web ambiciosas).

En una aplicación web típica, se trae información de la base de datos y se utiliza para crear el HTML que se envía posteriormente al navegador. Algunas aplicaciones también tienen APIs, las cuales pueden ser utilizadas para comunicarse con aplicaciones nativas, por ejemplo, en un Iphone, a través de IOS, o con un teléfono móvil Android. Y ésta es la parte interesante de EmberJs, está en el cliente, y se comunica con la aplicación a través de esta misma API, cogiendo los datos, y pasándolos a la pantalla por medio de distintas plantillas de visualización.

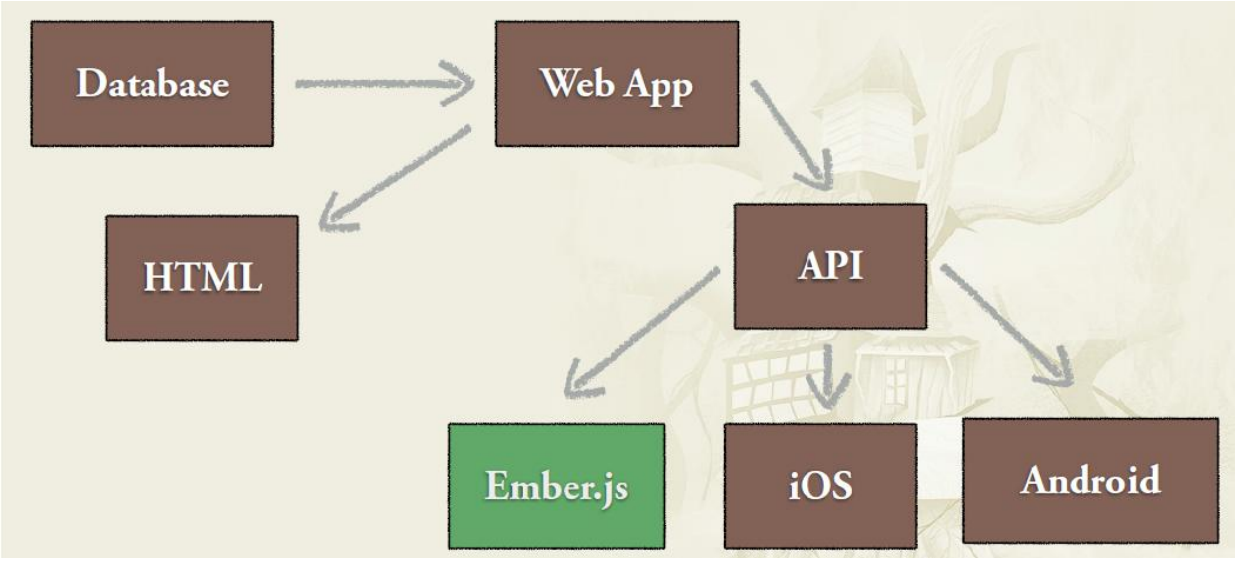

Figura 4.1 Estructura de una aplicación web moderna.

¿Qué se debería conocer para empezar a trabajar con EmberJs?

- HTML y CSS
- JavaScript
- jQuery

Tecnologías como EmberJs son extremadamente útiles a la hora de desarrollar una interfaz que va a tener mucha **interactividad**.

El primer paso para desarrollar una aplicación con EmberJs es ir a su página web y descargarse el "starter kit" que hay disponible y que contiene todas las librerías que se necesitan para empezar un nuevo proyecto [22].
Además, si se quiere comunicar con un servidor ("backend") para traer los datos a través de una API, puede ser de gran utilidad descargarse la librería Ember Data de la que se hablará más adelante. [23]

A continuación, se preparará un "index.html" muy básico de nuestro proyecto. Para ello, se incluye la librería *"jQuery.js"*, la librería "*handlebars.js"* que se verá más adelante, la librería *"Ember.js"*, la librería *"Ember-data"* y por último un archivo que se va a crear enseguida y sobre el que se estará trabajando durante mucho tiempo, ya que es donde se va a encontrar la mayor parte del código JavaScript: el archivo *"App.js"*.
Además, y de manera opcional, se incluye el CSS de Bootstrap para poder trabajar con una hoja de estilos básica.

```
<html>
  <head>
    <script src="jquery.js"></script>
    <script src="handlebars.js"></script>
    <script src="ember.js"></script>
    <script src="ember-data.js"></script>
    <script src="app.js"></script>

    <link href="bootstrap.css"
          media="screen"
          rel="stylesheet" />
  </head>
</html>
```

Figura 4.2 Página principal de nuestro proyecto. Index.html

El primer paso para crear una aplicación EmberJs, es crear el objeto aplicación dentro del fichero *"app.js"*:

```
var App = Ember.Application.create({ });
```

Figura 4.3 Objeto aplicación. *App.js*

Este objeto "App" contiene todo acerca de nuestra aplicación, y puede ser nombrado de la manera que se desee. Solo hay que crearlo una vez, y el resto de componentes vendrán de él.

Lo primero que se puede hacer con esta aplicación es pasarle una serie de opciones en forma de objeto JavaScript. Por ejemplo, si se quiere mostrar un mensaje en el navegador cada vez que se acceda a una página, se activa "LOG_TRANSITIONS", lo cual será muy útil a la hora de depurar la página:

```
var App = Ember.Application.create({
  LOG_TRANSITIONS: true
});
```

Figura 4.4 Activando el log de transiciones. *App.js*

A continuación, se añade el *"body"* a nuestro *"index.html"*. Para abrir esta aplicación se podría hacer doble clic en el archivo *"index.html"*, ya que no se necesita ningún servidor para ejecutar una aplicación de EmberJs. Así que, si se hace y se examina el código fuente HTML, se verá que Ember ha añadido una clase al *"body"* y una extensión de *"Ember Data"* al código. De esta manera, Ember sabe qué parte de la página puede controlar.

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body class='ember-application' data-ember-extension='1'>
    <div class='navbar'>...</div>
    <div class='container'>...</div>
    <footer class='container'>...</footer>
  </body>
</html>
```

Figura 4.5 Clase y extensión de *"Ember data"* al *"body"* para el control de la página.

Uno de los objetivos de la aplicación, es ser capaces de actualizar dinámicamente partes de la misma basándonos por ejemplo en el lugar en el que hace un clic el usuario. Para ello, se necesita ser capaz de decirle al navegador qué partes de la página son dinámicas y por lo tanto pueden ser modificadas.

Con este objetivo, se insertarán los elementos del *"body"* en una plantilla de visualización. EmberJs, por defecto, utiliza *"handlebars.js"* (la librería que se ha incluido previamente).

Figura 4.6 Plantilla *"Handlebars". Index.html*

Por lo tanto, si ahora se abre el "*index.html"* y se inspecciona en el navegador, se verá que se ha añadido un nuevo elemento *"div"*, representando esta plantilla de visualización *"Handlebars"*, y que a su vez, se ha generado un identificador único para referenciarlo.

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body class='ember-application' data-ember-extension='1'>
    <div id='ember248' class='ember-view'>
      <div class='navbar'>...</div>
      <div class='container'>...</div>
      <footer class='container'>...</footer>
    </div>
  </body>
</html>
```

Figura 4.7 Elemento *"div"* representando plantilla *"Handlebars"* con identificador único.

4.2. La aplicación.

Ahora que se tiene una plantilla de visualización *"Handlebars"*, se añadirá contenido dinámico al interior a través de una *"expresión Handlebars*", la cual permite insertar una variable dentro de una página HTML. La variable *"siteName"* tiene que ser dada por la aplicación Ember como se verá más adelante.

```
<script type='text/x-handlebars'>
  <div class='navbar'>...</div>
  <div class='container'>
    <h1>Welcome to {{siteName}}!</h1>
  </div>
  <footer class='container'>...</footer>
</script>
```

Figura 4.8 Expresión Handlebars. *Index.html*

Ahora mismo, la plantilla contiene 3 elementos:

- Una barra de navegación.
- El contenedor, donde irá la mayor parte del contenido de la página.
- El pie de página (*"footer"*).

La aplicación tendrá más páginas, y se quiere mantener la barra de navegación y el pie de página. Se verá cómo crear una página principal (*"Home Page"*), y una página sobre nosotros (*"About Page"*).
Para ello, se nombrará la plantilla actual como plantilla aplicación (*"application template"*), y se crearán dos nuevas plantillas: *"index template"* y *"about template"*. Cada plantilla de visualización necesita un nombre único.

La plantilla de visualización *"application"* se mostrará para todas las páginas por defecto.

```
<script type='text/x-handlebars' data-template-name='application'>
  <div class='navbar'>...</div>
  <div class='container'>...</div>
  <footer class='container'>...</footer>
</script>
```

Figura 4.9 Plantilla de visualización *"application". Index.html*

Se crea otra plantilla para el *"index"*, y dentro se pone el código HTML para la misma. Lo mismo para la plantilla *"about"*.

```
<script type='text/x-handlebars' data-template-name='application'>
  <div class='navbar'>...</div>
  <div class='container'>...</div>
  <footer class='container'>...</footer>
</script>

<script type='text/x-handlebars' data-template-name='index'>
  <h1>Welcome to The Flint & Flame!</h1>
</script>

<script type='text/x-handlebars' data-template-name='about'>
  <h1>About The Fire Spirits</h1>
</script>
```

Figura 4.10 Plantillas de visualización de *"index"* y *"about". Index.html*

Por último, se debe definir dónde se tienen que visualizar estas plantillas. Para ello, se utiliza un *"outlet"*, el cual le dice a la aplicación en qué lugar insertar las plantillas.
Cuando el código de Ember encuentra un *"outlet"*, por defecto, va a buscar la plantilla *"index"* y mostrarla en el lugar del *"outlet"*.

```
<script type='text/x-handlebars' data-template-name='application'>
  <div class='navbar'>...</div>
  <div class='container'>{{outlet}}</div>
  <footer class='container'>...</footer>
</script>

<script type='text/x-handlebars' data-template-name='index'>
  <h1>Welcome to The Flint & Flame!</h1>
</script>

<script type='text/x-handlebars' data-template-name='about'>
  <h1>About The Fire Spirits</h1>
</script>
```

Figura 4.11 Reutilización de código utilizando *"outlet". Index.html*

4.3. El enrutador.

En esta sección se verá cómo cargar la plantilla *"about"* en la aplicación, y cómo realizar un mapeo con la URL con el objetivo de que cuando un usuario ingrese en la ruta *"aplicacion.com#/about"* se cargue la plantilla correspondiente. Para ello, se utiliza lo que se conoce como *"Ember Router"*.

"Ember Router" traduce un *"path"* en una ruta, y es el lugar donde comienzan todas las peticiones a la aplicación. Cada página del sitio web se define en el *"Router"*.

```
App.Router.map(function() {
  this.route('about');
});
```

Figura 4.12 Enrutador de la aplicación. *App.js*

Por ejemplo, cuando el navegador entra al *"path about"*, la aplicación muestra la ruta *"about"* y el template *"about"* (cargado dentro del *"outlet"*).

Si se quiere definir un *"path"* distinto al *"path"* por defecto (*"/about"*), se le pasa un nuevo objeto a la ruta.

```
App.Router.map(function() {
  this.route('about', { path: '/aboutus' } );
});
```

Figura 4.13 Modificando en *"path"* por defecto en el enrutador. *App.js*

4.4. Handlebars Helpers.

En ocasiones, se quiere utilizar el mismo HTML varias veces en una aplicación. En estos casos, los *"helpers"* son muy útiles.
EmberJs incorpora varios *"helpers"* por defecto, pero además, se pueden crear nuevos que cubran necesidades específicas.

Un ejemplo de estos *"helpers"* por defecto es *"link-to"*, el cual utiliza una ruta para cargar dinámicamente URLs y realizar una transición de estados. Es decir, permite encontrar el *"path"* de una URL a través del nombre de la ruta.

```
{{#link-to 'index'}}Homepage{{/link-to}}
{{#link-to 'about'}}About{{/link-to}}
```

Figura 4.14 *"Helper Link-to"* por defecto. *Index.html*

Lo cual resulta en el siguiente código en el navegador:

```
<a class="ember-view" href="#/" id='ember2'>Homepage</a>
<a class="ember-view" href="#/about" id='ember3'>About</a>
```

Figura 4.15 Código en el navegador del *"Helper Link-to"* por defecto.

En el caso de que se quiera añadir una clase al link, se especificaría dentro del *"helper"*:

```
{{#link-to 'index' class='navbar-brand' }}Homepage{{/link-to}}
```

Figura 4.16 Añadir clase en el *"Helper"*. *Index.html*

```
<a class="ember-view navbar-brand" href="#/" id='ember2'>Homepage</a>
```

Figura 4.17 Clase en el *"Helper"*. Código en el navegador.

Si lo que se quiere es definir el elemento HTML que se va a utilizar, por ejemplo *"<li>"*, se utiliza la propiedad *"tagName"*.

```
{{#link-to 'index' class='navbar-brand' }}Homepage{{/link-to}}
```

Figura 4.18 Elemento HTML en el *"Helper"*. *Index.html*

```
<a class="ember-view navbar-brand" href="#/" id='ember2'>Homepage</a>
```

Figura 4.19 Elemento HTML en el *"Helper"*. Código en el navegador.

4.5. Controladores y vinculación de datos.

En este apartado se verán los controladores, encargados de decorar el modelo de datos para las plantillas. Además, son el lugar donde la aplicación busca el valor de un atributo, y contienen información que no se almacena en el servidor.

El siguiente código dentro de la plantilla *"index"*, va a buscar el contador de productos (*"productsCount"*) en el controlador *"index"*.
Ésta es una de las grandes ventajas de las convenciones de Ember: se sabe dónde buscar cada funcionalidad. La ruta *"index"* va asociada a la plantilla *"index"* y, al mismo tiempo, al controlador *"index"*.

```
<script type='text/x-handlebars' data-template-name='index'>
  <p>There are {{productsCount}} products</p>
</script>
```

Figura 4.20 Plantilla de visualización *"index"* con atributo contado de productor. *Index.html*

Se crea el controlador dentro del espacio de nombres de la aplicación (*"App"*), con el nombre "*IndexController"*, por convención, y se define la propiedad *"productsCount"*. Por defecto, éste es el controlador que la ruta va a buscar.

```
App.IndexController = Ember.Controller.extend({
  productsCount: 6
});
```

Figura 4.21 Controlador para la plantilla y ruta *"Index". App.js*

Automáticamente, Ember ya crea un controlador para cada ruta existente, el cual puede extenderse en el caso de querer crear atributos dentro de él (*"productsCount"*).

Si se inspecciona el código se encuentra lo siguiente:

```
<p>There are
  <script id="metamorph-2-start" type="text/x-placeholder"></script>
  6
  <script id="metamorph-2-end" type="text/x-placeholder"></script>
  products
</p>
```

Figura 4.22 HTML generado por la plantilla de visualización *"index"*. Código en el navegador.

El valor 6 está envuelto en un "script" con un identificador *"metamorph"* que Ember utiliza para seguir el rastro de estas propiedades y poder actualizarlas dinámicamente. Ésto es la **vinculación de datos**.

Al mismo tiempo, puede generar un problema cuando las propiedades son a su vez atributos, como en el siguiente ejemplo, en el que se pretende incluir un logo a la página:

```
App.IndexController = Ember.Controller.extend({
  productsCount: 6,
  logo: '/images/logo.png'
});
```

Figura 4.23 Controlador de *"Index"* con la propiedad logo. *App.js*

Según lo expuesto anteriormente, el código *"Handlebars"* se definiría de la siguiente manera:

```
<script type='text/x-handlebars' data-template-name='index'>
  <p>There are {{productsCount}} products</p>
  <img src='{{logo}}' alt='Logo' />
</script>
```

Figura 4.24 Plantilla de visualización *"index"* con un logo. *Index.html*

Pero entonces se generaría el siguiente código HTML para el navegador, que obviamente no es el que se desea.

```
<img src="
  <script id="metamorph-2-start" type="text/x-placeholder"></script>
     /images/logo.png
  <script id="metamorph-2-end" type="text/x-placeholder"></script>
>
```

Figura 4.25 Código erróneo en el navegador.

Para solucionar este problema, cuando las propiedades son al mismo tiempo atributos, se necesita realizar la vinculación de datos utilizando el *"helper bind-attr"*.

```
<script type='text/x-handlebars' data-template-name='index'>
  <p>There are {{productsCount}} products</p>
  <img {{bind-attr src='logo'}} alt='Logo' />
</script>
```

Figura 4.26 Plantilla *"Index"* con el atributo logo correctamente insertado. *Index.html*

De esta manera se genera correctamente el código, y además, se añade un atributo extra (*"data-bindattr"*) para que Ember pueda seguir manteniendo la vinculación de datos.

```
<img src="/images/logo.png" data-bindattr-1="1">
```

Figura 4.27 Código en el navegador con el atributo logo correctamente insertado.

También se tiene la posibilidad de definir propiedades en el controlador que sean funciones (propiedades computadas), las cuales se ejecutan cuando se necesita su valor, o con la modificación de otra propiedad de la que son observadoras.

4.6. Rutas.

En este apartado se analizan las rutas, responsables de cargar el modelo.
Cada ruta decide qué modelo se utilizará y que plantilla de visualización será mostrada cuando sea accedida.
Éste modelo pasa de la ruta al controlador, y finalmente a la plantilla de visualización.

No se debe confundir con el enrutador. Enrutador solamente hay uno, y es el encargado de traducir direcciones URL en rutas, mientras que rutas puede haber varias, y se encargan de proveer datos al controlador.

La ruta, al igual que el controlador, es creada por Ember si no está definida por el desarrollador.

```
App.ProductsRoute = Ember.Route.extend({ });
```

Figura 4.28 Ejemplo de una ruta creada por Ember por defecto. *App.js*

El siguiente es un ejemplo de un modelo estático que se podría utilizar para esta ruta:

```
App.PRODUCTS = [
  {
    title: 'Flint',
    price: 99,
    description: 'Flint is…',
    isOnSale: true,
    image: 'flint.png'
  },
  {
    title: 'Kindling',
    price: 249,
    description: 'Easily…',
    isOnSale: false,
    image: 'kindling.png'
  }
];
```

Figura 4.29 Ejemplo de modelo estático para la ruta productos. *App.js*

Para pasar este modelo al controlador, se utiliza la función *"model"*.

```
App.ProductsRoute = Ember.Route.extend({
  model: function() {
    return App.PRODUCTS;
  }
});
```

Figura 4.30 Función *"model"* que pasa los datos al controlador. *App.js.*

Para mostrar estos datos en la plantilla de visualización se pueden iterar en la misma plantilla:

```
<script type='text/x-handlebars' data-template-name='products'>
  <h2>Products</h2>
  {{#each product in model}}
    <h2>{{product.title}}</h2>
  {{/each}}
</script>
```

Figura 4.31 Iteración del modelo en una plantilla de visualización. *Index.html*

En este ejemplo, el modelo se está utilizando directamente, es decir, sin que haya sido decorado por el controlador. Sin embargo, no se debe olvidar que el controlador podría haberlo modificado y devolverlo de la forma deseada como una propiedad computada.

Éste es el código que se genera en el navegador al iterar el modelo:

```
<h1>Products</h1>
<h2>Flint</h2>
<h2>Kindling</h2>
```

Figura 4.32 Código generado en el navegador al iterar el modelo.

En la práctica, los modelos de estas rutas se cargarán dinámicamente a través de una API RESTful.

CAPÍTULO 5: ARQUITECTURA DE SAM3

5.1. Introducción.

Como caso de uso para la incorporación de EmberJs a los proyectos del grupo "**Soporte para la Computación Distribuida**" del departamento IT del CERN, se ha seleccionado la interfaz gráfica de la aplicación SAM3 ("Site Availability Monitoring") [24]. Ésta se utiliza para monitorizar la disponibilidad y fiabilidad de los centros de cómputo de los resultados de los experimentos del CERN y recoge información muy útil para los usuarios del WLCG ("Worldwide LHC Computing Grid") [25].

Actualmente, la interfaz del proyecto SAM2 (*) se encuentra desarrollada en JavaScript puro y la componen dos ficheros principales de unas mil líneas de código cada uno.

El objetivo es rescribirlo de una manera modular, utilizando la tecnología MVC EmberJs, e incrementar la facilidad de mantenimiento del código y la funcionalidad que provee la aplicación, gracias a un menor esfuerzo en el desarrollo al utilizar una tecnología que facilita el trabajo.

Finalmente, se desplegarán cuatro instancias del proyecto en máquinas de pre-producción (QA), y otras cuatro en máquinas de producción. Una para cada experimento del CERN: Atlas [26], Alice [27], CMS [28] y Lhcb [29].

(*) El proyecto SAM2 se encuentra adjunto a la memoria de este trabajo.

5.2. Estructura de ficheros.

Para el desarrollo de la interfaz de usuario de SAM3 se utilizará *"EmberJs"* y *"Ember-data"* como librerías principales del proyecto; *"Ember inspector"* como ayuda para inspeccionar el código; *"Highcharts"* como librería para crear los gráficos; *"raphaelJs"* [30] como librería de apoyo para los gráficos para un caso especial al que *"Highcharts"* no da soporte y *"Handlebars"* y *"jQuery"* como dependencias de EmberJs.

Se seguirá la siguiente estructura de ficheros para la interfaz gráfica:

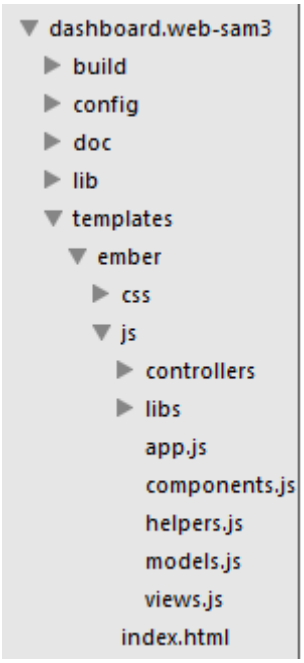

Figura 5.1 Estructura de ficheros de SAM3.

Dentro de la carpeta Ember se cargará el *"index.html"*, donde se insertarán todas las plantillas de visualización *"Handlebars"* (en el futuro se pueden separar en varios archivos *"*.html"* utilizando una herramienta llamada *Ember Cli* [31]).

Por otro lado, se tiene una carpeta dedicada a los CSS de la interfaz, y otra para todo el código JavaScript dividido de la siguiente manera:

- En el archivo *"App.Js"* se cargará el enrutador y todas las rutas necesarias.

- En el archivo *"Components.js"* se crearán tres componentes web [32]: dos para manejar gráficos (uno para *"Highcharts"* y otro para *"Raphael"*) y un tercero para crear un componente calendario que permita al usuario elegir fechas de una manera gráfica (debe ser un componente porque hay que asegurarse que el DOM está cargado cuando se lanza el calendario).

- En el archivo *"Helpers.js*" se prepararán dos *"helpers"* personalizados para cubrir necesidades específicas de la interfaz.

- En el archivo *"Models.js"* se definirán unos modelos de datos que se utilizarán para cachear datos y realizar consultas a los mismos cuando sea necesario utilizando *"Ember-Data"* sin tener que volver a conectar con el servidor.

- En el archivo *"Views.js"* se extenderá la funcionalidad de Ember para incluir un *"RadioButton"* que no provee.

5.3. Módulos principales.

La aplicación se dividirá, al menos, en **cinco grandes módulos** principales. Cada módulo se define como un estado en el enrutador y tiene su propia ruta, su controlador y su plantilla de visualización. Así mismo, podrá incorporar componentes, vistas y *"helpers*" compartidos por todos ellos, y utilizar los datos cacheados en los modelos.

Al inicio, se proveerán **dos módulos encargados de manejar los últimos resultados** de los test de disponibilidad y fiabilidad que calcula la aplicación.
Uno de los módulos será el encargado de los paneles de selección de opciones. Lo que conlleva, mostrar las agrupaciones de sitios, los sitios en función de las agrupaciones seleccionadas, los perfiles disponibles y los servicios y métricas de que disponen los perfiles seleccionados. Todos

estos paneles deberán actualizarse dinámicamente en función de las selecciones. Además, se proveerá de un campo de búsqueda de sitios en tiempo real, y de un filtrado en función del estado de las métricas, que se utilizará como parámetro dentro de la aplicación. Estos dos últimos campos se podrán manipular sin necesidad de conectarse con el servidor, realizando todos los cambios en el cliente.

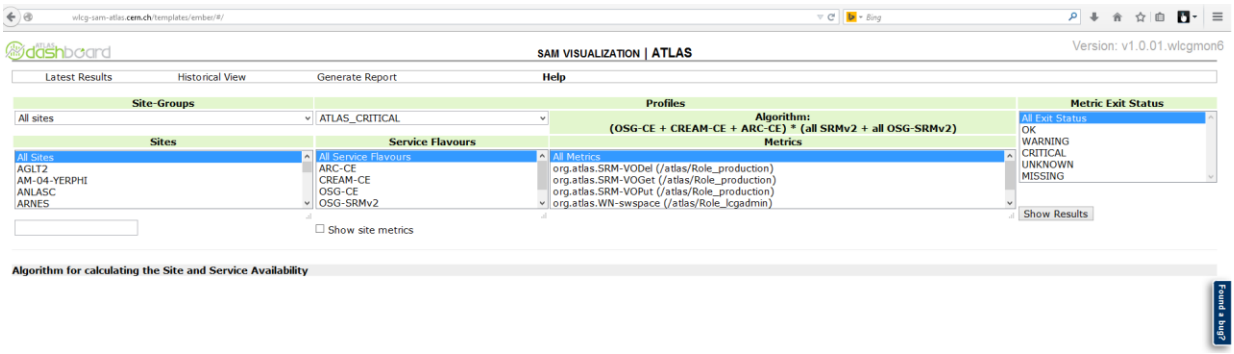

Figura 5.2 Módulo 1. Paneles de selección de opciones de últimos resultados.

El segundo módulo se encargará de interaccionar con la API del servidor, cargar los datos correspondientes a los parámetros definidos por las selecciones del primer módulo, y mostrarlos acordes con el filtrado del estado de las métricas. También se encargará de las interacciones del usuario con los resultados y de manejar los errores.

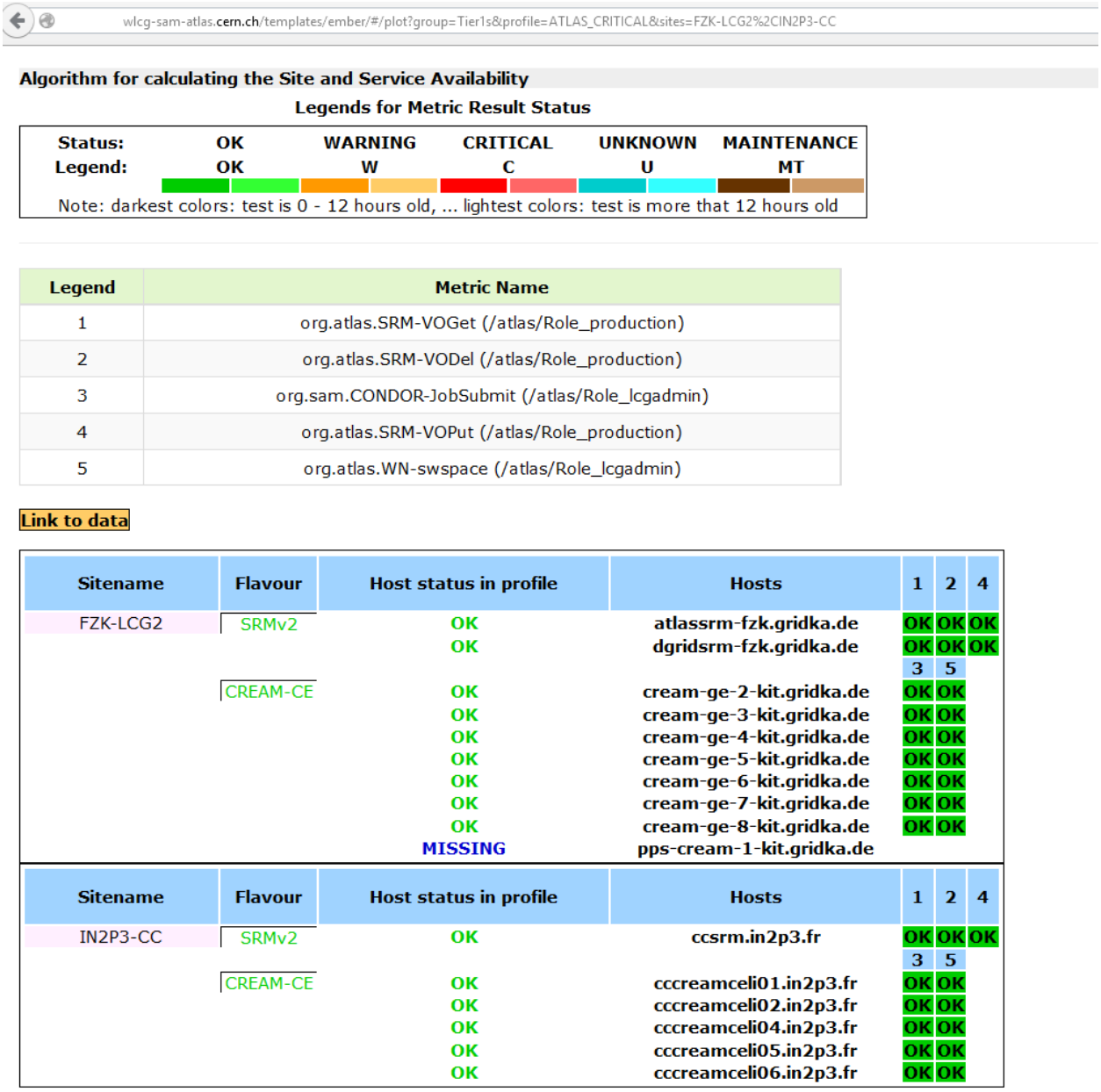

Figura 5.3 Módulo 2. Últimos resultados.

Se realizarán otros **dos módulos para manejar el histórico de resultados** de una manera similar. El primer módulo dispondrá de cinco vistas distintas de cara al usuario: disponibilidad de un sitio, fiabilidad de un sitio, disponibilidad de un servicio, fiabilidad de un servicio y finalmente histórico de test. Cada una de las vistas tiene diversas opciones de selección al igual que en el apartado anterior. De igual manera, el segundo módulo se encargará de interaccionar con la API del servidor, cargar los datos correspondientes a los parámetros definidos por las

selecciones del primer módulo, y mostrarlos utilizando para ello las dos librerías de gráficos que se han incorporado al proyecto. También se encarga de las interacciones del usuario con los resultados y de manejar los errores.

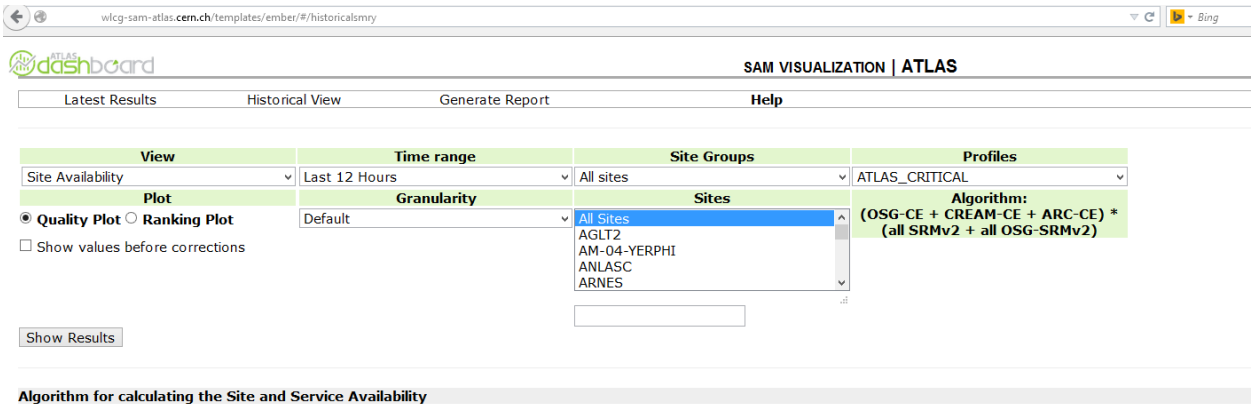

Figura 5.4 Módulo 3. Paneles de selección de opciones del histórico. Vista: Disponibilidad del sitio.

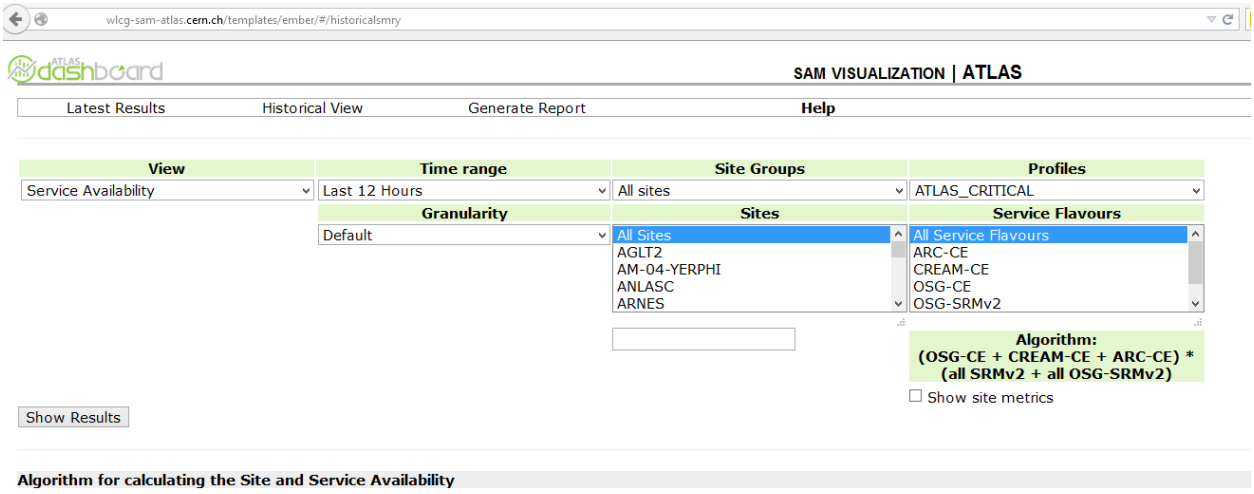

Figura 5.5 Módulo 3. Paneles de selección de opciones del histórico. Vista: Disponibilidad del servicio.

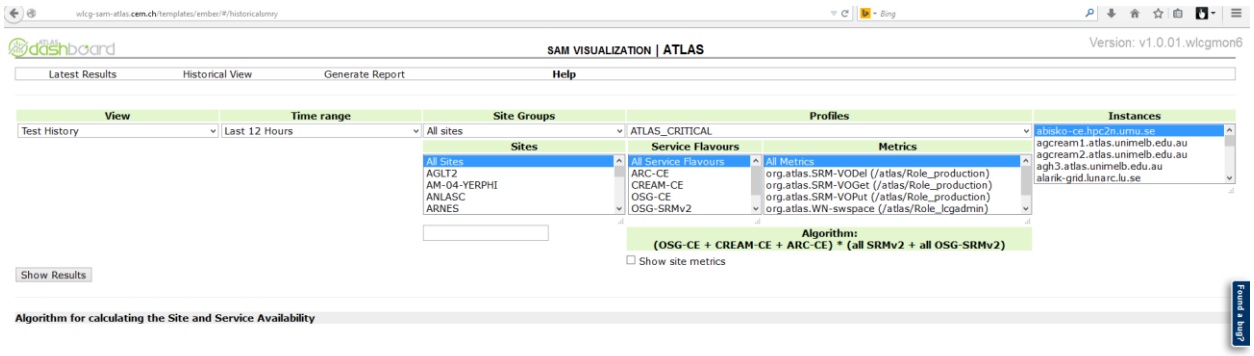

Figura 5.6 Módulo 3. Paneles de selección de opciones del histórico. Vista: Histórico de tests.

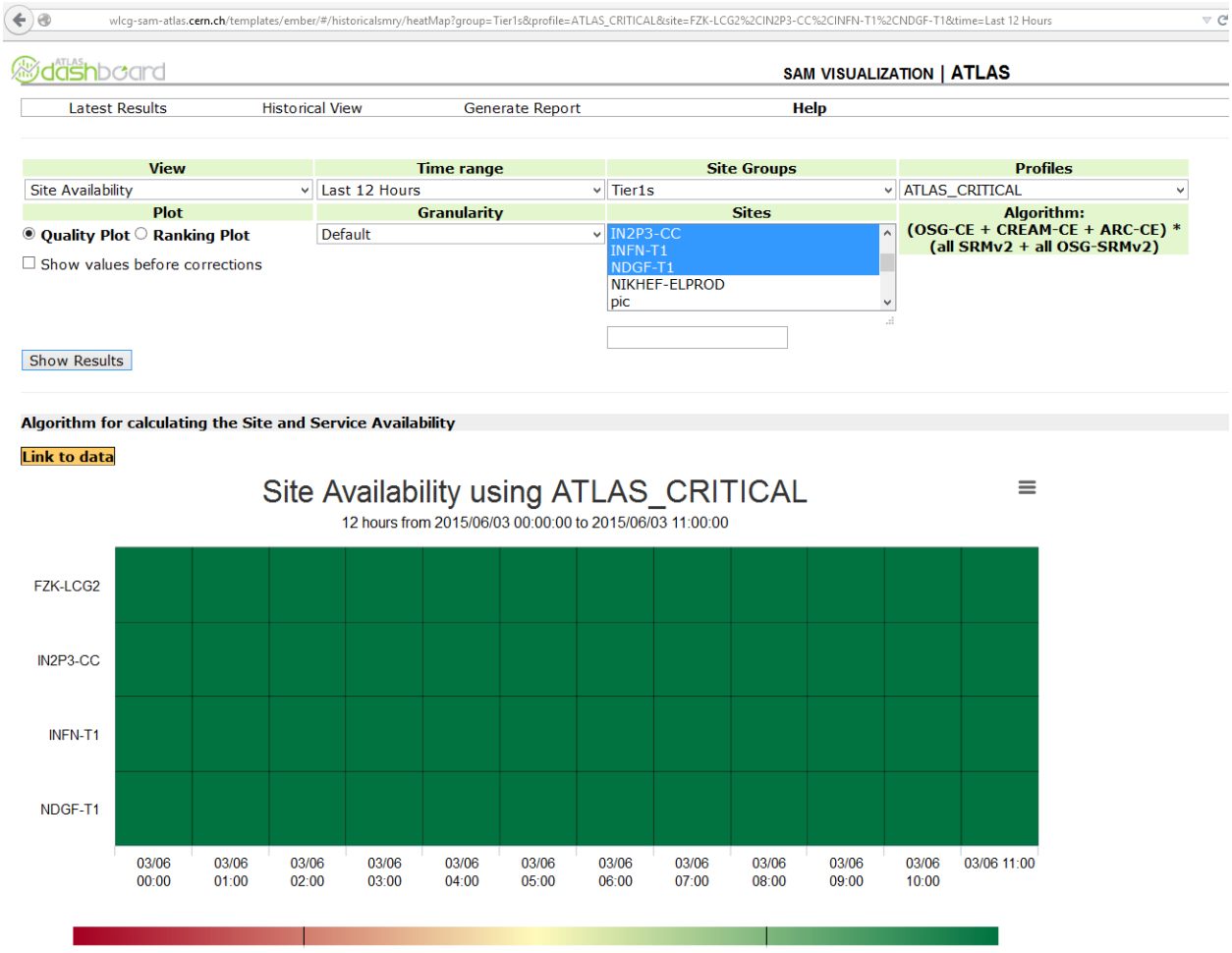

Figura 5.7 Módulo 4. Mapa caliente del histórico de resultados. Vista: Disponibilidad del sitio.

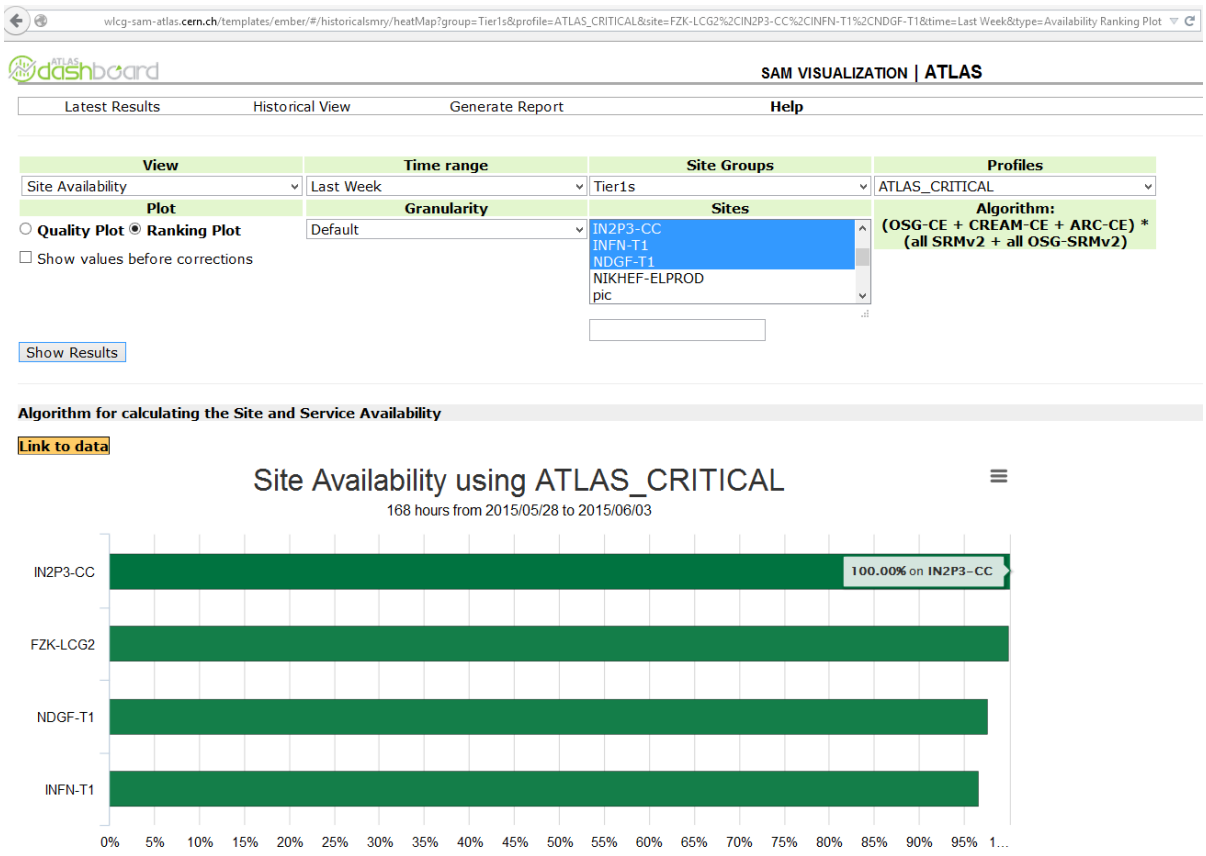

Figura 5.8 Módulo 4. Ranking del histórico de resultados. Vista: Disponibilidad del sitio.

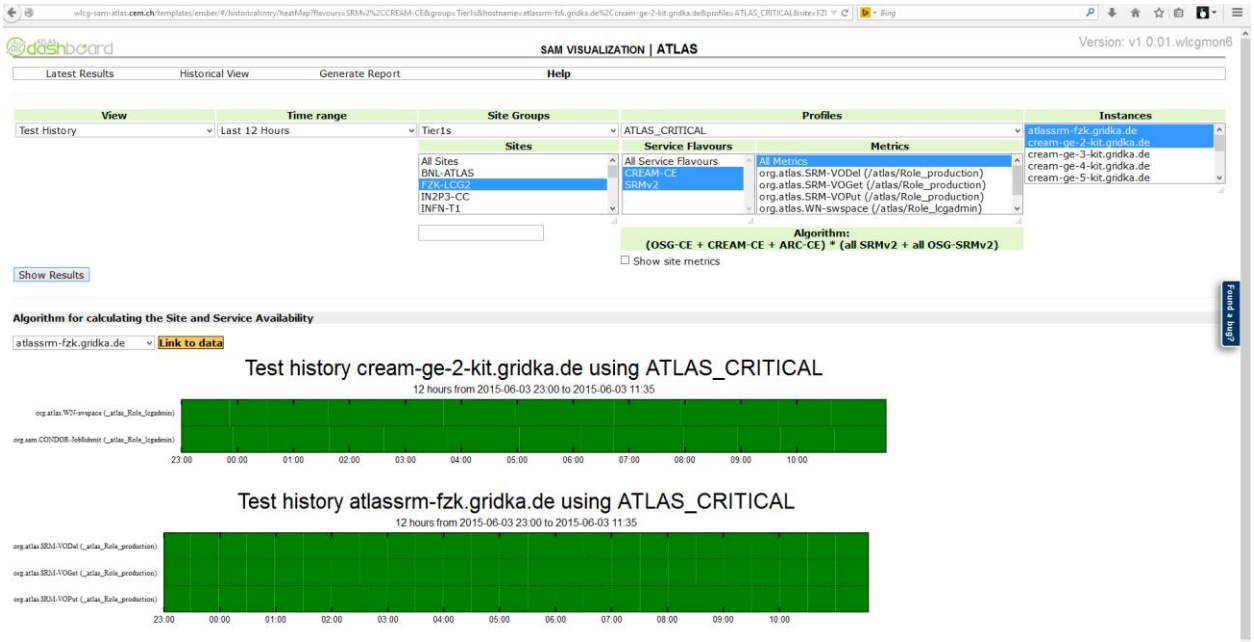

Figura 5.9 Módulo 4. Mapa caliente del histórico de resultados. Vista: Histórico de test.

Finalmente, se dispondrá de **un quinto módulo** que será la interfaz utilizada **para generar informes** basados en los datos de disponibilidad y fiabilidad de los centros por experimento y perfil.

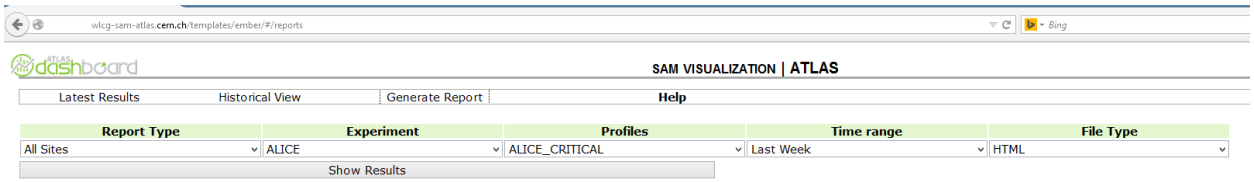

Figura 5.10 Módulo 5. Interfaz para generar informes.

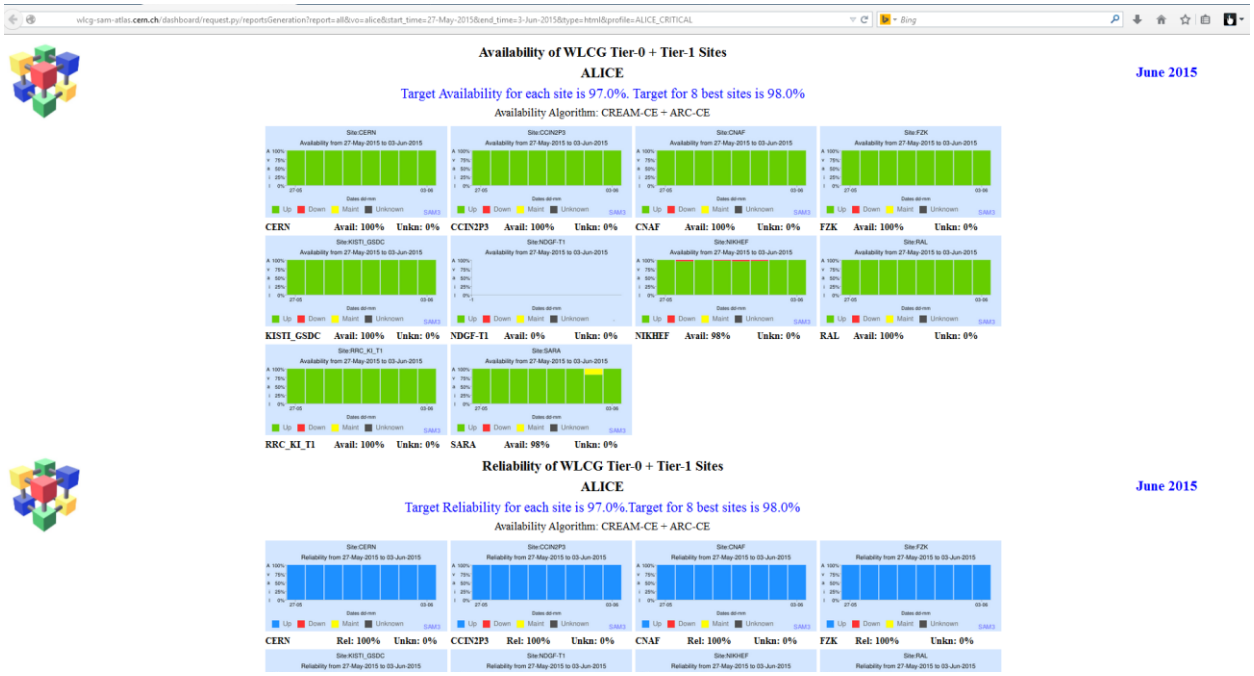

Figura 5.11 Ejemplo de informe generado por el módulo cinco.

En los dos primeros módulos de las secciones “Últimos Resultados” e “Histórico de Resultados”, se cachean los datos que se utilizan para cargar las opciones de las selecciones. Para los segundos módulos, en cambio, se utiliza AJAX a la hora de traer los datos del servidor. Los segundos módulos se definirán como hijos de los primeros, ya que son dependientes de ellos, y necesitan las selecciones realizadas en los padres y algunos de los datos cacheados en ellos para realizar su función.

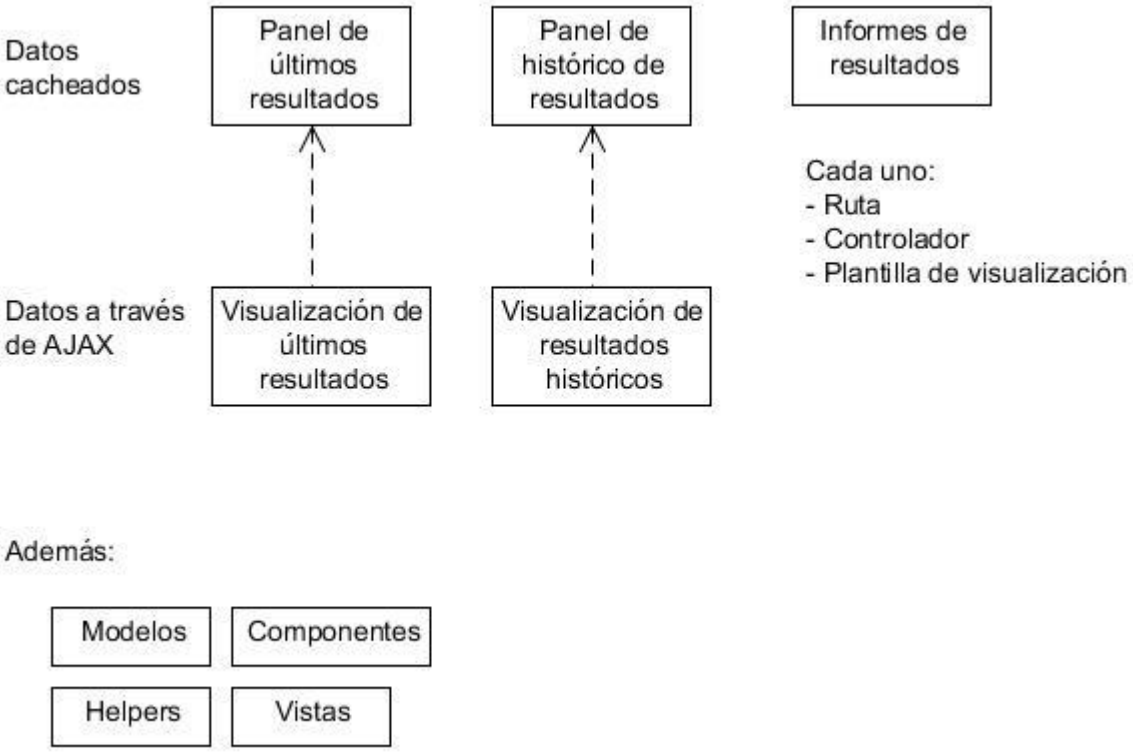

Figura 5.12 Estructura de la aplicación. Módulos.

5.4. Arquitectura de los modelos de datos.

Un modelo en EmberJs es una clase que define las propiedades y el comportamiento de los datos que se presentan al usuario. Cada ruta puede definir qué modelo utilizará el controlador.
En la introducción a Ember se mostró de qué manera utilizar un modelo estático.

En este proyecto, se quieren definir modelos para los datos que van a utilizar los tres módulos de selección. Además, más adelante se verá cómo realizar una optimización para cachear estos datos en la máquina del cliente utilizando estos mismos modelos.

Los modelos deben seguir la estructura que tiene el JSON que devuelve la API del servidor.
Para su correcto entendimiento, se aconseja revisar el ejemplo de JSON devuelto que se encuentra anexado a este trabajo para la topología y para las métricas: *Apéndice A* y *Apéndice B.*

En base al JSON, se trabajará primeramente con sitios y agrupaciones de sitios, es decir, con una topología, por lo que se deben definir unos modelos acordes a estos datos.

```
App.Topologymin = DS.Model.extend({
  siteGroup: DS.attr('string'),
  sites: DS.hasMany('site')
});

App.Site = DS.Model.extend({
  name: DS.attr('string'),
  hosts: DS.attr()
});
```

Figura 5.13 Modelos para la topología y los sitios. *Models.js*

En el modelo topología se define la relación de agrupación de los sitios, y en el modelo sitio, una relación entre el nombre de los sitios y los “hosts” que tiene asociados.

Los atributos que no tienen un tipo definido como por ejemplo *"string"*, tomarán el valor por defecto que se encuentre en los datos JSON con los que se van a rellenar estos modelos. A las estructuras complejas no se las puede definir un tipo, sólo se admite definición de estructuras simples.

Se realiza lo mismo para los datos referentes a los Perfiles y Servicios siguiendo la estructura que tiene el JSON, devuelto por la API del servidor.

```
App.Metricmap = DS.Model.extend({
  ProfileName: DS.attr('string'),
  Critical: DS.attr('string'),
  algorithm: DS.attr('string'),
  FlavourMetrics: DS.attr()
});
```

Figura 5.14 Modelo para los perfiles y servicios. *Models.js*

De esta manera, se crean los modelos necesarios para todos los datos de los paneles de selección.

Finalmente, se deberán rellenar estos modelos con los datos deseados. En nuestro caso, vamos a utilizar una librería, *"Ember-data"*, para facilitar el tener estos dados cacheados en cliente. La incorporación y configuración del adaptador REST se verá en la sección de optimización.

Los resultados mostrados al usuario, en cambio, se traerán en tiempo de ejecución desde la ruta utilizando para ello AJAX, por lo que no definimos ningún modelo para los mismos.

5.5. Arquitectura de los parámetros.

El flujo básico de transmisión asíncrona de datos del histórico y de los últimos resultados es el siguiente:
En los dos módulos padre se seleccionan las opciones en los paneles disponibles. Una vez seleccionadas, se pasa el control a los módulos hijos mediante el botón "Mostrar Resultados" y se acompaña de las selecciones en forma de parámetros.
La ruta del hijo (el "model hook") se encarga, en este momento, de traer los datos definidos por los parámetros del servidor mediante una llamada AJAX y de pasárselos al controlador. Esta acción está dirigida por la URL, por lo que se puede compartir la ruta del hijo, junto con los parámetros, entre distintos usuarios, refrescar la página, o acceder directamente al hijo sin haber estado en el padre previamente, y funcionará correctamente.

En EmberJs, los parámetros (*"query params"*) se declaran en los controladores. Por lo tanto, se tienen dos opciones de diseño diferentes: declararlos en los controladores padres (los de selección) o en los controladores hijos.
Para escoger el diseño más adecuado se debe comprender en profundidad el funcionamiento de EmberJs.
Cuando un usuario **accede directamente** a la ruta de un hijo (*/padre/hijo?parámetros*) el orden de carga es el siguiente:

- Carga del modelo del padre (*"model hook"*).
- Carga del modelo del hijo (*"model hook"*).
- Carga del controlador del padre (*"setupController"*).
- Carga del controlador del hijo (*"setupController"*).

El hijo necesita los parámetros para cargar su modelo, por lo tanto, **si se definen los parámetros en el controlador del padre**, cuando **se accede directamente** a la ruta del hijo sin pasar previamente por la del padre, bien porque la URL se ha compartido, o porque se refresca la página, a la hora de cargar el modelo del hijo, todavía no está listo el controlador del padre, por lo que **no se tiene acceso a sus parámetros**.

Por lo tanto, para este diseño de aplicación, **los parámetros se deben declarar en el controlador del hijo,** ya que un módulo en concreto, si puede acceder a los parámetros de su controlador desde la función que carga el modelo, aunque este controlador no haya sido aún inicializado.

Las selecciones del padre dirigen la URL, y la URL dirige los datos del hijo, lo que crea una sincronía entre estos tres elementos: selecciones, URL y datos mostrados. Ahora bien, cuando se **accede directamente al hijo**, la URL dirige los datos, pero las selecciones no están en sincronía, si no que muestran los valores por defecto.

Figura 5.15 Sincronía entre las selecciones, la URL y los datos mostrados.

Por ello, para cerrar el círculo, cuando los parámetros están listos (controlador del hijo), se debe comprobar si la vinculación con las selecciones del padre es la correcta, y si no es así, corregirlo.

Figura 5.16 Sincronía correcta entre las selecciones, la URL y los datos mostrados.

Por otra parte, como unas selecciones disparan la recarga del contenido de otros paneles de selección, la comprobación y corrección debe realizarse en diferentes etapas.

```
153   bindingParams: function(){
154     Ember.run.once(this, 'setSelects');
155     Em.run.next(this, 'nextLoop');
156   }.on('init'),
157
158   setSelects: function(){
159    if (this.get('group') != undefined && this.get('controllers.application.selectedSiteGroup') != this.get('group'
160        this.set('controllers.application.selectedSiteGroup', this.get('group'));
161
162    var siteNamesInPanel = this.get('controllers.application').allSitesSelected().map(function(data){
163        return data.get('name');
164    });
165    var siteNames = (this.get('sites')).split(',');
166    if (siteNames.sort().join() != siteNamesInPanel.sort().join()){
167      Ember.Logger.log("Sites set");
168      var sites = []; //refereces
169      var self = this;
170      siteNames.forEach(function(e,i,a){
171         sites.push(self.store.all('site').filterBy('name', e).get('firstObject'));
172      });
173  //     this.set('controllers.application.selectedSiteDefault', sites);
174       this.set('controllers.application.selectedSite', sites);
175    }
176
177    if (this.get('profile') != undefined && this.get('controllers.application.selectedProfile') != this.get('profil
178        this.set('controllers.application.selectedProfile', this.get('profile'));
179
180    if (this.get('status') != undefined && this.get('controllers.application.selectedMetricExitStatus') != this.get
181       Ember.Logger.log("Metric status set");
182       this.set('controllers.application.selectedMetricExitStatus', this.get('status').split(','));
183    }
```

Figura 5.17 Manteniendo la sincronía entre selecciones, URL y datos mostrados. PlotController.js

De esta manera, se asegura siempre la sincronía de los tres elementos dirigida por la URL.

5.6. Optimización de la estructura.

El módulo del panel de últimos resultados y el módulo del panel de histórico de resultados comparten una serie de lógica en la aplicación. Comparten la manera de calcular y ordenar los perfiles, los grupos de sitios, los sitios, los servicios y las métricas. También comparten el cálculo de la URL del logo, y del algoritmo, así como la lógica que controla los valores por defecto de las selecciones.

En un principio, la parte del histórico de resultados reutilizaba esta lógica del controlador de últimos resultados:

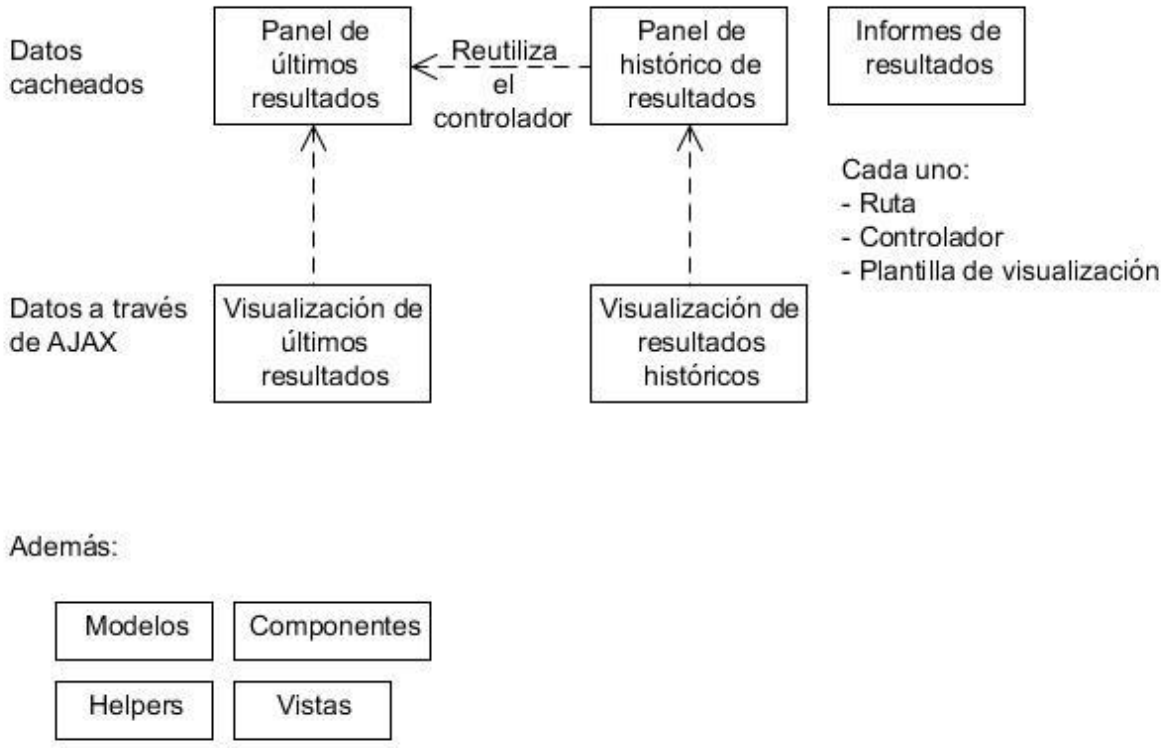

Figura 5.18 El módulo histórico reutiliza el controlador de los últimos resultados.

Sin embargo, esta estructura tiene un pequeño inconveniente, cuando se accede directamente a la ruta del histórico de resultados, sin haber estado previamente en el módulo de últimos resultados, el modelo y el controlador de últimos resultados no está cargado, y cuando se intenta reutilizar el controlador, falla. Para resolverlo, la ruta del módulo histórico de resultados se debe encargar de asegurarse de que está cargado, y en caso contrario, cargarlo. Una vez cargado, el controlador del histórico puede recibir el control del programa.

```
App.HistoricalsmryRoute = Ember.Route.extend({
  model: function(){
        Ember.Logger.log("Se carga el model de Historicalsmry");
        var topologymin = this.store.find('topologymin');
        var metricmap = this.store.find('metricmap');
        return Ember.RSVP.hash({
            topologymin: topologymin,
            metricmap: metricmap
        });
  },

  setupController: function(controller, model) {
    controller.set('model', model);
    this.controllerFor('panel').set("model", model);
  }
});
```

Figura 5.19 La ruta del módulo Histórico prepara el modelo del módulo Últimos resultados (Panel). *App.js*

Como esta aproximación no es consistente con el hecho de que los dos módulos padres se encuentran al mismo nivel de abstracción, y además, presenta el problema previamente descrito, se decidió mover la lógica compartida por ambos a nuevo un módulo superior que se llamó *"Application"*.

Ésta es la estructura resultante:

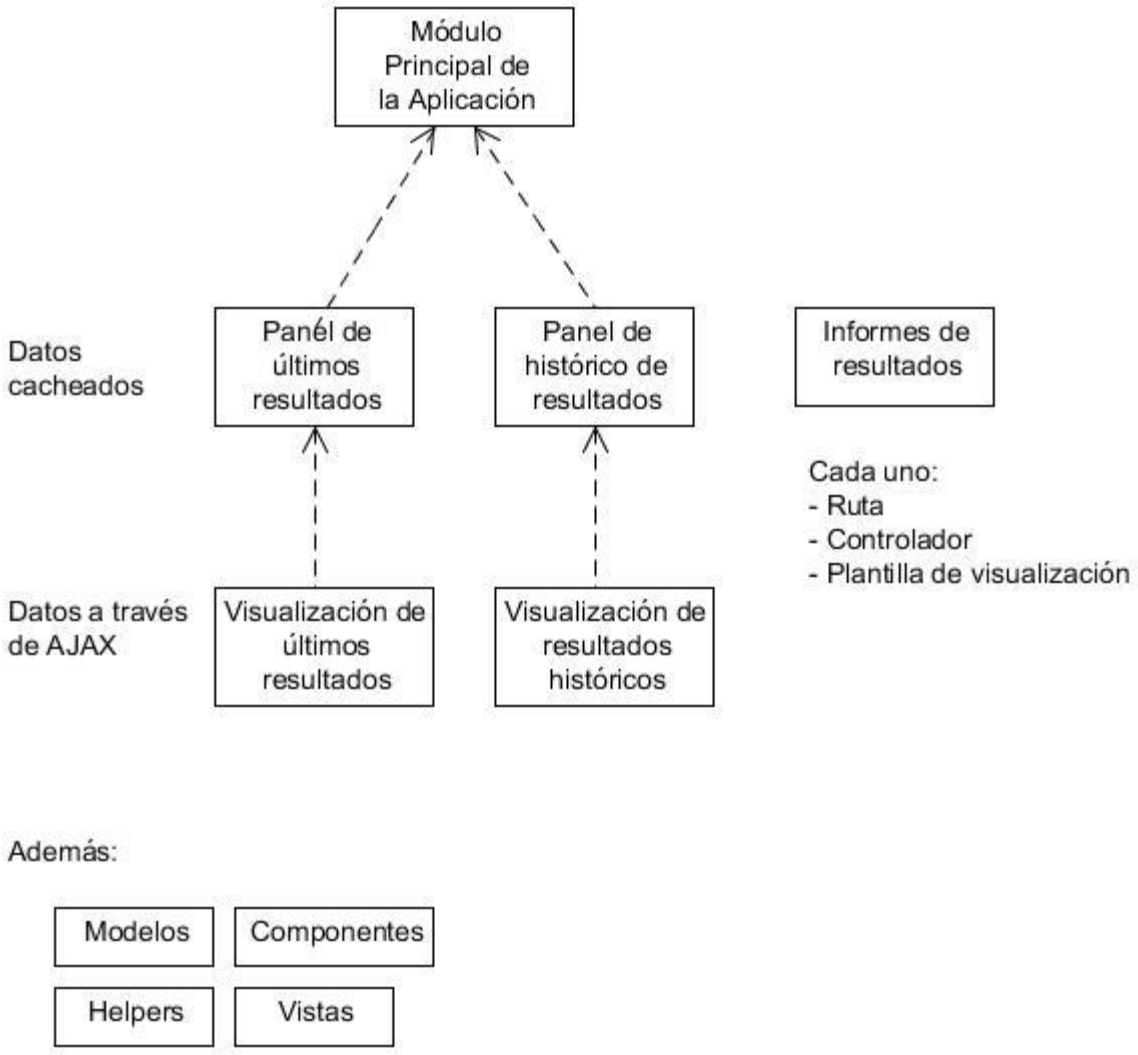

Figura 5.20 Estructura optimizada de la aplicación. Seis módulos.

Ambos utilizan la lógica definida en el módulo aplicación y heredan su modelo.

```
App.HistoricalsmryRoute = Ember.Route.extend({
  model: function(){
    Ember.Logger.log("Se carga el model de Historicalsmry");
    return this.modelFor("application");
  },
});
```

Figura 5.21 Ruta del Histórico reutilizando el modelo de Aplicación. *App.js*

5.7. Optimización de la caché.

Como se ha comentado anteriormente, se ha decidido utilizar la librería *"Ember-data"* para optimizar el almacenamiento de datos de los módulos de selección, cacheándolos en el lado del cliente.
"Ember-data" es una librería de persistencia de datos para EmberJs.
Ésto nos permitirá que sólo se interaccione una vez con el servidor para traer todos estos datos, en el momento en el que se inicia la aplicación, y que no sea necesario realizarlo de nuevo al transitar por ejemplo de la vista del panel de selección de últimos resultados, al del histórico de resultados, o al de los informes. Cuando uno de estos tres módulos es accedido, se traen los datos y posteriormente, se reutilizan para el resto de ellos.

Para empezar a utilizar *"Ember-data"*, lo primero que se necesita es crear unos modelos de datos como se ha explicado en el apartado *3.4.3. Modelos de datos*.

"Ember-data" hace más fácil utilizar modelos para traer datos del servidor, cachearlos para mejorar el rendimiento, guardar actualizaciones de vuelta al servidor, y crear nuevos datos en el cliente.
También posee diferentes adaptadores que se pueden utilizar para cargar los datos. En este caso, se va a utilizar el adaptador por defecto, REST, que nos permitirá comunicarnos con un servidor HTTP utilizando JSON.

Al configurar el adaptador REST, se debe indicar la URL donde contactar con la API del servidor para traer los datos que van a llenar las distintas "instancias" de las "clases" de los modelos.

```
DS.RESTAdapter.reopen({
  host: '../../dashboard/request.py'
});
```

Figura 5.22 Configuración del adaptador REST. *Models.js*

Si no se realizara esto, por defecto buscaría en el nombre del dominio actual.

En nuestro caso, también se debe indicar cómo llamar al servicio requerido que devolverá todos los datos en formato JSON necesarios para rellenar el modelo de la topología visto anteriormente.

```
App.TopologyminAdapter = DS.RESTAdapter.extend({
  pathForType: function(type) {
    return "gettopologymin";
  },
});
```

Figura 5.23 Configuración del *"path"* para cargar los datos. *Models.js*

De lo contrario, el adaptador REST utilizaría las convenciones, y trataría de crear la URL con el nombre del modelo, en este caso, buscaría los datos en:
"../../dashboard/request.py/topologymins".

Además, se ha de indicar que no se dispone de un identificador único numérico para cada dato en el atributo *"id"*, y que se ha de tomar como identificador el nombre del sitio y el nombre de la agrupación.

```
App.TopologyminSerializer = DS.RESTSerializer.extend({
  normalize: function(type, hash, property) {
    // Ember Data use the zone name as the ID.
    hash.id = hash.siteGroup;
    // Delegate to any type-specific normalizations.
    return this._super(type, hash, property);
  }
});

App.SiteSerializer = DS.RESTSerializer.extend({
  normalize: function(type, hash, property) {
    hash.id = hash.name;
    return this._super(type, hash, property);
  }
});
```

Figura 5.24 Distinto identificador por defecto. *Models.js*

Cuando la función que carga el modelo dentro de una de las rutas llame a la "topología", esta configuración se pondrá en funcionamiento y los datos se cachearán dentro de los modelos, siendo posible acceder a ellos posteriormente sin más interacción con el servidor.

Una vez se tiene configurado el adaptador REST, se verá como utiliza la *"store"*.

"Ember-data" posee una *"store"*, que es el nombre que recibe el repositorio de los datos en la aplicación, y el cual se encuentra disponible para su acceso desde las rutas y desde los controladores.
La primera vez que se llama a un modelo de la *"store"*, se pondrá en funcionamiento la configuración del adaptador REST, y se traerán los datos para ese modelo, permaneciendo cacheados en el repositorio para el futuro.

```
App.ApplicationRoute = Ember.Route.extend({
  model: function(){
    Ember.Logger.log("Retrieving model for App..");
    var topologymin = this.store.find('topologymin');
    var metricmap = this.store.find('metricmap');
    var Url = "../../dashboard/request.py/getMeta";
    var meta = Ember.$.ajax({ url: Url , dataType: "json", type: 'GET' });
    return Ember.RSVP.hash({
        topologymin: topologymin,
        metricmap: metricmap,
        meta: meta
    });
  }
});
```

Figura 5.25 Cargando la topología y las métricas del repositorio *"store"*. *App.js*

En la imagen, se están cargando todos los datos de la topología y las métricas con una simple petición al repositorio.

También se podrían filtrar por identificador, o por cualquiera de los atributos que tienen. De forma que para una ruta correspondiente a una URL de la manera *"/producto/id"*, sería muy sencillo extraer sólo los datos con un identificador único dentro un modelo en concreto, y

mostrar la visualización correspondiente a *"producto/1"* o a *"producto/2"* automáticamente a través de este filtrado.

5.8. Optimización basada en entorno Cloud.

Una práctica común en el desarrollo de aplicaciones es la de delegar parte del trabajo que realiza la aplicación a algún servidor Cloud bajo demanda. En este apartado se analiza si esta práctica podría dar solución a dos problemas teóricos planteados: que la máquina cliente tenga una potencia muy baja, y/o que el ancho de banda del cliente sea muy limitado.

Es posible que un usuario acceda a la aplicación desde una **máquina cliente con una baja potencia** (un teléfono móvil, por ejemplo). Esto podría llegar a suponer un problema si el dispositivo es tan poco potente que no tiene capacidad de procesar la lógica que se ejecuta en el navegador, es decir, para ejecutar el código *JavaScript + HTML*.

Para solucionarlo, se proponen tres opciones:

1. Nada de computación de datos en el cliente. Todo se realiza en el servidor y se envía solo lo que se vaya a mostrar. Esto implicaría modificar el servidor y la interfaz de usuario.

2. Reducir al máximo el peso de la interfaz grafica en el cliente. Esto implicaría incorporar una interfaz de usuario diferente para estos casos.

3. Utilización de un **navegador virtual desplegado en un entorno Cloud** para ejecutar la aplicación.

La mejor solución de cara al desarrollo de la aplicación es la tercera, ya que no requiere de una implementación extra, únicamente se debe de ejecutar la aplicación en un navegador virtual al que el usuario se conecta.
Sin embargo, se cree que este problema **no es realista**. Cualquier dispositivo de hoy en día tiene una potencia suficiente, y al realizar varias pruebas se llega a la conclusión de que para que se dé este caso se necesita un procesador con tan poca potencia que no existe en el mercado para ningún dispositivo que disponga de un navegador web.
Por otro lado, se plantea el caso de que el ancho de banda del cliente sea muy limitado, y se estudia si se podría resolver mediante un **navegador virtual**. Para ello, la conexión entre el cliente y el navegador virtual debería ser menos exigente que la conexión con la aplicación, cosa que no ocurre, ya que conectarse al navegador virtual es extremadamente costoso al requerir la visualización remota del navegador en el dispositivo del usuario.

Finalmente, podría utilizarse un Cloud para las bases de datos o para ejecutar alguno de los algoritmos del servidor, pero el CERN dispone de una gran potencia de servidores propios, por lo que estos escenarios no aportan una mejoría.

Por lo tanto, no realizamos ninguna optimización basada en un entorno Cloud a la aplicación.

CAPÍTULO 6: RESULTADOS, CONCLUSIONES Y TRABAJOS FUTUROS.

6.1. Contribuciones.

En este apartado se habla sobre cuáles son las principales contribuciones del proyecto y se destacan las conclusiones de mayor relevancia sobre el trabajo llevado a cabo.

En primer lugar, se ha definido un modelo de adecuación de tecnologías JavaScript MVC en el cliente, que establece una métricas atemporales que permiten diferenciarlas y valorarlas, y que ayudan a determinar y elegir la más adecuada acorde a las necesidades de un proyecto.

Se ha probado este modelo de métricas en el caso real que lo motivó, el Experimento Dashboard del CERN, del grupo "Soporte a la Computación Distribuida" del departamento IT.

La tecnología seleccionada se ha utilizado para un caso real de trabajo en el CERN, el desarrollo de la nueva interfaz de SAM3 (*"Site Availability Monitoring"*), una aplicación altamente escalable, y además, se ha analizado el diseño de su arquitectura.

La aplicación SAM3 se encuentra en producción para los cuatro grandes experimentos del CERN: Atlas, Alice, Cms y Lhcb.

Entre los principales beneficios, se encuentran **una reducción del 50% en el número de líneas de código,** un gran incremento de la **modularidad**, pasando a tener seis grandes módulos principales donde cada uno tiene una estructura modular siguiendo el patrón **Modelo Vista Controlador** y un notable incremento de la robustez, y de la facilidad de mantenimiento y ampliación de funciones.

6.2. Trabajos futuros.

- **Ampliación de las métricas**. En primer lugar, se han asentado unas métricas base para el modelo de adecuación de tecnologías JavaScript MVC en el cliente a proyectos, pero estas tecnologías se encuentran en desarrollo, y las métricas deberían ser actualizadas y ampliadas con las características futuras, por ejemplo, midiendo cómo enfocar el desarrollo de unidades de pruebas.

- **Mejoras en la aplicación.** De cara a la aplicación que se ha desarrollado, se podría diseñar un sistema de cacheado de los resultados para que la aplicación pudiera funcionar completamente offline. Crear modelos acordes y adaptar la API para que devuelva un JSON correctamente formateado para los mismos.

  Además, aunque la interfaz actual se adapta a dispositivos móviles, se podría crear una interfaz mucho más ligera para los mismos.

Referencias

[1] Experimento Dashboard:
http://dashboard.cern.ch/

[2] CERN. Organización Europea para la Investigación Nuclear:
http://home.web.cern.ch/

[3] Pebble. Reloj programable:
https://getpebble.com/pebble

[4] Raspberry Pi. Una placa de ordenador del tamaño de una tarjeta de crédito:
https://www.raspberrypi.org/

[5] Kickstarter. Sitio web de crowdfunding:
https://www.kickstarter.com/

[6] Y-Combinator. Aceleradora de Start-ups:
https://www.ycombinator.com/

[7] Incremento de la popularidad de jQuery:
http://data.stackexchange.com/stackoverflow/query/90306/compare-size-and-growth-trends-for-stackoverflow-tags#resultSets

[8] JavaScript lenguaje más popular en GitHub:
https://github.com/trending

[9] MongoDB. base de datos NoSQL:
https://www.mongodb.org/

[10] Mardanov, Azat. Rapid Prototyping with JS. Página 13.

[11] Zakas, Nicholas C. How many users have JavaScript disabled.

[12] Osmani, Addy. 2012, Developing Backbone.js Applications.

[13] "The missing cdn for javascript and css" https://cdnjs.com/

[14] xbrowse:
https://twiki.cern.ch/twiki/bin/view/ArdaGrid/Xbrowseframework

[15] hbrowse:
https://code.google.com/p/hbrowse/

[16] Dashboards Backbone.js. SSB (ATLAS y totros VOs):
http://dashb-atlas-ssb.cern.ch/dashboard/request.py/siteview#currentView=Shifter+view&highlight=false

[17] Dashboards xbrowse:
DDM Dashboard: http://dashb-atlas-data.cern.ch/ddm2/
FTS Dashboard: http://dashb-fts-transfers.cern.ch/ui/
XRootD Dashboard (ATLAS y CMS): http://dashb-atlas-xrootd-transfers.cern.ch/ui/
WLCG Transfers Dashboard: http://dashb-wlcg-transfers.cern.ch/ui/

[18] Dashboards hbrowse:
Real-time Job Monitoring (ATLAS y CMS): http://dashb-atlas-jobdev.cern.ch/dashboard/request.py/jobsummary
User Task Monitoring (ATLAS y CMS): https://dashb-atlas-task.cern.ch/templates/task-analysis/
ATLAS Production Task Monitoring: http://dashb-atlas-task-prod.cern.ch/templates/task-prod/

[19] Curso colaboración entre Google y CodeSchool de AngularJs:
http://campus.codeschool.com/courses/shaping-up-with-angular-js/intro

[20] Objeto "mock":
http://en.wikipedia.org/wiki/Mock_object

[21] “Change listeners vs dirty checking”:
http://stackoverflow.com/questions/9682092/databinding-in-angularjs

[22] EmberJs, descarga del kit de principiantes.
http://emberjs.com/

[23] Ember-Data. Librería de persistencia de datos para EmberJs.
https://github.com/emberjs/data

[24] SAM3 (“Site Availability Monitoring”):
http://wlcg.web.cern.ch/samnagios

[25] WLCG (“Worldwide LHC Computing Grid”):
http://wlcg.web.cern.ch/

[26] Experimento Atlas del CERN:
http://atlas.web.cern.ch/

[27] Experimento Alice del CERN:
http://alice.web.cern.ch/

[28] Experimento CMS del CERN:
http://cms.web.cern.ch/

[29] Experimento Lhcb del CERN:
http://lhcb.web.cern.ch/

[30] librería para crear los gráficos, raphaelJs:
http://g.raphaeljs.com/

[31] Ember CLI, interfaz de línea de comandos:
http://www.ember-cli.com/

[32] Componentes web:
http://www.w3.org/TR/components-intro/

Apéndice A. Ejemplo de métricas JSON.


```
{
  "metricmaps": [
     {

      "profile": "ATLAS_CRITICAL",
      "link": "http://grid-monitoring.cern.ch/mywlcg/sam-
pi/metrics_in_profiles?vo_name=atlas&output=json&profile_name=ATLAS_CRITICAL"
,
      "services": [
        {
          "servicename": "CE",
          "flavours": [
             {
            "flavourname": "CE",
            "metrics": ["emi.ce.CREAMCE-JobSubmit", "org.atlas.WN-swspace"]
             },
             {
            "flavourname": "OSG-CE",
            "metrics": ["emi.ce.CREAMCE-JobSubmit", "org.atlas.WN-swspace"]
             },
             {
            "flavourname": "CREAM-CE",
            "metrics": ["emi.cream.CREAMCE-JobSubmit", "org.atlas.WN-
swspace"]
             }
          ]
        },
        {
          "servicename": "SRM",
          "flavours": [
             {
            "flavourname": "OSG-SRMv2",
            "metrics": ["org.atlas.SRM-VODel", "org.atlas.SRM-VOGet",
"org.atlas.SRM-VOPut"]
             },
             {
            "flavourname": "SRMv2",
            "metrics": ["org.atlas.SRM-VODel", "org.atlas.SRM-VOGet",
"org.atlas.SRM-VOPut"]
             }
          ]
        }
      ]
     }
  ]

}
```

Apéndice B. Ejemplo de topología JSON.

```
{
  "topologymins": [
      {

      "siteGroup": "ATLAS_Cloud_FR",
      "sites": ["BEIJING-LCG2","GRIF-IRFU"]
      },
      {

      "siteGroup": "ATLAS_Federation_UK-ScotGrid",
      "sites": ["BEIJING-LCG2"]
      }
  ],

  "sites": [
    {

      "name": "BEIJING-LCG2",
      "hosts": [
            {
      "flavour": "SRMv2",
      "hosts": ["ccsrm.ihep.ac.cn"]
      },
      {
      "flavour": "CREAM-CE",
      "hosts": ["cce.ihep.ac.cn"]
      }
      ]
    },
    {

      "name": "GRIF-IRFU",
      "hosts": [
            {
      "flavour": "SRMv2",
      "hosts": ["node12.datagrid.cea.fr"]
      },
      {
      "flavour": "CREAM-CE",
      "hosts": ["node74.datagrid.cea.fr"]
      }
      ]
    }
  ]

}
```