Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

ESTUDIO DEL ESTADO DEL ARTE SOBRE FRAMEWORKS MVC PARA EL DESARROLLO DE APLICACIONES WEB CLIENTE, CASO DE ESTUDIO EMBER.JS

TRABAJO FIN DE MÁSTER

Aymar Carolina García Martínez

2013

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

ESTUDIO DEL ESTADO DEL ARTE SOBRE FRAMEWORKS MVC PARA EL DESARROLLO DE APLICACIONES WEB CLIENTE, CASO DE ESTUDIO EMBER.JS

Autor
Aymar Carolina García Martínez

Director
Santiago Pavón Gómez

Departamento de Ingeniería de Sistemas Telemáticos

2013

Resumen

Con el objetivo de efectuar un estudio del estado del arte sobre frameworks MVC para el desarrollo de aplicaciones web del lado del cliente, se hace uso de Ember.js para describir paso a paso la creación de una aplicación de este tipo. La arquitectura de ejecución implicada en este estudio es la que comúnmente utilizan las empresas dedicadas a desarrollo web, la cual indica que se deben desplegar cuatro tipos de modelos: Arquitectura funcional, arquitectura técnica, arquitectura middleware y por último un modelo de arquitectura física. No obstante para el caso de estudio de esta investigación el interés se centra en detallar la parte software de la arquitectura de ejecución, específicamente el modelo de arquitectura técnica, en el cual se involucra el patrón de diseño MVC. Lo interesante será observa como a partir de la exigencia en ofrecer aplicaciones web con una experiencia de usuario aceptable, existe una tendencia creciente a utilizar aplicaciones web dinámicas; aún más, ver como lo que lo que antes se hacía en el servidor, ahora se pueda hacer del lado cliente dejando al servidor solo como un API de acceso a la persistencia de los datos. En este sentido, el uso del lenguaje JavaScript, ha permitido la creación de frameworks, que junto al patrón de diseño MVC, facilitan el entendimiento del código y por tanto, reduce considerablemente el coste de mantenimiento de una aplicación, añadiendo escalabilidad y reutilización de código.

Lo recomendable es que un desarrollador decida implementar un framework MVC basado en JavaScript, cuando se está construyendo una aplicación de una sola página. Este estudio se inclinó por utilizar el Ember.js y agrega además la descripción de un grupo de frameworks con características similares; mas sin embargo, las características presentes en un framework MVC para el desarrollo de aplicaciones web cliente, engloban un sinfín de rasgos específicos, y si el deseo del desarrollador es conocer a fondo los detalles de algún framework MVC en particular, se recomienda dedique tiempo a revisar tanto el código fuente como las características del framework en su página web oficial para determinar que tan bien se puede adaptar a sus necesidades.

Este trabajo se estructura en cinco secciones, donde el primer punto lo ocupa la introducción, el segundo punto el marco conceptual, como tercer punto se estudia el estado actual de algunos frameworks MVC, el punto número cuatro Ember.js como caso de estudio y el último punto, el número cinco, que hace referencia a las conclusiones del estudio.

Abstract

In order to conduct a study of the state of the art in MVC frameworks for web application development client side, Ember.js is used to describe step by step the creation of an application of this type. The execution architecture involved in this study is commonly used for web development companies, which indicates that you should deploy four types of models: functional architecture, technical architecture, middleware architecture and finally a physical architecture model.However, for the case study of this research the focus is on detailing the software implementation architecture, specifically technical architecture model, which engages the MVC design pattern. It will be interesting to see how from the requirement to provide web applications with acceptable user experience, there is a growing tendency to use dynamic web applications, even more, to see that what you previously done on the server, now you can make it in the client side, leaving the server alone as an API to access the data persistence. In this way, the use of the JavaScript language has allowed the creation of frameworks, that together with the MVC design pattern, facilitate the understanding of the code and therefore significantly reduces the cost of maintaining an application, adding scalability and code reuse.

It is recommended that a developer decides to implement an MVC framework based on JavaScript when building a one-page application. For this study used Ember.js also adds the description of a set of frameworks with similar characteristics, but however, the features in an MVC framework for web application development client, encompass a large number of specific features, for this reason if the desire of the developer is to know in depth the details of any particular MVC framework, it is recommended to take time to review both the source code as framework features in its official website to determine how well they can adapt to their needs.

This work is divided into five sections, where the first point is the introduction, the second point the conceptual framework, as the third point is discussed the current state of some MVC frameworks, the number four Ember.js as a case study and finally the conclusions.

Índice general

Resumen i

Abstract iii

Índice general v

Índice de figuras ix

Siglas xi

1 Introducción 1

1.1 Objetivo general 2

1.2 Objetivos específicos 2

2 Marco Conceptual 3

2.1 Historia del desarrollo de aplicaciones web 3

2.2 Generación de los sitios web. 4

2.2.1 Primera generación. 4

2.2.2 Segunda generación. 6

2.2.3 Tercera generación. 7

2.2.4 Cuarta generación. 8

2.3 Aplicaciones web 9

2.3.1 Funcionamiento de una aplicación web del lado del cliente. 10

2.3.2 Arquitectura y diseño de una aplicación web. 11

2.3.2.1 Arquitectura de ejecución. 11

2.4 Tecnologías para el desarrollo de Aplicaciones web del lado del cliente 15

2.4.1.1.1 HTML (*HyperText Markup Language*) 15

2.4.1.1.2 CSS 17

2.4.1.1.3 JavaScript 18

3 Frameworks MVC 19

3.1 Angular.js 20

3.1.1 Principales características ... 20
3.1.2 Estructura de ficheros y organización ... 21
3.1.3 App.ng ... 22
3.1.4 Routing ... 22
3.1.5 Vistas ... 22
3.1.6 Controladores ... 22
3.2 Backbone.js ... 23
3.2.1 Principales características ... 23
3.2.2 Estructura y organización ... 23
3.2.3 El modelo ... 24
3.2.4 La vista ... 25
3.2.5 Routing ... 25
3.3 Spine.js ... 25
3.3.1 Principales características ... 26
3.3.2 Modelo ... 26
3.3.3 Vista y plantillas. ... 27
3.3.4 Controladores ... 27
3.4 Batman.js ... 27
3.4.1 Principales características ... 27
3.4.2 Modelo ... 27
3.4.3 Vistas ... 28
3.4.4 Controladores ... 28
3.5 Ember.js ... 28
3.6 Cuadro comparativo de frameworks MVC ... 29
4 Caso de estudio Ember.js ... 31
4.1 ¿Qué es Ember.js? ... 31
4.1.1 Principales características ... 31
4.1.2 Modelo ... 32
4.1.3 Vista ... 32
4.1.4 Routing ... 33
4.1.5 Controladores ... 33

4.2 Conversiones de nomenclaturas 34
4.3 Sintaxis, funciones e instrucciones en Ember.js 34
4.3.1 Usos de instrucciones en la plantilla 35
4.3.2 Declaración de rutas 37
4.3.3 Declaración y uso de un componente 38
4.3.4 Configurar un Controlador 41
4.3.5 Definición de un modelo 43
4.4 Configuración del entorno de desarrollo 44
4.5 Componentes básicos del *"Kit de inicio"* 45
5 Creando una aplicación con Ember.js 46
6 Conclusiones 60
Bibliografía 62

Índice de figuras

Figura 1. Ejemplo de página web de primera generación. 5
Figura 2. Ejemplo de página web de segunda generación. 7
Figura 3. Ejemplo de un sitio web de tercera generación. 8
Figura 4. Traslado de la lógica hacia el cliente. [12] 9
Figura 5. Flujo de una aplicación cliente con una implementación actual. [13] 10
Figura 6. Arquitectura de ejecución de una aplicación web. 11
Figura 7. Diagrama de sistema. Arquitectura técnica. 12
Figura 8. Diagrama de Subsistema-Arquitectura Técnica. 13
Figura 9. Meta Patrón MV* 15
Figura 10. Estructura básica de un documento HTML 16
Figura 11. Inclusión de CSS en HTML 17
Figura 12. JavaScript embebido en HTML. 18
Figura 13. Estructura App en Backbone 24
Figura 14. Ejemplo de creación de modelo en Spine.js (en lenguaje JavaScript) 26
Figura 15. Acoplamiento plantilla, modelo, controlador en Ember.js 34
Figura 16. Estructura de una plantilla en Ember.js. 35
Figura 17. Uso de plantilla automática handlebars. 36
Figura 18. Uso de condicionales para la plantilla. 36
Figura 19. Enlace a una imagen 37
Figura 20. Enlace a una ruta 37
Figura 21. Enlace al atributo de una clase 37
Figura 22. Uso de Route en Ember.js 38
Figura 23. Declaración de un componente 39
Figura 24. Uso de un elemento personalizado. 39
Figura 25. Vista de un componente en la plantilla index. 40
Figura 26. Uso de propiedades de una plantilla en elementos personalizados. 40
Figura 27. Representación del uso de las propiedades de una plantilla en un componente. 40
Figura 28. Extendiendo la clase ObjectController 41
Figura 29. Configurar la vista para las propiedades a editar 42
Figura 30. Agregar el elemento de cambio. 42
Figura 31. Extendiendo ArrayController 43
Figura 32. Vista parcial y uso de {{#each}} 43
Figura 33. Definición de un modelo en Ember.js 43

Figura 34. Definición de un modelo personalizado. 44
Figura 35. Asignación de atributos para un modelo. 44
Figura 36. Componentes básicos del kit de inicio de Ember.js. 45
Figura 37. Vista index del Starter-Kit en el navegador. 46
Figura 38. Barra de navegación de la aplicación. 49
Figura 39. Esquema de posición de plantillas para el posts. 50
Figura 40. Tabla para la plantilla posts 52
Figura 41. Mensajes recientes 53
Figura 42. Vista del formulario Nuevo Mensaje 55
Figura 43. Ruta "post" anidada. 55

Siglas

AJAX: (*Asynchronous JavaScript And XML).* JavaScript asíncrono y XML. Es una técnica utilizada para crear aplicaciones web interactivas del lado del cliente.
API: *(Application Programming Interface).* Interfaz de programación de aplicación. Hace referencia a un conjunto de funciones y procedimientos que permite ejercer una biblioteca para que pueda ser utilizada por otro software como capa de abstracción.
APP: aplicación web que puede ser accedida por diversos navegadores y terminales de última generación.
ASP: *(Active Server Pages).* Modelo de programación rápida del lado del servidor, creado por Microsoft para crear páginas web generadas dinámicamente, se ha comercializado como un anexo a Internet Information Services (IIS), conocido como ASP clásico. Se asemeja a la programación en Visual Basic Script.
CGI: *Common Gateway Interface.* Es una destacada tecnología aplicada a internet, que permite a un navegador (cliente) solicitar datos de una aplicación ejecutada en un servidor web.
CSS: *Cascading Style Sheets*. Acrónimo utilizado para definir las *"hojas de estilo en cascada"*. Un lenguaje de hojas de estilos usado para describir la presentación semántica de un documento escrito en lenguaje de marcas.
DHTMLX: HTML Dinámico. Es la unión del lenguaje HTML estático con algún lenguaje de interpretación del lado del cliente (por lo general JavaScript), el lenguaje CSS y la jerarquía DOM; para permitir que los Script del lado del cliente de una aplicación web modifiquen el HTML del documento cuando este se haya cargado completamente.
DOM: (*Document Object Model*). Modelo de objeto del documento; es un modelo estándar que permite representar documentos HTML y XML en una interfaz con la finalidad de estos puedan ser modificados y estructurados según un estilo.
EJB: (*Enterprise JavaBeans).* Son API´s de producción empresarial estándares, que especifican en detalle como los servidores de aplicaciones proveen los objetos desde el lado del servidor. Actualmente descrito por los J2EE (Java Plataform, Enterprise Edition o Java Empresarial) de Oracle Corporation.
GUI: *(Graphical user interface)*. Interfaz gráfica de usuario.
HTML: *(HyperText Markup Language).* Lenguaje de marcado hipertextual utilizado para crear páginas web.
JAVA: Lenguaje de programación de propósito general orientado a objetos.
JSON: *(JavaScript Object Notation).*es un formato ligero para el intercambio de datos que se ha convertido en una alternativa a XML en AJAX.
JSP: *(JavaServer Pages).* Tecnología utilizada para crear páginas web dinámicas en HTML y XML bajo el lenguaje JAVA y similar a la programación en PHP.
JSR: *(Java Specification Requests).* Documentos formales que describen especificaciones propuestas y tecnologías para agregar a la plataforma JAVA.
MV*: meta modelo del patrón MVC.
MVC: siglas que representan al patrón de diseño Model View Controller (modelo vista controlador).
MVP: *(Modelo-Vista-Presentador).* Patrón de diseño derivado del meta modelo MV*

MVVM: *(Model View ViewModel)*. Patrón de diseño derivado del meta modelo MV*
ORM: *()*.
POJO: *(Plain Old Java Object*). Es una sigla creada por Martin Fowler, Rebecca Parsons y Josh MacKenzie en septiembre del año 2000 que es utilizada por programadores Java para señalar el uso de clases simples que no dependen de un framework en especial. En particular surge en oposición al modelo planteado por los estándares EJB anteriores al 3.0, en los que los "Enterprise JavaBeans" debían implementar interfaces especiales. Específicamente, un objeto POJO es una instancia de una clase que no extiende ni implementa nada en especial.
REST: *(Representational State Transfer).*Transferencia de estado representacional. Técnica de arquitectura de software que permite describir cualquier interfaz web simple que utilice XML y HTTP.
TCP-IP: *(Transmission Control Protocol- Internet Protocol).* Protocolo de control para la transmisión de datos en internet.
UI: interfaz de usuario.
UML: *Unified Modeling Language.* Lenguaje unificado de modelado. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. Ofrece un estándar para definir un plano del modelo de una aplicación, incluyendo el proceso de negocio, tecnologías utilizadas, esquemas de bases de datos entre otros.
URI: *(Uniform Resource Identifier).* Identificador uniforme de recurso. Es una cadena de caracteres corta que se usa para identificar inequívocamente un recurso accedido desde una red.
URL: *(uniform resource locator).* Localizador uniforme de recurso. Es una secuencia de caracteres que siguen un formato estándar y se utiliza para nombrar recursos en internet.
W3C: *(World Wide Web Consortium).* Consorcio internacional que produce recomendaciones para la WWW.
XML: *(eXtensible Markup Language).* Lenguaje de marcas extensible. Es un lenguaje de marcas derivado del SGML, creado por la W3C.
XSS: *(Cross-site scripting).* XSS es un vector de ataque que puede ser utilizado para robar información delicada del navegador. Las vulnerabilidades XSS han existido desde los primeros días de la Web y son consideradas como un "agujero de seguridad" típico en las aplicaciones web.

1 Introducción

Entendiendo el desarrollo de aplicaciones Web, como la implementación de la solución a una necesidad, con la creación de un proyecto o bien alguna idea en internet a través del uso de las tecnologías web que mejor se adapten a este; cabe mencionar el hecho de que actualmente el mercado se mueve al ritmo en el que avanza la inclusión de la tecnología; y demanda aplicaciones web que puedan ser igualmente accedidas por múltiples navegadores desde un Smartphone, una Tablet, un PC o cualquier dispositivo de última generación. Por lo que requisitos como la escalabilidad, usabilidad, disponibilidad y reutilización de código, ahora son necesarios para el desarrollo de aplicaciones web.

Al tomar en cuenta estas consideraciones, los cambios más claros e importantes en el proceso de desarrollo de una aplicación web han sido la separación de funciones y la reutilización de código. Este punto hay que contemplarlo desde dos perspectivas distintas: no desarrollar elementos o componentes que ya existan y estén probados, y desarrollar pensando en que puede que mañana el código que se está implementando pueda ser reutilizado en distintos proyectos. A un nivel más alto de abstracción se debe también procurar desarrollar módulos lo más independientes posible del resto de la aplicación. Por lo que ahora, al momento de plantear el diseño de una aplicación web, el primer paso es conseguir separar conceptualmente las tareas que el sistema debe desempeñar entre las distintas capas lógicas, tomando como base a la naturaleza de tales tareas.

Se habla ahora de un desarrollo Back-end (detrás del cliente) y un desarrollo Front-end (del lado del cliente). Dónde la aplicación de patrones de diseño y frameworks, facilita el entendimiento del código y, por tanto, reduce considerablemente el coste de mantenimiento, dado que además de aportar soluciones eficientes a los planteamientos anteriores, son muy interesantes como medio de entendimiento entre diseñadores e implementadores.

A consecuencia de lo anterior, este trabajo se centrará en realizar un estudio sobre las novedades en tecnologías que rodean la implementación de un Frameworks MVC para el desarrollo de aplicaciones web del lado del cliente.

1.1 Objetivo general

"Efectuar un estudio del estado del arte sobre frameworks MVC para el desarrollo de aplicaciones web del lado del cliente, empleando Ember.js para detallar la creación de aplicaciones web de este tipo".

1.2 Objetivos específicos

- Considerar la historia y generaciones de las aplicaciones web para comprender la evolución de estas hacia un modelo de desarrollo estructurado, con separación de funciones y reutilizable.
- Describir una arquitectura de desarrollo para aplicaciones web del lado del cliente.
- Explorar las tecnologías implicadas en el uso de frameworks MVC para el desarrollo de aplicaciones web cliente.
- Conceptualizar algunos frameworks que cumplan con el patrón MVC para el desarrollo de aplicaciones web del lado del cliente.
- Detallar el desarrollo de una aplicación web cliente, utilizando el framework Ember.js.

2 Marco Conceptual

Dentro de este punto, se pretende hacer un recorrido por la historia de las aplicaciones web, hasta alcanzar la época actual y comprender teóricamente la necesidad del uso de las tecnologías utilizadas hasta hoy. Por lo que se describirá en detalle el funcionamiento, arquitectura, diseño y tecnologías para el desarrollo de una aplicación web del lado del cliente, con el uso de frameworks MVC.

2.1 Historia del desarrollo de aplicaciones web

Al igual que internet, el comienzo del desarrollo de las aplicaciones web no se debe a una única persona, hizo falta el desencadenamiento de varios sucesos hasta llegar al modo de desarrollar aplicaciones web como las que tenemos hoy en día. Obviamente para ello, la creación de los lenguajes de programación y los navegadores jugaron un papel relevante, estos sucesos se remontan a la época de los ochenta y noventa:

- 1987. Larry Wall, antes de que internet fuera accesible al público crea uno de los primeros lenguajes de programación "Perl".
- 1994. Se lanza al mercado Netscape Navigator 1.0. el 1 de Octubre de este año se funda World Wide Web Consortium (W3C).
- 1995. Rasmus Lerdorf, permite el despegue del desarrollo de las aplicaciones web, con el lanzamiento del lenguaje PHP. (aplicaciones actualmente reconocidas se han diseñado bajo este lenguaje).
- 1995. (Mayo), SUN Microsistem, anuncia la existencia de JAVA 1.0, soportado por Netscape Navigator a través de los applets. (En este mismo año comienza lo que se conoce como la guerra de los navegadores). En Noviembre de ese mismo año se lanza al mercado Microsoft Internet Explorer 2.0.
- 1996. (Marzo), Netscape Comunications Corporation, lanza al mercado Netscape Navigator 2.0, incorporando nuevas características como elementos de HTML 3.0, frameworks, la capacidad de ejecutar applets programados en JAVA, soporte de JavaScripts, entre otros. Esto permitió un nuevo enfoque para el desarrollo de aplicaciones web, cambiando la forma de texto estático como se venía produciendo las páginas web a un contenido dinámico.
- 1996. Sabeer Bhatia y Jack Smith, lanzan al mercado HOTMAIL (no siendo un desarrollo original de Microsoft quien en años posteriores lo adquiere).
- 1997. Hace su aparición la plataforma Shockwave Flash, adquirido posteriormente por Macromedia y Adobe Flash; convirtiéndose en una plataforma para desarrollar aplicaciones web interactivas.

- 1998. Web The Drudge Report, publicó por primera vez en la historia un informe de noticias antes de que se difundiera en los medios televisivos y la prensa tradicional; marcando un punto de inflexión para los medios de comunicación en línea. Para este año, Google desarrolla su motor de búsqueda en línea, innovando en la forma de indexar las páginas web, facilitando así la búsqueda de información en internet.
- 2001. Sale al mercado WIKIPEDIA como un subproyecto de Nupedia.
- 2003. Se funda Myspace, convirtiéndose años más tarde en la plataforma de impulso de otras aplicaciones web como Youtube, la cual comenzó siendo un módulo adicional para los usuarios de Myspace, para luego convertirse en un sitio web con sus propios derechos.
- 2004. Jhon Battelle y Tim O`Reilly, en su conferencia de web 2.0, se menciona el concepto de la "Web como plataforma". Determinando el camino de las futuras aplicaciones web, es decir, un software que aprovecha las ventajas de la conexión a internet, desviándose del uso tradicional del escritorio.
- 2005. Youtube, se lanza oficialmente, permitiendo a los usuarios compartir videos en línea.
- 2006 (Marzo). Jack Dorsey crea Twitter, un microblogging con sede en San Francisco.
- 2007. Marcado por la aparición de Iphone, quien sin duda estableció un punto de inflexión para las plataformas móviles y las aplicaciones web.

2.2 Generación de los sitios web.

La tecnología empleada para la Web, ha evolucionado desde el año 1996, fecha en la cual se estableció su clasificación; En la actualidad, son cuatro las generaciones que conviven para la creación de sitios Web, siendo únicamente las dos últimas las que se emplean hoy en día para tal fin.

2.2.1 Primera generación.

La primera generación, abarca desde el año 1992 (nacimiento de la web), hasta el año 1994. La creación de páginas web de esta generación, se ve limitada por varias razones tecnológicas; como navegadores poco desarrollados, módems de 2.4 Kbps. Monitores monocromos, entre otros. Las características que presentaron las páginas de esta generación fueron:

- Tiempo de carga rápida: Páginas basadas en texto, con pocas imágenes y sin la presencia de algún recurso multimedia.
- Navegación poco estructurada con poca coherencia.

- Información organizada en una sola página (lo que hacía de esta una página extremadamente larga). Lo que reduce el número de transferencias.
- El texto dentro de las páginas era de una sola línea, desde el principio hasta el final de la página.
- Se empleaban los saltos de línea como separadores.
- Uso de líneas horizontales, para separar secciones de una misma página.
- Uso de listas para organizar la información.
- Escaso uso de enlaces entre páginas del mismo sitio web. Los enlaces existentes en su mayoría eran intradocumentales y enlaces a sitios externos.
- Visualización correcta en cualquier navegador.
- El contenido de las páginas era educativo o científico. Para esta época, pocas empresas se preocupaban por crear su sitio web.

La Figura 1. Muestra un ejemplo en el que se puede apreciar cómo eran las páginas de esta generación. En este, se observan las principales características de esta generación: páginas simples, pocas o a veces ninguna imagen, barras horizontales como separadores, contenido extenso y enlaces intradocumentales. Para finalizar, en esta generación el uso de la web se limitaba a semejar libros o revistas. Se desconocía como aprovechar las ventajas que ofrece la web.

Figura 1. Ejemplo de página web de primera generación.

2.2.2 Segunda generación.

A finales del periodo de la primera generación de sitios web, aparece la tecnología ***Common Gataway Interface (CGI).*** Permitiendo la creación de páginas dinámicas e impulsando el comienzo de una segunda generación de sitios web, que data desde el año 1995, dónde se observa la incorporación masiva de elementos gráficos en las páginas web, lo que inicia la sustitución de palabras por iconos, el color de fondo por imágenes, encabezados por banners[1] y las listas normales por bullets[2] gráficos. Entre sus características principales se destacan:

- Tiempo de carga lento: páginas con exceso de imágenes, colores y animaciones. No se toma en cuenta el usuario final ni el rendimiento de las páginas.
- Empleo de tablas para mostrar datos tabulados.
- Continua el esquema de páginas de arriba hacia abajo.
- La navegación comienza hacer jerárquica, sin manifestarse aún una filosofía de planificación para la navegación de contenidos.
- Aparición de tecnologías multimedia de propietario, que necesitan la instalación de plug-in para su visualización.
- El uso de CGI posibilita el acceso de las aplicaciones web a bases de datos. Las primeras aplicaciones creadas eran pequeñas y sencillas (libro de visitas o algún formulario con información extra). Si se necesitaba almacenar información de forma persistente se utilizaban ficheros en vez de bases de datos.

Aunque en esta generación la mayoría de las páginas continuaba siendo estática, el uso de la tecnología ***CGI*** añadió diversidad de opciones para el diseño de las páginas en esta generación. La prioridad era obviamente el uso de la tecnología; como plano secundario, la legibilidad del contenido y en muchas ocasiones se olvidó el propósito general del sitio web. *La* Figura 2. Muestra un ejemplo de ello.

[1] Formato publicitario de internet, creado con imágenes (gif, jpeg o png), o animaciones.

[2] Bullets (topos o bolos): son características de imprenta, o elementos gráficos que representan una figura geométrica. Se emplean para destacar el comienzo de un párrafo o apartado, sumario o bien las acepciones de un diccionario.

Figura 2. Ejemplo de página web de segunda generación.

2.2.3 Tercera generación.

La tercera generación aparece a mediados del año 1996, este periodo supone la consolidación de la creación de páginas web dinámicas. El protagonismo lo tiene MICROSOFT con la creación de nuevas tecnologías, la primera de ellas ***Internet Database Connector (IDC)***, posteriormente con ***Active Server Pages (ASP)***; a partir de estas otras tecnologías van apareciendo alguna de ellas ***Coldfusion, PHP***, o ***Java Server Pages (JSP)*** basadas en Java. Las páginas de esta generación se siguen observando hoy en día; las características principales que presentan son:

- Tiempo de carga rápida: el tiempo de carga del sitio web se minimiza gracias al uso minimalista de los recursos gráficos, uso de *Cascading Style Sheets* (CSS) y la optimización del código HTML. En esta generación el rendimiento de las páginas web es testeado a través de conexiones de variada velocidad.
- Visualización completa en una sola página, sin tener que realizar desplazamientos.
- Los sitios web se crean tomando en cuenta al usuario final y los objetivos del sitio (ventas, servicios, otros).
- Limitación del número de enlaces.
- Planificación de la navegación (va de una página inicial hasta una final, ofreciendo diversos caminos).
- Incorporación de los principios de navegabilidad y accesibilidad.
- El sitio web es testeado con usuarios reales para su validación.
- Uso de metáforas y temas visuales para captar la atención del usuario y guiarlo para una correcta navegación por el sitio web.

La Figura 3. Muestra un ejemplo de un sitio web de tercera generación.

Figura 3. Ejemplo de un sitio web de tercera generación.

2.2.4 Cuarta generación.

La cuarta y última generación comienza a desarrollarse en el año 1999 y discurre hasta la actualidad. La mayoría de las páginas desarrolladas en esta generación se crean a partir de información almacenada en bases de datos, las características resaltantes en estas son:

- HTML evoluciona, continúan apareciendo tecnologías para la creación de aplicaciones web dinámicas como DHTMLX permitiendo el uso de componentes como elementos separados, o combinarlos en una interfaz basada en AJAX.
- Uso de nuevas tecnologías multimedia como Macromedia Flash, que permite crear sitios web sin necesidad de utilizar HTML.
- Se conforman equipos multidisciplinar para el desarrollo de sitios web (informáticos, expertos en contenidos, diseñador gráfico, otros).
- El aumento del ancho de banda, permite el streaming de video y audio en tiempo real.
- El objetivo de desarrollar un sitio web, se centra en la experiencia completa del usuario, desde que este visualiza la primera página, hasta que abandona el sitio web.
- En los inicios de esta generación, se observa nuevamente el uso a veces excesivo de los recursos gráficos. Se trata de sacar provecho del último pixel para presentar información.

2.3 Aplicaciones web

Una aplicación Web es una aplicación distribuida[3] que realiza una función, generalmente de negocio; basada en la implementación de un conjunto de tecnologías Web tales como: componentes de servidor dinámicos, bibliotecas de clases Java utilitarias, elementos web estáticos (páginas HTML, imágenes, sonidos), componentes de clientes dinámicos (applets, JavaBeans y clases), un descriptor de desarrollo y de configuración de la aplicación Web en forma de uno o múltiples archivos en formato XML [1]. Se caracteriza por presentar una evolución rápida con distribución a gran escala, multi-usuario, multi-lenguaje y multicultural, debe proporcionar seguridad y confidencialidad, permitir el acceso mediante diferentes medios y agentes de usuarios, capaces de procesar gran volumen de información en varios formatos y con varios procesos.

Existe una tendencia creciente a utilizar aplicaciones web dinámicas, debido a que estas adaptan su contenido en función de los datos de entrada del usuario. La idea es permitir que aplicaciones web puedan ser accedidas desde distintos tipos de clientes; es decir, que pueda ejecutarse desde un Smartphone, una Tablet un PC, o desde cualquier navegador HTML estándar. Más aun, que lo que antes se hacía en el servidor, ahora se pueda hacer en el cliente (navegador) y dejar al servidor solo como un API de acceso a la persistencia de los datos, todo esto utilizando una variación del patrón MVC como bien observamos en la Figura 4

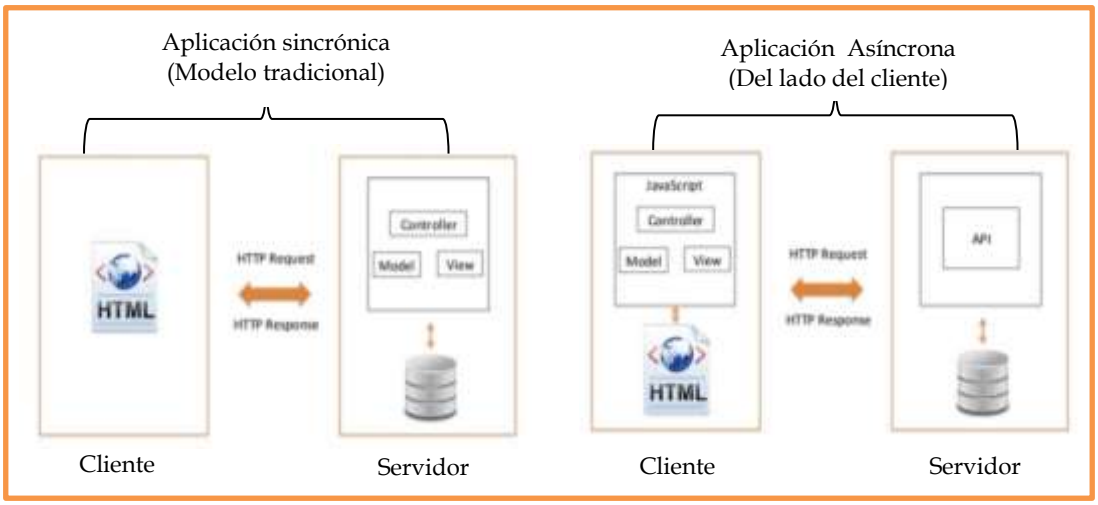

Figura 4. Traslado de la lógica hacia el cliente. [12]

[3] aplicación con distintos componentes que se ejecutan en entornos separados, normalmente en diferentes plataformas conectadas a través de una red.

2.3.1 Funcionamiento de una aplicación web del lado del cliente.

Cuando se transfiere la lógica hacia el cliente gracias a AJAX, el flujo de las aplicaciones web comienza a ser continuo (asíncrono) y aparecen las aplicaciones web de una sola página con la posibilidad de enviar sólo una partes de la página web en cada respuesta. El código JavaScript definido recibiría esta respuesta en el cliente y se sustituirá el contenido de los elementos HTML dentro de la misma aplicación web, como se observa en la Figura 4. Esta primera implementación presentaba un inconveniente y era la manera de mantener actualizada la aplicación web según las interacciones del usuario, ya que con este esquema solo se podía mantener actualizado un elemento por cada petición al servidor, una aproximación a la solución de este inconveniente se observa en la *Figura 5*, donde el cliente solicita una única vez el sitio web completo al servidor, y las siguientes peticiones a este serán solo de archivos de datos ligeros para la actualización de ciertos elementos (uno o varios) marcados dentro de la aplicación web.

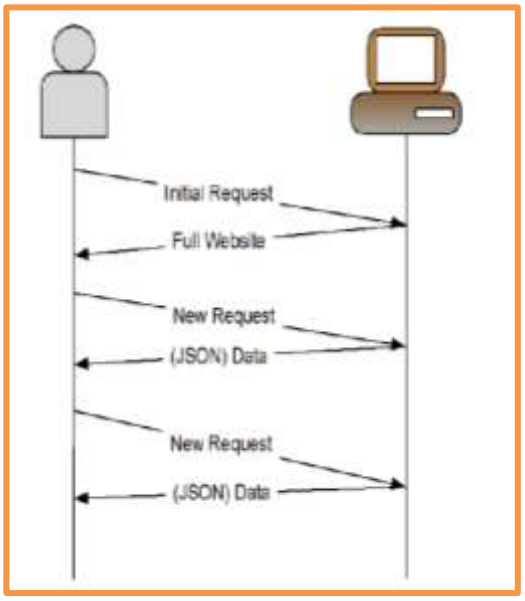

Figura 5. Flujo de una aplicación cliente con una implementación actual. [13]

Con esta implementación actual, es entonces el cliente quien se encarga completamente de la interfaz y la lógica de presentación, dejando al servidor únicamente para dar respuesta a las solicitudes persistentes de datos.

2.3.2 Arquitectura y diseño de una aplicación web.

Definir una arquitectura ayudará a decidir cómo organizar los componentes de una aplicación web, y marcar la pauta para que el conjunto de tecnologías implicadas en ello, se acoplen y den soporte a su desarrollo. Dentro de este ámbito es correcto hablar de una arquitectura de ejecución, que va a permitir a través de diferentes modelos que una aplicación web se ejecute [2].

2.3.2.1 Arquitectura de ejecución.

La Figura 6. Muestra una arquitectura de ejecución generalmente utilizada por las empresas para el desarrollo de aplicaciones web; la cual indica que se deben desplegar diferentes vistas o modelos. Un modelo de arquitectura funcional, donde se describe que es lo que hay que hacer, referenciado por los requisitos del usuario; la arquitectura técnica, donde se detalla el diseño de aplicación y las tecnologías involucradas en su desarrollo; un modelo de arquitectura middleware, en la que se especifica la infraestructura sobre la cual se implantará la aplicación (tipo de servidores, despliegue y diseño de la interacción entre ellos); por último, una arquitectura física (las redes y la forma en que está conectada), que permite especificar como está desplegada la red para establecer la interacción entre las distintas aplicaciones.

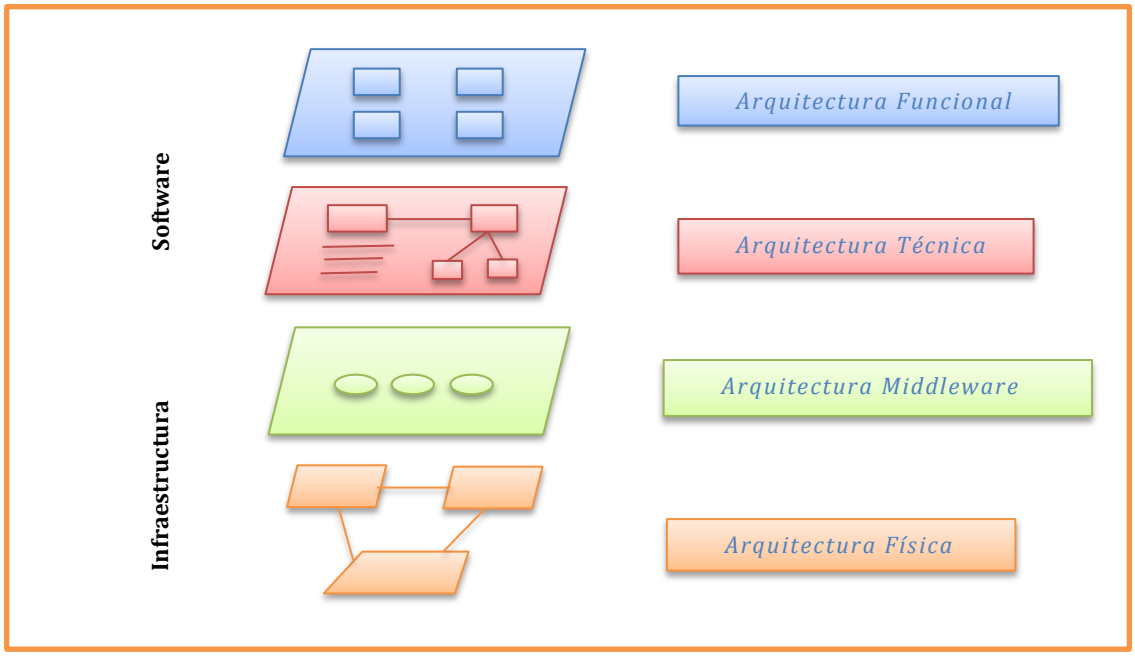

Figura 6. Arquitectura de ejecución de una aplicación web.

En lo que respecta a este estudio, el interés se centra en detallar la parte software de la arquitectura de ejecución, específicamente el modelo de arquitectura técnica, en el cual se emplean tres tipos de diagramas. Un *diagrama de contexto* donde se establece un mapa que relaciona los diferentes sistemas y servicios que posee la

empresa dentro del cual se debe integrar la aplicación web a desarrollar. Por lo general se usa un diagrama diseñado en UML. El siguiente paso sería especificar la aplicación a desarrollar: el *diagrama del sistema*, el cual muestra la relación de los sistemas fusionados, es decir, lo que posee o necesita el usuario y lo que se necesita que la aplicación implemente, especificando en detalle las diferentes funcionalidades de esta. Ver Figura 7. Por último, enfocándose en la aplicación web, en la parte de desarrollo del producto, se elabora un *diagrama de subsistema*, un modelo igualmente descrito en lenguaje UML que relaciona los diferentes componentes dentro de la aplicación web.

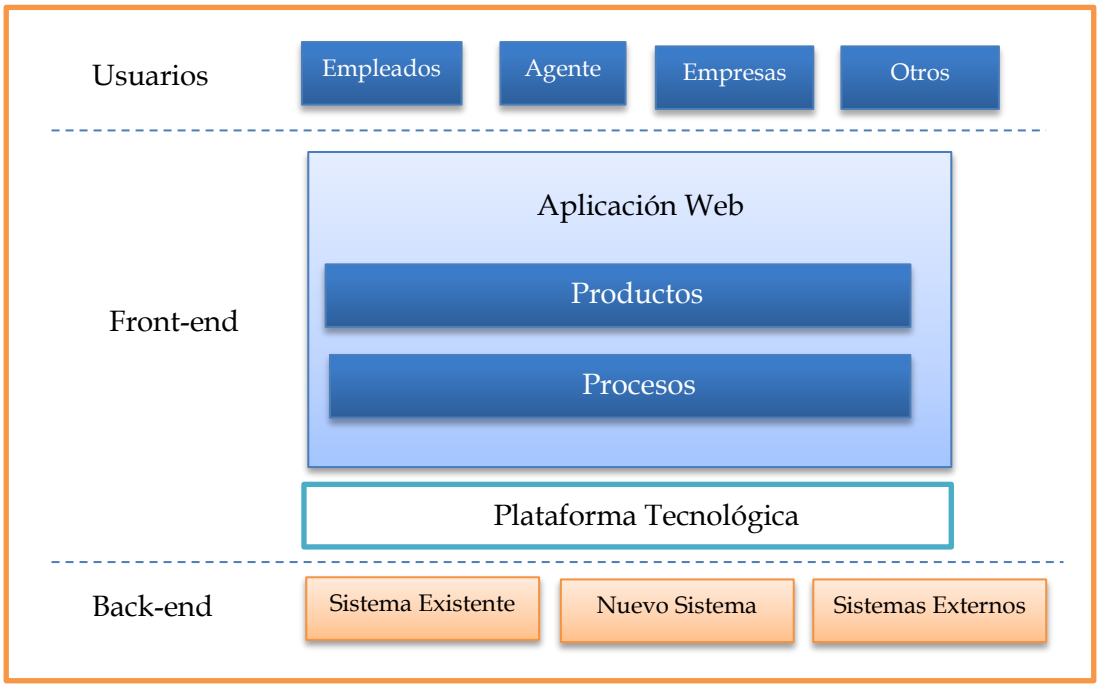

Figura 7. Diagrama de sistema. Arquitectura técnica.

Hasta este punto, todo este proceso forma parte de la definición de una arquitectura para una aplicación web, mas sin embargo, del diagrama de subsistema se desglosa la relación entre los distintos componentes de la aplicación, a partir de entonces este proceso es comúnmente conocido como el diseño detallado de una aplicación web.

En este aspecto, partiendo del modelo conocido de tres niveles, en el que se presenta en primera instancia la capa de presentación, incluyendo el navegador, y el servidor web, que se encargan de dar formato adecuado a los datos; el segundo nivel representado generalmente por un programa o Script; por último el tercer nivel representa los datos que son proporcionados al nivel dos para su ejecución. Evans

(2004) [3] propone una forma de modelar la capa de presentación de una aplicación web, en cuatro capas, que básicamente se han convertido en un estándar.

- ***Interfaz de Usuario:*** cuya función es únicamente la representación de los datos (sin lógica ni eventos).
- ***Capa de Aplicación:*** considerada como el cerebro de la aplicación, se encarga de la siguiente acción a realiza después de cada petición del cliente.
- ***Capa de dominio:*** Involucra la lógica de presentación. (no así la lógica de negocio). En esta capa no se realiza ningún cálculo, solo se encarga de dar formato a los datos para ser presentados en pantalla.
- ***Capa de interoperabilidad de servicio:*** Su función, es la de solicitar los datos y devolverlos a la capa anterior.

La finalidad de trabajar por capas, es mantener un diseño débilmente acoplado y altamente coaccionado. Así, se pueden reemplazar o modificar capas sin que las capas superiores o inferiores se vean afectadas. De forma general, mientras que la aplicación ejecuta las reglas de negocio, la capa de presentación se encarga únicamente de dar formato para mostrarlo al usuario, permitiendo crear aplicaciones con diferentes vistas. *Ver* Figura 8. Ahora bien, todo esto supone un problema de acoplamiento, resuelto con la incorporación de un patrón de diseño denominado MVC.

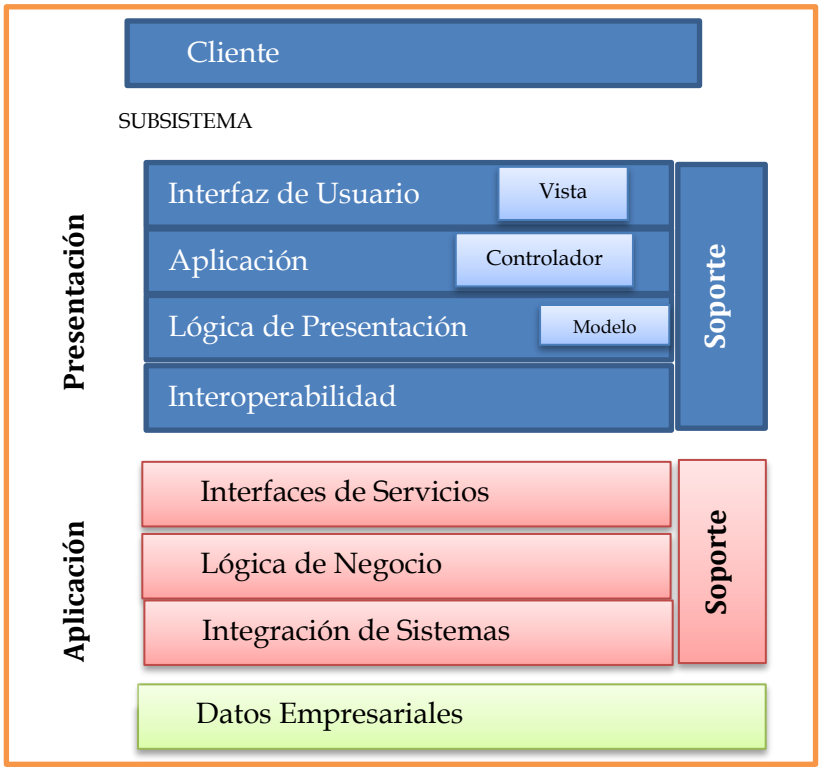

Figura 8. Diagrama de Subsistema-Arquitectura Técnica.

MVC es específicamente definido como un patrón de diseño que proviene de las siglas en inglés (Model View Controller / Modelo Vista Controlador). Una correcta implementación y uso de este patrón logran separar la lógica de negocios de la interfaz de usuario, resultando en aplicaciones donde es más fácil modificar tanto la apariencia visual como las reglas de negocio sin que la otra se vea afectada [4].

Al utilizar un patrón MVC, se divide la aplicación en tres partes:

- ***El modelo:*** que contiene la lógica de presentación de la aplicación; puede estar compuesto por un conjunto de clases Java (POJO4 o bien EJB5)
- ***La vista:*** muestra la información que necesita el cliente, a través de un conjunto de páginas Web dinámicas, que son interpretadas como simples páginas HTML. Actualmente existen diversidad de framework para generar estas páginas Web en diferentes formatos.
- ***Controlador:*** donde se recibe e interpreta la interacción del cliente. El controlador, actúa sobre el modelo y la vista para adecuar los cambios de estado en la representación interna de los datos y su visualización.

Este patrón de diseño, ya sea aplicado a un modelo de dos o tres capas, se viene implementado desde mediados del año 2005 en desarrollo de aplicaciones, ya sea con framework de terceros o framework JSR oficiales. Es a comienzos del presente año, que han surgido diferentes representaciones, cada una con diferente aplicabilidad; se habla ahora del meta patrón MV*. Los desarrolladores comenzaron a darse cuenta de que, para las empresas "aplicaciones inteligentes" conectadas, no era más que un patrón MVC anidado. El servidor (controlador) realizaba la lógica de negocio de la información de base de datos (modelo) mediante el uso de objetos de negocio, y después de pasar esa información a una aplicación cliente (una "vista"). El cliente recibe esta información desde el servidor, y lo trata como su propio "modelo" personal. El cliente entonces actúa como un controlador adecuado, ejecutando la lógica, y enviando la información a la vista que se muestra en la pantalla. La Figura 9. Muestra un ejemplo de cada patrón su diferente aplicabilidad.

[4] Plain Old Java Object: Objetos Java tradicionales.

[5] Enterprise JavaBeans: API para el desarrollo de soluciones empresariales.

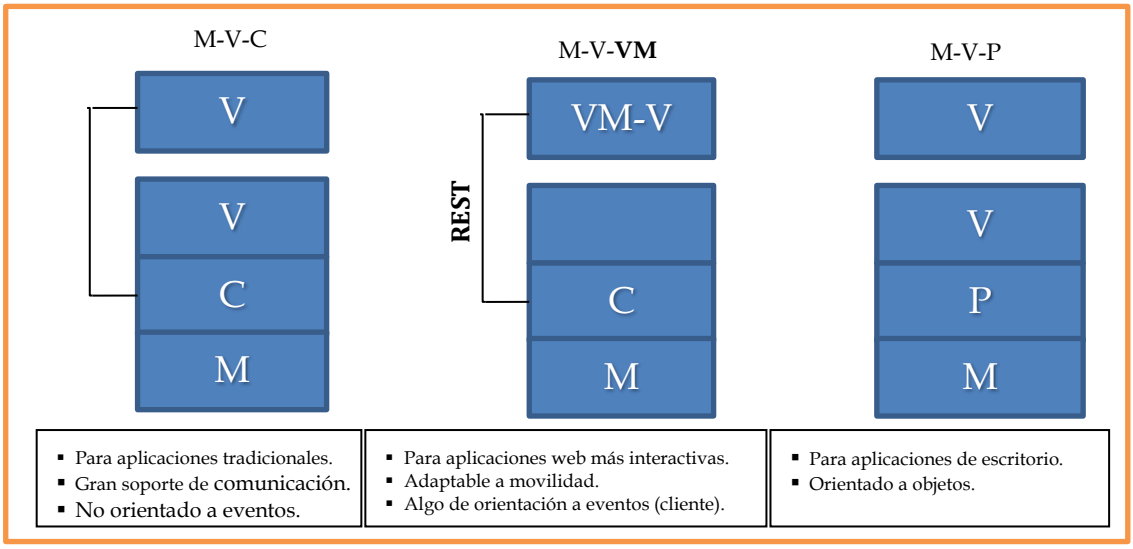

Figura 9. Meta Patrón MV*

2.4 Tecnologías para el desarrollo de Aplicaciones web del lado del cliente

El método de desarrollo web ha llegado a un punto donde la exigencia en ofrecer aplicaciones con una experiencia de usuario aceptable, que respondan al ritmo evolutivo de la tecnología y necesidades de un usuario en constante movimiento con acceso a la información desde cualquier lugar o dispositivo, pero que además puedan ser desarrolladas en un tiempo aceptable, con un diseño escalable y un código reutilizable; originó la división del esquema de desarrollo de una aplicación web de forma tal, que como ya se ha descrito, nos encontramos con un desarrollo Front-end que involucra toda la parte de programación de diseño (toda la parte visual) encargada de la interacción con el usuario; agrupando para ello un conjunto de tecnologías que trabajan del lado del navegador y un desarrollo Back-end involucrando tecnologías que corren del lado del servidor.

En virtud de que el objeto de estudio de esta investigación se centra en dar a conocer los frameworks MVC que trabajan del lado del cliente, esta sección estará dedicada a describir las tecnologías que hacen posible maquetar la estructura semántica del contenido (HTML), codificar el diseño en hojas de estilo (CSS) y agregar la interacción con el usuario (JavaScript).

2.4.1.1.1 HTML (*HyperText Markup Language*)

HTML es un lenguaje de marcado utilizado para describir las páginas web. estandarizado por el *World Wide Web Consortium* (W3C) y el *Web Hypertext Application Technology Working Group* (WHATWG); este lenguaje, puede ser creado desde un simple editor de texto, debido a que se encuentra contenido dentro de un archivo de

texto, con la peculiaridad de que su extensión es *.html.* La sintaxis que se utiliza para crear un documento de este tipo, es el uso de etiquetas [5].

Un documento HTML Se encuentra conformado por un conjunto de etiquetas simples de inicio y fin cada una de ellas con una función específica, por ejemplo la etiqueta "<*p*>" indica el comienzo de un párrafo y </*p*> para indicar el fin de ese párrafo, lo que se vería del siguiente modo: " aquí el párrafo ". La estructura general de un documento HTML debe seguir un esquema estricto, donde se indique el de inicio del documento, encabezado, cuerpo y fin del documento. En la Figura 10. Se observa la estructura general que se debe seguir.

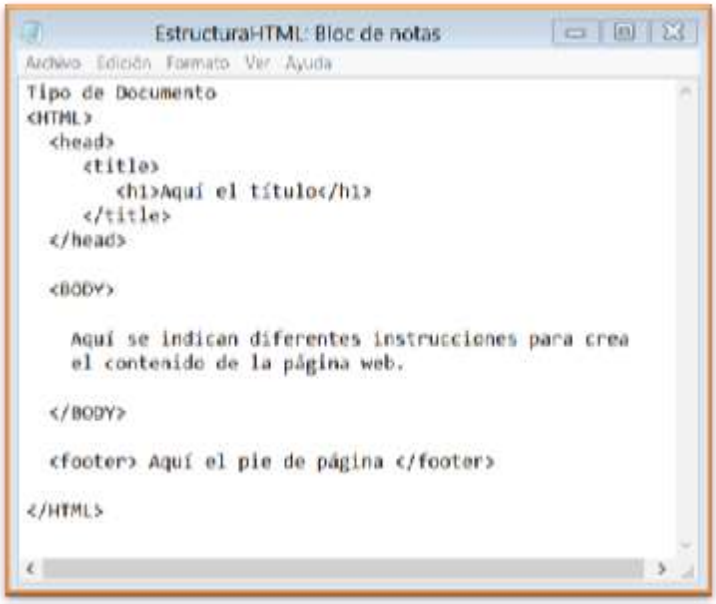

Figura 10. Estructura básica de un documento HTML

Cabe destacar que este lenguaje ha existido desde los principios de la web, por lo que existen diferentes versiones. *Ver Tabla 1. Versiones de HTML* pero han sido sus últimas versiones, específicamente la versión ***HTML5*** la que ha permitido una completa implementación de funciones como *geolocalización*, *dibujo vectorial*, *guardar datos en el disco del usuario*, *insertar audio y video*, entre otras cosas. Todas ellas para un desarrollo íntegramente front-end.

Tabla 1. Versiones de HTML

Versión de HTML	Año de creación
HTML	1991
HTML+	1993
HTML 2.0	1995
HTML 3.2	1997
HTML 4.0.1	1999
XHTML 1.0	2000
HTML5	2012
XHTML5	2013

2.4.1.1.2 CSS

Es una tecnología desarrollada por el W3C, con la finalidad de separar la presentación semántica de una aplicación de su estructura. CSS es definido como hojas de estilo en cascada creadas en un documento con extensión ***.css***. Estos estilos fueron creados en primera instancia para la versión 4 de HTML, con el fin de simplificar el tamaño de estos archivos, pero a lo largo del tiempo, el uso de hojas de estilo fue ofreciendo más ventajas como la definición del "*estilo visual*" de una aplicación completa sin la necesidad de hacerlo etiqueta por etiqueta. Actualmente con la última versión de esta tecnología (CSS3), se pueden crear bordes redondeados, sombras, degradados, fondos múltiples, entre otros, sin la necesidad de imágenes cortadas, sólo usando código.

La sintaxis de una hoja de estilo se basa en reglas de texto simple que describen el aspecto de los elementos dentro de una página. Estas reglas CSS se componen de dos elementos: un selector de tipo que indica a cuales etiquetas dentro del documento aplicar el estilo y una declaración de estilo que se encapsula entre paréntesis para contener una propiedad con su respectivo valor. El modo de vincular una hoja de estilo a un archivo HTML se observa en la Figura 11

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8">
5  <title>Inclusión de CSS</title>
6    <link rel="stylesheet" href="css/normalize.css">
7    <link rel="stylesheet" href="css/style.css">
8    <link rel="stylesheet" href="css/bootstrap.css">
9  </head>
10 <body>
11 </html>
```

Figura 11. Inclusión de CSS en HTML

2.4.1.1.3 JavaScript

Es un lenguaje de programación de propósito general orientado a objetos que permite la ejecución de acciones dentro de las páginas web sin alguna compilación, puesto que funciona del lado del cliente; siendo por consecuencia el propio navegador el encargado de interpretar y ejecutar sus instrucciones. Este lenguaje es soportado por la mayoría de los navegadores, e impulsa el surgimiento de otras tecnologías como AJAX al ser unido con lenguajes del lado del servidor como PHP. JavaScript presenta una sintaxis muy similar a la de C, C++ y JAVA que es integrada generalmente dentro de la etiqueta <*BODY*> de un documento HTML, a través del uso de la etiqueta <*script*>, aunque puede también ser utilizando dentro de un fichero externo usando el siguiente código:

<script type="text/javascript" src="codigopersonalizado.js"></script>. *La* Figura 12 Muestra un ejemplo de esta integración.

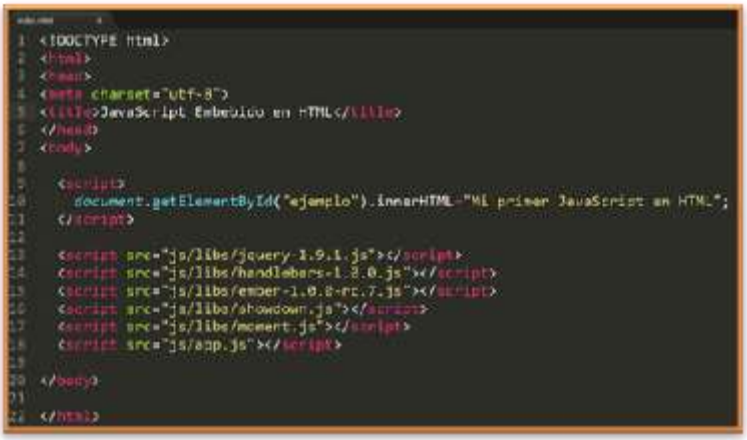

Figura 12. JavaScript embebido en HTML.

Uno de los recientes usos de JavaScript, ha sido la creación de frameworks, los cuales añaden el componente de interactividad y conexión al servidor. Ahora es posible comunicarse con el back-end sin recargar la página con el usando AJAX, recibir esos datos y cambiar el diseño entero de la aplicación web. Más adelante en la sección número tres, se expondrá en detalle algunos de estos frameworks que permiten el desarrollo de aplicaciones en un escenario front-end.

3 Frameworks MVC

Un framework es una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación web. Es considerada una aplicación genérica incompleta y configurable a la que el desarrollador le puede añadir las piezas restantes para construir una aplicación web completa.

Un framework MVC involucra la filosofía de trabajo de separación de funciones dentro de esa estructura configurable, obteniendo los beneficios que ese patrón de diseño otorga. Las ventajas que ofrece el diseñar aplicaciones web utilizando un framework MVC consiste principalmente en obtener un desarrollo estructurado con código reutilizable, aportando un desarrollo rápido debido a que se disminuye el esfuerzo de trabajo al aprovechar funcionalidades ya implementadas,

En este aspecto JavaScript, se considera una solución interesante para la creación de framework ya que permite incorporar a ellos un sin fin de funciones diferentes, entre las cuales predomina el ofrecer una interfaz única para todos los navegadores, corrigiendo fallos e incompatibilidad entre ellos, añadiendo funcionalidades de alto nivel especialmente en el DOM. Es por ello que en esta sección se describirán los frameworks de JavaScript que proveen aplicabilidad para manejar formularios de adquisición y presentación de datos en pantalla de manera interactiva implementando el patrón MVC. Pero antes de mencionarlos y hacer una introducción a ellos es necesario familiarizarse con los términos básicos referidos al uso de estos frameworks.

- ***Plantilla:*** describe la interfaz de usuario de la aplicación. Cada plantilla está respaldada por un modelo, por lo que esta es cambiada automáticamente si el modelo cambia. En las versiones de HTML estas plantillas suelen contener *expresiones* que contienen información sobre el modelo de la plantilla como por ejemplo *{{firtsname}}*; {{*outlet*}} (marcador que indica al **router** la posición para una plantilla adecuada, en base a una URL actual); y *componentes*.
- ***Router***: es un enrutador que traduce una dirección URL en una serie de plantillas añadidas, cada una de ellas apoyada por un modelo como ya se ha indicado.
- ***Componentes***: Son etiquetas HTML personalizadas, cuyo comportamiento implica el uso de JavaScript. Y su apariencia depende del uso de múltiples plantillas. Estos componentes permiten crear controles reutilizables que pueden simplificar las plantillas de la aplicación eliminando aquellas plantillas repetidas.
- ***Modelo***: un modelo es un objeto que almacena el "estado persistente". En este caso, las plantillas son responsables de mostrar el modelo al usuario

convirtiéndolo en HTML. En la mayoría de los frameworks, estos modelos se cargan a través de un API JSON HTTP.

- ***Route***: objeto que indica a la plantilla la ruta del modelo a mostrar.
- ***Controladores***: un controlador, es un objeto que almacena el estado de la aplicación, para recuperar si fuera el caso el estado de esta. Una plantilla puede poseer de forma opcional un controlador, además de un modelo y recuperar las propiedades de ambos.

Ahora bien, al iniciar una búsqueda sobre los frameworks MVC basados en JavaScript, se obtienen una diversidad de opciones. Por lo que en este punto se describen un cúmulo de estos que como desarrollador se pueden implementar:

3.1 Angular.js

Es un framework opensource desarrollado por Google para crear Webapps en lenguaje cliente con JavaScript ejecutándose a través de SPA[6] , extendiendo el código HTML con etiquetas propias, tales como *ng-app, ng-controller, ng-model, ng-viewEs* [6].

Angular es reconocido como un framework sólido, que implementa el patrón MVC, separando en su totalidad el modelo, la vista y el controlador, con inyección de dependencia que nos permite crear pequeños módulos, está orientado a código testable, e iguala el desarrollo front-end con el back-end.

3.1.1 Principales características.

Entre las características principales de este framework se encuentran:

- Permite extender HTML con tags personalizados, definir y vincular (data-bind) variables vista/controlador, consultas AJAX con peticiones HTTP, sistema óptimo de templating, manipulación de datos en JSON, inyección de dependencias, deep linking, formularios de validación, desacoplamiento del DOM de JavaScript, filtros, unit testiting, otros
- Es compatible con los navegadores de última generación (Chrome, Firefox, Safari, Opera, Webkits, IE 9+) y se puede hacer compatible para Internet Explorer 8 o anterior mediante varios hacks.
- Flexibilidad en conexiones REST.
- Disminución de código en data-bind.
- Soporte de animaciones a través de los eventos ***hide, show, enter, leave y move.***

[6] single-page application, traducido como aplicación de una sola página.

3.1.2 Estructura de ficheros y organización

El framework Angular.js se carga con la inclusión de un único fichero ***http://ajax.googleapis.com/ajax/libs/angularjs/1.0.4/angular.min.js.*** Realmente Angular.js no indica (como en otros frameworks) cuál sería la estructura de carpetas a seguir. Pero por similitud hacia otros frameworks es deducible establecer su estructura de la siguiente manera:

Como carpetas raíces estarían:

- ***css/***: en la cual se encuentran los estilos CSS de la interfaz.
- ***data/***: incluye lo ficheros JSON de datos. (No serían necesarios si se invocara a servicios web).
- ***img/***: la carpeta *img* contiene las imágenes de los libros.
- ***lib/***: incluye las librerías JavaScript comunes a todos los proyectos.
- ***src/***: contiene todo el código JavaScript de la aplicación.
- ***index.html***: es la sección principal, en código HTML que carga toda la aplicación, (la página de inicio de la aplicación).

La aplicación a desarrollar se incluiría dentro de la carpeta raíz ***src/***: y se desglosaría en las siguientes subcarpetas:

- ***config/***: incluye las constantes de configuración como rutas, URL de conexión a WS, entre otros.
- ***controllers/***: subcarpeta que contiene los controladores de la App.
- ***directives/***: en esta se encuentran lo componentes o etiquetas extendidas de HTML.
- ***filters/***: en esta subcarpeta se encuentran los filtros para búsquedas de objetos.
- ***lib/***: en esta se incluyen las librerías JavaScript propias de la aplicación.
- ***models/***: incluye los modelos de la aplicación.
- ***services/***: incluye los servicios de la App. En esta sección por lo general se reflejan las llamadas a los webservices.
- ***views/***: en esta subcarpeta se incluyen las vistas y parciales de vistas de la App.
- ***app.js***: en esta se encuentra el archivo ***.js*** que da inicio a la aplicación. Incluyendo en este el routing.

Sin lugar a duda, Angular.js marca la transición entre páginas webs y aplicaciones web sin recarga de página, extendiendo las limitaciones de HTML. Por supuesto, existen otros frameworks basados en JavaScript como podrían ser Backbone.js y Ember.js que comparten la misma filosofía que Angular.js.

3.1.3 App.ng

Es la inicialización de una aplicación en Angular.js. Definida en el ***index.html*** (indicado en la sección de carpetas raíces); es en este documento HTML es donde se ejecuta el framework Angular.js y todos las clases JavaScript que se necesiten. Ahora bien, para que este framework se ejecute con efectividad, se debe indicar la directiva ***ng-app***[7] para referenciar el nombre de la aplicación. Correspondientemente, se cargan los JavaScript de inicialización de la aplicación "***app.js***" y los controladores de la aplicación (***NombreDetailController.js*** y ***NombreListController.js***). Finalmente, en el ***index.html*** se define un contenedor, en el cual los controladores irán cambiando las vistas mediante la directiva ***AngularJS ng-view: <div ng-view> ***.

3.1.4 Routing

El enrutamiento (Routing), se encuentra configurado en el archivo ***app.js*** contenido en la carpeta ***src***; es la inicialización de la Aplicación, donde se definen las rutas (mediante la indicación "***/#***"), la URL y el par controlador-vista (mediante la instrucción "*$routes.when*") que muestra la App.

3.1.5 Vistas

En Angular.js las vistas se configuran en un documento ***.html*** contenido dentro de la subcarpeta ***view***. En este se configuran los valores de las variables para los controladores, los atributos que reciben los modelos y vistas mediante la directiva "***ng-model",*** aunque también es admisible el uso de código JavaScript descrito entre doble llaves ***"{{código JavaScript}}"*** y generar directamente la vista de la ejecución de dicho código.

En este documento, también se utiliza la directiva ***ng-repeat*** que permite repetir una etiqueta donde fue definida tantas veces como objetos hayan. Siendo beneficioso el uso de esta directiva para mantener la filosofía de un código "limpio".

3.1.6 Controladores

En Angular.js los controladores de definen en un archivo ***.js*** contenido dentro de la subcarpeta ***controllers.*** En este se pueden utilizar variables ya definidas con anterioridad en la vista o bien crear nuevas para que esta las utilice. En este archivo, se hace uso de las directivas "***$scope"*** y ***"$http".***

- ***$http:*** a través de esta directiva se realizan peticiones AJAX y se obtienen sus datos a través de ***$routeParams.***
- ***$scope:*** mediante el uso de esta directiva se tiene acceso a las variables de las vistas. Se puede definir variables para que la vista las utilice a través de

[7] Directiva de autoarranque de una aplicación. una App debe poseer solo una directiva de este tipo en el documento HTML ya que esta designa la raíz de la aplicación. por lo que siempre estará indicada en la página raíz.

$scope.var1 = "variable"*** y visualizarlas en la vista con ***({{var1}})*** y viceversa, o bien utilizar variables ya definidas en la vista con ***$scope.orderField = "variable";

3.2 Backbone.js

Backbone.js es uno de los frameworks de la serie de JavaScript para la creación de aplicaciones web MVC del lado del cliente, es un framework relativamente ligero que se puede probar fácilmente con herramientas de terceros, como jazmín o QUnit. Backbone.js es mantenido por un número de contribuyentes, sobre todo: Jeremy Ashkenas, creador de CoffeeScript, Docco y Underscore.js. Independientemente de la plataforma o dispositivo de destino, este framework pretende ayudar a organizar la estructura de la aplicación, simplificar la persistencia del lado del servidor, desvincular el DOM a partir de datos de la página, trabajar los datos del modelo y routers de manera concisa, proporcionar DOM, el modelo y la sincronización de la colección. [7].

3.2.1 Principales características

- Incorporación de los componentes básicos del patrón de diseño MVC: Modelo, Vista, Colección, router. Aplicando en este aspecto su propio meta modelo de MV *.
- Es un framework utilizado por grandes empresas como SoundCloud y Foursquare para la construcción de aplicaciones poco comunes.
- Los eventos son impulsados por la comunicación entre la vista y el modelo. Haciendo sencillo añadir detectores de eventos a cualquier atributo de un modelo, dando a los desarrolladores el control preciso sobre los cambios que deseen en la vista.
- Soporta tanto, enlaces de datos a través de eventos manuales o bien, una biblioteca de observación de Clave-Valor (KVO) por separado.
- Apoyo a interfaces de RESTful, lo que permite que se pueda vincular fácilmente el modelo a un Back-end.
- Los prototipos pueden ser definido con palabras claves determinadas por el desarrollador.
- No es compatible con modelos anidados, aunque existen plugins que pueden ayudar en este caso.
- Convenios claros y flexibles para aplicaciones de estructuración. Backbone.js no obliga al uso de todos sus componentes y puede trabajar con sólo los necesarios.

3.2.2 Estructura y organización

Al igual que en Angular.js, en Backbone.js no existe una forma general de organizar un proyecto, aunque existen muchos recursos que ofrecen posibles estructuraciones de

código de una App, de forma no intrusiva sin modificar el core de Backbone.js, una de ellas es un proyecto opensource llamado ***Backbone Boilerplate***. En la Figura 13. Se puede observar un ejemplo de esta estructura de código para Backbone.js.

Backbone-js	*//carpeta principal de la App*
css	*//hojas de estilo*
data	*//"Server" de datos simulados (JSON)*
contenido	
1	
2	
3	
contenido.json	
img	*//imágenes de la App*
src	*//código JavaScript de la App*
lib	*//"includes"*
backbone-min.js	
underscore-min.js	
models	*//objetos modelos*
model.js	
views	*//objetos vistas*
view.js	
app.js	*//main y router de la app*
Index.html	*//HTML principal que carga la App*

Figura 13. Estructura App en Backbone

3.2.3 El modelo

En Backbone.js, el modelo es el corazón de la aplicación, y no sólo contendrá los datos y su lógica, sino que también será el encargado de notificar a las vistas de los cambios en estos datos mediante eventos. Estos, son agrupados en la "*Collections*" de Backbone, como objetos que tiene acceso a todos los métodos de las "*Collection*" de *Underscore.js*, una librería de utilidades que es el único requerimiento de Backbone, aparte del jQuery.

Los modelos pueden pedir datos al server mediante la función `fetch` de Backbone.Model, que enlazará el server con la URL predefinida en el propio modelo o en la *collection* a la que pertenece, y concatenando automáticamente el id del modelo que estamos "fetcheando". Por este motivo se recomienda asignar los nombres de los ficheros JSON igual a los ids de los objetos a los que representan, para adecuarse así al comportamiento por defecto de los modelos Backbone.js.

3.2.4 La vista

Las Vistas de Backbone.js no incluyen un motor de plantillas. Esto significa que se pueden generar etiquetas HTML manualmente en las vistas y luego añadirlas al DOM mediante jQuery sin ningún problema, o bien, es posible utilizar alguno de los motores de templating de JavaScript que existen.

En la Backbone puede observar el uso de *Underscore.js* que es una biblioteca de utilidades de JavaScript que funciona como enlace entre jQuery y Backbone para trabajar con una serie de funcionalidades comunes como mapa, seleccionar, invocar, entre muchas otras. A parte de esto, esta librería es utilizada para renderizar las vistas, a través de la función *render* del objeto Backbone.View. Por lo que *Underscore.js* debe ser añadida al atributo HTML de la vista generada.

3.2.5 Routing

El Routing en Backbone es muy similar al de Angular.js y a muchos otros frameworks web. Se pueden definir rutas con parámetros, que estos parámetros sean opcionales, de hecho la sección de la ruta puede ser opcional.

Cabe destacar que Backbone.js soporta por defecto el *pushState de HTML5* simplemente añadiendo una directiva, para que las rutas no empiecen siempre por un "#" gracias a la *HTML5 history API*, aunque el soporte de este elemento en diferentes navegadores este aún por comprobar.

3.3 Spine.js

Spine.js es considerado un framework ligero para la creación de aplicaciones web en un entorno real [10]. Aunque está dentro de la clasificación de un framework MVC, no cumple con esta concepción en su totalidad, ya que al principio estructura la aplicación bajo el enfoque de este patrón de diseño pero luego se aparta un poco de ello. Está escrito en CoffeeScript pero también se puede utilizar JavaScript; por lo que existen varias formas de trabajar con Spine.js para la creación de aplicaciones web, estas son:

- *Utilizando JavaScript:* descargar los archivos JavaScript e incluirlos en el código HTML.
- *Usando Node.js*[8]*, CoffeeScript y Hem:* mientras que Spine.app genera la estructura de directorios para la aplicación *"Hem"* será utilizado para gestionar dichos directorios de forma dinámica, y mostrarlos al usuario.

[8] Entorno de programación back-end basado en JavaScript para crear aplicaciones de red rápidas y escalables. Ideal para aplicaciones en tiempo real de datos intensivos que se ejecutan a través de dispositivos distribuidos.

- *A través de la integración con Rails*[9]*:* mientras que *Rails* hace una gran API back-end para el almacenamiento de datos y autenticación, Spine proporciona a la aplicación con una experiencia de usuario front-end.

3.3.1 Principales características

- Posee una ligera implementación del controlador (basado en la API de Backbone).
- La capa del modelo es lo bastante completa además incluye ORM.
- Ejerce una comunicación asíncrona hacia el servidor.
- Incorpora adaptadores de almacenamiento local de Ajax y HTML5.
- Incorpora una extensión móvil, que le permite crear aplicaciones móviles y PhoneGap fácilmente.
- Incorpora un gerente de dependencia y servidor de desarrollo denominado "*Hem*".
- Funcionamiento en varios navegadores (IE, Safari, Chrome, Firefox, otros).
- Posee una biblioteca que contienen las clases principales como *model* y *controller.*

3.3.2 Modelo

En Spine.js, los modelos son creados como cualquier clase en CoffeeScript, mediante la extensión "`Spine.Model`", por lo cual se les puede añadir y extender propiedades o bien, estos pueden ser creados como subclases. Se hace uso de la función `configure ()` para pasar argumentos al modelo que luego serán llamados para inicializar la instancia de una clase. *La* Figura 14. Muestra un ejemplo de ello.

Concretamente, los modelos en spine.js se utilizan para el almacenamiento de datos de la aplicación (*models*), así como cualquier lógica asociada a estos datos.

```
Var Contacto = Spine.Model.sub ();

Contacto.configure ("Contacto", "nombre", "apellido"); //atributos del modelo
```

Figura 14. Ejemplo de creación de modelo en Spine.js (en lenguaje JavaScript)

[9] Framework de aplicaciones web de cliente en código abierto escrito en el lenguaje de programación Ruby bajo el esquema MVC.

3.3.3 Vista y plantillas.

Spine.js, no posee ningún *Widgets UI* para generar sus vistas, estas son hechas desde el mismo framework y se componen por simples fragmentos de código HTML creando así la interfaz de la aplicación; por lo que es necesario el uso de plantillas JavaScript del lado del cliente para garantizar que la aplicación sea totalmente asincrónica y sensible. Estas plantillas son construidas en el *Hem*, las más utilizadas en Spine.js son Jade y ECO.

3.3.4 Controladores

La función que cumplen los controladores en Spine.js al igual que en otros frameworks es entrelazar los distintos componentes de la aplicación, respondiendo a eventos del DOM, generando plantillas y manteniendo la vista y el modelo sincronizados.

3.4 Batman.js

Batman.js es un framework para la creación de aplicaciones cliente. Al igual que Spine.js; está escrito y desarrollado con CoffeeScript, permitiendo a su vez el uso de JavaScript. Este framework se estructura bajo el patrón de diseño MVC, y su objeto central es "***Batman.app***", un espacio de nombres para los modelos y vistas con el que se da inicio a la aplicación [11].

Una aplicación batman.js se presenta en una sola carga de la página, seguido de peticiones asíncronas de diversos recursos dependiendo de la interacción del usuario. La navegación dentro de la aplicación se puede ejercer mediante *pushState*, o recargarse mediante *hash-bang*.

3.4.1 Principales características

- API fuertemente inspirado en Rails.
- Acciones del controlador enrutables.
- Las vistas están creadas en HTML puro.
- Trabaja en conjunto con herramientas construidos sobre Node.js y Cake.
- Puede ser ejecutado tanto en el navegador o sobre node.js.
- Es compatible solo con Chrome, Safari 4, IE 7 y Firefox 3.

3.4.2 Modelo

En Batman.js se utiliza el objeto *Batman.model* para representar los datos en la aplicación y proporcionar una interfaz fluida que pueda ser ejecutada fuera del back-end. Claramente, este objeto solo se limita a definir la lógica que rodea la carga y el salvado de la aplicación, pero no el mecanismo real para hacerlo. (Ese trabajo se ejecuta en una subclase llamada *Batman.StorageAdaper*).

Las operaciones dentro del *Batman.model* son asíncronas, lo que significa que como última instancia se hará el llamado a una función *Node*, una vez se haya completado la operación; por ejemplo cuando toda la respuesta HTTP se ha recibido desde el servidor, por lo que suelen pasar algunos segundos después de la llamada original a la función de la operación.

3.4.3 Vistas

Las vistas son representadas con el objeto Batman.View, entre sus varias funciones se encuentra la creación de componentes configurables y reutilizables que puedan ser creados dentro de las plantillas.

El flujo básico para la generación de vistas en Batman.js seria de la siguiente manera: Un usuario activa un evento generando una ruta del mismo, la cual es remitida en una acción al controlador, por lo que este crea un archivo HTML con el mismo nombre de la acción que se ha ejecutado (la plantilla) contenido en la carpeta app/views. Haciendo uso del sistema data-bind, solicitará datos al usuario y los reservará para un próximo evento. Todo este proceso se lleva a cabo dentro del controlador, que por medio de la creación de nuevas instancias de Batman.View aloja los archivos HTML de estas vistas dejando indicado el objeto Render () y la inserta en el *node*; así, para cuando la propiedad HTML se establezca hacer el llamado a la plantilla correspondiente ubicarla dentro del *node* y generar la ruta hacia ella.

3.4.4 Controladores

Los controladores en Batman.js está definidos en una clase base de la que descienden todos los controladores de la aplicación esta se denomina ***Batman.Controllers,*** la cual es responsable de la ejecución de las acciones es decir, de la activación de las solicitudes de datos al modelo, crear vistas, o redirigir acciones en función de los cambios de estado del navegador. Cada controlador dentro de la aplicación es independiente y ejecuta solo una instancia que le permita recuperar datos además de presentar una vista para mostrarlos.

3.5 Ember.js

Ember.js forma parte del grupo de framework MVC para el desarrollo de aplicaciones web cliente, siendo en particular el caso de estudio en este trabajo, se dedicará el punto número cuatro para a su especificación.

3.6 Cuadro comparativo de frameworks MVC

En esta sección, la idea es resumir las características generales de los frameworks MVC algunas de ellas indicadas anteriormente, para poder establecer una comparación entre los frameworks descritos en este estudio.

Se debe acotar en este aspecto que las características presentes en cada framework, engloban un sinfín de rasgos específicos, y si el deseo de un desarrollador es conocer a fondo los detalles de algún framework MVC para el desarrollo de aplicaciones web cliente en particular, se recomienda dedicar tiempo a revisar tanto el código fuente como las características del mismo para estudiar que tan bien se puede adaptar a sus necesidades.

A continuación se describen brevemente las características generales contenidas en la Tabla 2. Comparación de características de los framework MVC descritos. Que no se hayan descrito con anterioridad.

- ***Observable:*** Es una clase *(mixin)* que da a los objetos la capacidad de notificar sobre los cambios en sus propiedades.
- ***View Bindings:*** Indica como los enlaces se muestran y se recogen por la parte del usuario. Específicamente cuando se permite que las vistas para los objetos se actualicen automáticamente de acuerdo a algún cambio observable generado por la interacción del usuario.
- ***Dos enlaces:*** permite que la vista pueda ejercer cambios en el objeto observable de forma automática.
- ***Vistas parciales:*** vistas que incluyen otras vistas.
- ***Vistas de listas filtradas:*** que el framework posea vistas que puedan filtrar objetos de acuerdo a criterios previamente establecidos.
- ***Capa de presentación web:*** Dentro de esta característica, lo que se evalúa es que el framework no posea widgets de estilo nativo, la idea es que su conceptualización este centrada en el desarrollo del lado del cliente.
- ***Vinculación UI:*** establecer si el framework MVC posea un enfoque declarativo para la actualización automática de la capa vista, para cuando se realicen cambios en el modelo base.
- ***Composición de vistas:*** en esta característica se evalúa si el framework en realidad permite la creación de código reutilizable, planteándose si en este se pueden disponer varias vistas para crear una nueva.

Finalmente en la Tabla 2, s*e* establece un cruce entre los framework MVC para el desarrollo de aplicaciones cliente, utilizados en este estudio con las características mencionadas anteriormente.

Tabla 2. Comparación de características de los framework MVC descritos.

Características	Angular.js	Backbone.js	Ember.js	Spine.js	Batman.js
Observable	✓	✓	✓	x	✓
Routing	✓	✓	✓	✓	✓
View Bindings	✓	✓	✓	✓	✓
Dos enlaces	✓	X	✓	x	✓
Vistas parciales	✓	X	✓	x	x
Vistas de listas filtradas	✓	X	✓	x	✓
Vinculación UI	✓	X	✓	x	x
Capa de presentación web	✓	✓	✓	✓	✓
Composición de vistas	x	x	✓	x	x

4 Caso de estudio Ember.js

En esta sección se definirá en detalle ¿Qué es Ember.js? y como este implementa el el modelo MVC. Se especificarán las sintaxis y conversiones necesarias para trabajar con este framework y finalmente se creará una aplicación básica para demostrar su uso.

4.1 ¿Qué es Ember.js?

Ember.js un framework opensource para el desarrollo de aplicaciones web del lado del cliente, fue creado a partir de conceptos introducidos por framework de aplicaciones nativas tales como "cacao" y "Smalltalk"; permite organizar todas las interacciones directas que puede realizar un usuario, ayuda a los desarrolladores de aplicaciones web para ampliar los límites de lo que es posible desarrollar para la web, manteniendo el código fuente de la aplicación estructurado, y forzando el modelo de desarrollo para ser tan orientado a objetos como sea posible.

Ember.js soporta enlaces, un mecanismo en el que los cambios en una variable propagan su valor en otras variables, y viceversa; propiedades calculadas, lo que permite marcar funciones como propiedades que se actualizarán de forma automática sobre plantillas dependientes generadas igualmente de forma automática, garantizando de este modo que la interfaz gráfica de usuario (GUI) se mantendrá al día cada vez que ocurran cambios en los datos subyacentes, todo esto a través del patrón de diseño MVC. [8].

4.1.1 Principales características

Las características resaltantes de este framework MVC son:

- Ember.js tiene como objetivo general eliminar las formas explicitas del comportamiento asíncrono en una aplicación.
- Posee políticas de seguridad claramente definidas; incorpora una lista de tráfico exclusivo para avisos de seguridad. Anunciado para el pasado 25 de Julio de 2013 la liberación de Ember.js 1.0 RC6.1, RC5.1, RC4.1, RC3.1, RC2.1 y RC1.1. (versiones de seguridad que se ocupan de un posible problema de seguridad XSS).
- Contiene un elaborado sistema para crear, administrar y representar una jerarquía de puntos de vista que se conectan al DOM del navegador. (Las vistas son responsables de responder a eventos de usuario, como clics y arrastres, de igual modo la actualización de los contenidos del DOM cuando los datos subyacentes de la vista cambian).
- Muchos conceptos en Ember.js, como enlaces y propiedades calculadas, se han diseñado para ayudar a controlar el comportamiento asincrónico. (por lo que al comenzar a utilizar este framework, será importante familiarizarse

con estos conceptos y con algunas conversiones de nomenclatura a la hora de hacer el llamado a objetos en la aplicación).

- Posee "Generación Implícita de Código" que permite cargar dinámicamente los objetos en memoria.

4.1.2 Modelo

En la mayoría de aplicaciones Ember.js, los modelos son manejados por "Ember datos", una biblioteca construida con y para Ember.js, diseñados para recuperar archivos de un servidor, o bien, hacer cambios en el navegador. Ofrece muchas de las comodidades que se encontrarían en un servidor ORM[10] como *ActiveRecord*, pero está diseñado específicamente para el entorno único de JavaScript en el navegador. En *Ember datos*, se pueden cargar y guardar documentos y relaciones servidas sin ninguna configuración previa a través de una API REST JSON, siguiendo ciertas convenciones.

4.1.3 Vista

Las vistas en Ember.js normalmente sólo se crean por las siguientes razones: cuando necesite manipulación sofisticada de eventos de usuario y si desea crear un componente reutilizable. A menudo, estos dos requisitos estarán presentes al mismo tiempo.

Si el desarrollador, se encuentra en una de estas opciones y desea finalmente crear una vista, cabe mencionar, que como Ember.js se encuentra integrado con el motor de plantillas *Handlebars*, se podrá hacer uso del helpers {{parcial}}, permitiendo definir vistas parciales e incluirlas posteriormente en una plantilla. Algunas de las funciones más utilizadas que este helpers permite para generar las vistas dentro de un plantilla, serian:

- ***{{render "nombreTemplate"}}:*** genera una plantilla asociada a un objeto vista con los datos del modelo del controlador de dicha vista.
- ***{{view objectView}}:*** genera un objeto vista, que puede tener una plantilla asociada o generarlo dinámicamente a partir de los datos de su controlador.
- ***{{outlet}}:*** Por defecto, genera con la vista asociada al controlador actual.

De cualquier modo, la función que cumple la vista en una aplicación en Ember.js es traducir los eventos primitivos del navegador, en eventos que tienen significado para la aplicación.

[10] Object Relational Mapping: técnica de programación para crear una base de datos virtual orientada a objetos, sobre una base de datos relacional, proporcionando el uso de características como polimorfismo y herencia.

4.1.4 Routing

En ember.js, cada uno de los estados posibles de una aplicación, está representado por una URL. Estas URL de respuestas de estado se encuentran encapsuladas por un manipulador de ruta para generar una respuesta clara y precisa a los cambios de estado.

Dado que una aplicación puede tener uno o más controladores de ruta activos, que pueden cambiar bien sea porque el usuario interactúe dentro de una vista generando un evento que implique el cambio de la URL o porque el usuario cambie la dirección de la página manualmente. Estos controladores están programados para reaccionar en cualquiera de los caso de la siguiente manera:

- Condicionalmente redireccionar a una nueva URL.
- Actualizar un controlador de modo que represente un modelo particular.
- Cambiar la plantilla en la pantalla, o colocar una nueva plantilla a una toma existente.

Es significativo resaltar, que como se mencionó en las características, Ember.js trabaja con generación implícita de código, por lo que para este aspecto, cada vez que se define una ruta, se realizara una serie de intentos para ubicar la ruta correspondiente (View, Controller y las clases de plantilla con los nombres correspondientes de convenciones de nomenclatura). Si no se encuentra la aplicación de cualquiera de estos objetos, Ember.js generará los objetos correspondientes en la memoria.

4.1.5 Controladores

"Un controlador es un objeto que almacena el estado de una aplicación. Una plantilla puede tener opcionalmente un controlador, además de un modelo, y puede recuperar las propiedades de ambos" [9]. Los controladores en Ember.js permiten que se realcen los modelos con lógica de visualización o presentación, sin ser necesario que la plantilla guarde estas características de realce o propiedades del controlador en el servidor.

En esta circunstancia, si se colocan estas propiedades en el controlador, se logra separar la lógica relacionada con el modelo de datos de la lógica de presentación, haciendo fácil las pruebas unitarias del modelo. Y es que en la práctica, Ember.js crearán un controlador de una plantilla para toda la aplicación, pero el modelo del controlador puede cambiar durante la vida útil de la aplicación sin necesidad de que la vista conozca sobre ello.

En este punto, debemos comprender que en ember.js las plantillas obtienen las propiedades que realzan un modelo, de los controladores. Lo que significa que las

plantillas conocen sobre los controladores y los controladores conocen los modelos, pero un modelo no reconoce lo que los controladores están modificando, y por ende el controlador tampoco sabe cuáles de sus propiedades se están presentando. *Ver* Figura 15.

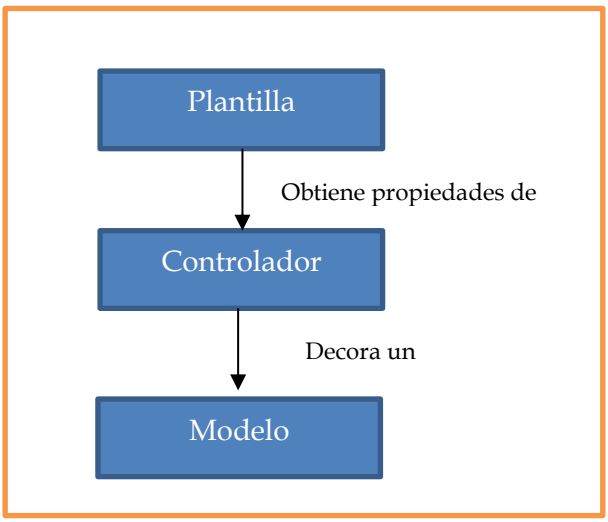

Figura 15. Acoplamiento plantilla, modelo, controlador en Ember.js

4.2 Conversiones de nomenclaturas

Como se mencionó en el punto anterior, antes de iniciar el desarrollo de una aplicación con Ember.js, es necesario familiarizase con algunos términos y conversiones de nomenclaturas para las rutas, controladores y plantillas, que se deben utilizar para evitar repeticiones de términos en el desarrollo del código. La *Tabla 3* muestra las conversiones de nomenclaturas reconocidas por ember.js.

4.3 Sintaxis, funciones e instrucciones en Ember.js

En esta sección se mostrarán elementos y funciones utilizada por Ember.js para la creación de una aplicación web, a través de imágenes detalladas su sintaxis, para luego, a través del desarrollo de una aplicación simple, ilustrar el uso las principales funciones que este framework ofrece y que como desarrollador se pueden aprovechar para la creación de un proyecto web. Debido a que Ember.js presenta una curva de aprendizaje empinada, el objetivo será iniciar al programador en el uso de este framework MVC para el desarrollo front-end y sea tomado en cuenta ya que ofrece grandes beneficios para los desarrolladores que logren dominarlo.

Tabla 3. Conversiones de nomenclaturas en Ember.js

Elemento	Sintaxis	Objetos buscados por Ember.js /Descripción

La Aplicación	`App = Ember.Application.create()`	Ember.js ubica la plantilla de la aplicación y la adapta como plantilla principal. Si `App.ApplicationController` es proporcionado, ember.js establece una instancia de este y lo establece como el controlador de la plantilla. Los objetos que buscara serán:
		`App.ApplicationRouter`
		`App.ApplicationController`
		`La plantilla de la aplicación`

Rutas Simples	`App.Router.map(function){`
	`  This.route('imágenes')`
	`});`
		Cada ruta tiene su controlador y su plantilla asociada. Si un usuario navega por una ruta, los objetos que ember.js buscaría serian:
		`App.ImagenesRouter`
		`App.ImagenesController`
		`La plantilla imágenes`

Segmentos Dinámicos	`App.Route.map(function(){`
	`  This.resource('post',`
	`  {path:'/posts/:post_id'});`
	`});`
		En este caso, la ruta es Post, y ember.js buscará los objetos
		`App.PostRoute`
		`App.PostController`
		`La plantilla Post`

Nesting	`App.Route.map (function() {`
	`this.resource('posts', function (){`
	`  this.Route (. 'favoritos');`
	`  this.resource ('post');`
	`  )};`
	`)};`
		Se pueden anidar rutas bajo un resource. (Un resource marca donde debe comenzar la ruta anteponiendo un punto (.) a esta.). A pesar de que el recorrido de la ruta 'posts' esta anidiado (que es solo parte de una ruta), Ember.js seguirá buscando los elementos:
		`App.PostRoute`
		`App.PostController`
		`La plantilla Post`

4.3.1 Usos de instrucciones en la plantilla.

Las plantillas de la aplicación en Ember.js se estructuran de la siguiente forma*: ver* Figura 16.

```
3  <header>
4    <h1>Primeros pasos con Ember.js</h1>
5  </header>
6
7  <div>
8    {{outlet}}
9  </div>
10
11 <footer>
12     @2013 Publicación de Aymar García, Inc
13 </footer>
```

Figura 16. Estructura de una plantilla en Ember.js.

Como se observa la plantilla está compuesta por tres simples secciones, un "<*header*>" que encapsula el encabezado, un contenido de introducción o bien un conjunto de enlace de navegación. <*footer*> que muestra información al pie de la plantilla y <*div*> cuyo contenido varía dependiendo de la navegación del usuario por el sitio web. En esta sección es común emplear el elemento {{outlet}} para marcar al **router** desde donde se puede mostrar el contenido de alguna otra plantilla seleccionada.

Si no se están utilizando herramientas de construcción para la creación de plantillas, se puede definir la plantilla principal de la aplicación dentro de un HTML colocando dentro de una etiqueta <script> el siguiente código: *Ver* Figura 17. Esto permitirá que Ember.js compile de forma automática la plantilla para cuando el **router** la necesite.

```
13  <body>
14    <scripttype="text/x-handlebars"> Hello,<strong>
      {{nombre}} {{apellido}}</strong>!</script>
15  </body>
16  </html>
```

Figura 17. Uso de plantilla automática handlebars.

Otras veces solo se necesita mostrar parte de la plantilla, para estos casos Ember.js permite el uso de **condicionales**, un ejemplo de esta instrucción se observa en la Figura 18.

```
7   {{#ifpersona}}Bienvenido nuevamente, <b>
    {{persona.nombre}}{{persona.apellido}}</b>!
8   {{/if}}
```

Figura 18. Uso de condicionales para la plantilla.

Recordando que cada plantilla es manejada por un controlador, a estas se le pueden añadir atributos asociados a su controlador, como por ejemplo enlaces a imágenes, enlaces a atributos de una clase de un elemento HTML o bien enlaces a recursos básicos. Algunas sintaxis de ejemplo se reflejan en las Figura 19, Figura 20 *y* Figura 21.

Figura 19. Enlace a una imagen

Figura 20. Enlace a una ruta

{{#linkTo}} es determinado como un ayudante que trabaja como un apuntador hacia una ruta activa previamente creada; en este caso *'imágenes'*, este añade tambien la caracteristica de "*clase activa*" al enlace que apunta, según el ejemplo de la Figura 21. Seria a la palabra *Imagenes*.

Figura 21. Enlace al atributo de una clase

4.3.2 Declaración de rutas

El *Router* en ember.js es el encargado de crear la ruta, mostrar las plantillas y cargar los datos cuando se inicia una aplicación. Entre sus funciones resaltantes está la de comparar la URL actual de las plantillas para informar de su estado (*activo o inactivo*). La sintaxis para definir una ruta simple, segmentos dinámicos y rutas anidadas sobre un recurso, se encuentran en la en la *Tabla 3.*

Por lo que ahora, tomando como ejemplo el código mostrado en la Figura 22, se explicara en detalle su funcionamiento.

```
App.Router.map(function() {
  this.resource('comentar', { path: '/comentar/:comentar_id' }, function() {
    this.route('editar');
    this.resource('commentarios', function() {
      this.route('nuevo');
    });
  });
});
```

Figura 22. Uso de Route en Ember.js

En la Figura 22 se observan elementos que vale la pena definir antes de explicar el funcionamiento de su código, estos son:

- *Resource ( )*: Se denomina recurso al principio de un nombre de una ruta, controlador, o plantilla.
- *_id*: el elemento *_id* es un método estándar utilizado para la serialización de un segmento dinámico.
- *Un segmento dinámico*, es la parte de una dirección URL que comienza con dos puntos (:) y es seguido de un identificador (por ejemplo _id).
- *"Path: ":* Es utilizado para anidar las rutas (si la ruta del *Path:* posee el mismo nombre del recurso, esta instrucción puede ser obviada).
- *"{path: '/comentar/:comentar_id'}"*: En esta línea se indica que por ejemplo, para un comentario con *id*=2, la ruta debe iniciar del siguiente modo, /comentar/2.

En Ember.js no se pueden anidar rutas directamente, es permitido anidar rutas bajo un recurso, por lo que el código mostrado por la *fig. 18* demuestra el modo de hacer esto. El resultado de la ejecución de este código sería la creación de cinco rutas: *App.IndexRoute* de nombre *Index, App.ComentarIndexRoute* cuyo nombre seria *Comentar.Index, App.ComentarEditarRoute* de nombre *Comentar.Editar, App.CommentariosIndexRoute* de nombre *Comentarios.Index* y por último *App.CommentariosNuevoRoute* con el nombre *Comentarios.Nuevo*. Todas estas con su respectivo controlador y plantilla.

4.3.3 Declaración y uso de un componente.

Los componentes en Ember.js son elementos personalizados que permiten definición de etiquetas propias de HTML para una aplicación las cuales puedan ser utilizadas posteriormente sobre JavaScript. La creación de estos componentes en Ember.js se ajusta estrictamente a la especificación de componentes creados para la web.

El uso de estos elementos personalizados, es una idea que la W3C está trabajando actualmente, por consiguiente, el hecho de que en Ember.js ya se encuentren componentes regidos por estos estándares se plantea como una ventaja, puestos que los componentes creado con anterioridad dentro de las aplicaciones, puedan ser migrados y reutilizados sin mayor dificultad en otros framework si el desarrollador así lo desea.

La forma de declarar un componente se muestra en la Figura 23.

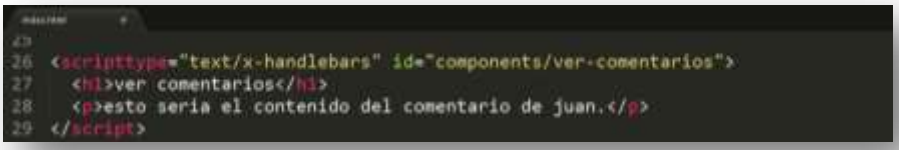

Figura 23. Declaración de un componente

Siguiendo el ejemplo de laFigura 23, se observa que para definir un nuevo componente y hacer uso de el a través del elemento personalizado *{{ver-comentarios}}*, se debe crear una plantilla cuyo nombre comience con "***componentes/***". Para ello, la plantilla handlebars se incluye dentro de una etiqueta <*script*> de HTML asignandole el valor *id* de la nueva plantilla asociada. *(components/ver-comentarios)*. Dentro de este <*script*> se indicaran los parametros de esa nueva plantilla.

Una vez realizada esta inscripción, se puede hacer uso del elemento personalizado en este caso *{{ver-comentarios}}* dentro de la aplicación. Figura 24.

Figura 24. Uso de un elemento personalizado.

Esta declación le indica a ember.js, mostrar el componente *{{ver-comentarios}}* en la plantilla *index*. Observe que se hace uso de la instrucción *{{#each}}*, la cual representa un **bucle**, que para este caso en particular, se emplea para asignar un **item** a cada componente y enlazalo al modelo correspondiente en cada **loop.** Figura 25

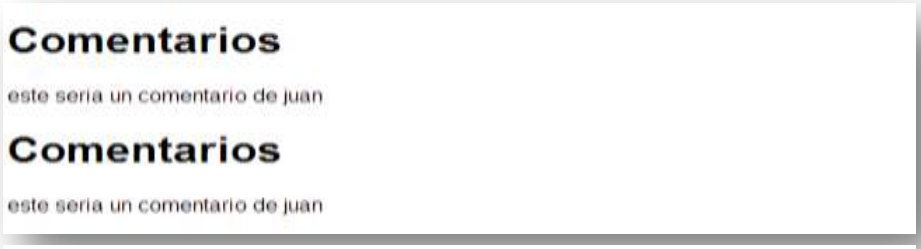

Figura 25. Vista de un componente en la plantilla index.

En esta vista, se genera una repetición del componente, por ejemplificar el comportamiento básico de un elemento personalizado. La idea es pulir el código a través de una estructuración de propiedades al momento de definir las plantillas (por ejemplo: titulo: "comentario" , autor: "juan", fechadepublicación: (`date'), comentario: "aquí el comentario de juan", otras). Pero por defecto, en Ember.js, un componente no puede acceder a las propiedades de la plantilla a la que se encuentra enlazado directamente, para ello debe hacer de la propiedad de la plantilla origen una propiedad de la plantilla *componets/*, esto se logra asignando el valor una propiedad de la plantilla origen a una propiedad de la plantilla *componets/* al momento de invocar al elemento personalizado. Figura 26

```
12 <script type="text/x-handlebars" id="components/ver-comentarios">
13     <h1>componente: {{title}}</h1>
14     <p>comentario de juan.</p>
15   </script>
16
17   <script type="text/x-handlebars" id="index">
18     <h1>Titulo de la Plantilla: {{title}}</h1>
19     {{ver-comentarios title=title}}
20 </script>
```

Figura 26. Uso de propiedades de una plantilla en elementos personalizados.

Al ejecutar el codigo se obtiene el siguiente resultado: Figura 27

Título de la Plantilla: Comentario

Componente: Comentario

Comentario de juan.

Figura 27. Representación del uso de las propiedades de una plantilla en un componente.

4.3.4 Configurar un Controlador

Un controlador en Ember.js contiene las propiedades de la aplicación que no son necesarias que se guarden en el servidor. El controlador es un objeto que responde a los eventos generados desde la plantilla y recibe el modelo que representa desde el controlador de la ruta. Existen dos formas de trabajar con los controladores dentro de Ember.js, una de ellas cuando dentro de la plantilla no se indica alguna acción a la que se deba dar respuesta en pantalla, en este caso, el controlador de la plantilla solo actúa como un paso más entre el modelo y la plantilla (no habrá que configurar el controlador), si por el contrario dentro de la plantilla de la aplicación se indica una acción ({{#if}} o {{action}}) es el momento para que el controlador entre en acción (se debe configurar). En este caso, en Ember.js se diferencias dos tipos de controladores: un contralor de un único objeto definido con *Ember.ObjectController* y un controlador para matrices *EmberArrayCantroller*.

Si desde la plantilla se solicita una propiedad al controlador, lo primero que este hará será verificar si tiene la propiedad definida dentro de él. Si ese es el caso, simplemente dará respuesta a la solicitud, enviando su valor actual. Si por el contrario, no posee la propiedad solicitada, este consultará con el modelo haciendo la petición de la propiedad correspondiente.

El modo de configurar un *ObjectController* sería el siguiente:

1. Extender la clase ObjectController para implementar las funciones que se necesiten. *Ver* Figura 28.

```
App.NotasController = Ember.ObjectController.extend ({
    siEditando: false,
     editar: function()
     {
      this.set('siEditando',true);
     },
     Edicion: function()
     {
      this.set('siEditando',false);
      //para salvar en el servidor
      this.get('store').commit();
     }
});
```

Figura 28. Extendiendo la clase ObjectController

2. Definir en el Router el modelo del controlador que se utilice.
3. Se debe indicar ahora, que aspecto debe tomar la aplicación cuando se active la función editar. Para ello agregamos una plantilla externa (como vista parcial) y se le asignan plantillas predefinidas por ember.js (aunque se permite el uso de externas) a

las propiedades implicadas en el cambio a través de *Data Binding*. Como se observa en la Figura 29. Lo que indica es que las propiedades de título y texto tomaran en aspecto de la plantilla a la que se han vinculado, una vez se active la función editar.

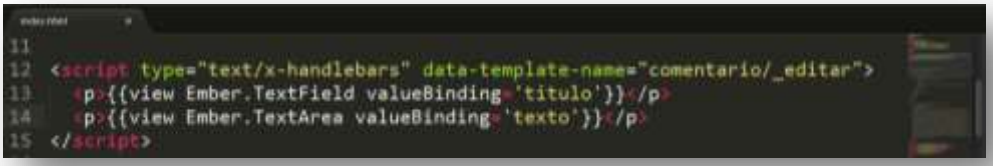

Figura 29. Configurar la vista para las propiedades a editar

4. Como último paso, se ubica en la plantilla principal de la aplicación la condición que solicita el cambio en la plantilla y se agrega el elemento que ejerce el cambio, en este caso la visa parcial creada en la Figura 29. Configurar la vista para las propiedades a editar Cuando se cumple la condición "siEditando". *Ver* Figura 30.

Figura 30. Agregar el elemento de cambio.

Por su parte, *ArrayController* proporciona una manera de publicar una colección de objetos y unirlos fácilmente a la colección *handlebars* con el uso del helper {{#each}}, Ember.CollectionView u otros controladores. La ventaja de utilizar un *ArrayController* es que solo se tiene que configurar los enlaces una sola vez en la vista, y si se desea cambiar el contenido (o lo que se muestra en la vista) solo se debe modificar el contenido de la propiedad en el controlador.

Para configurar un *ArrayController* se pueden seguir los siguientes pasos: Si se supone que se desea configurar una lista de entradas de un blog entonces:

1. Se debe extender la clase *ArrayController* en la plantilla de ember.js. Ver Figura 31.
2. Definir en el Router el modelo del controlador que se utilice.
3. App.Post.FIXTURES es una colección estatica de publiaciones (usada a modo de ejemplo)que se debe crear y llenar para manejar las variables o elementos a modificar con la vista parcial.

4. Se agrega la vista parcial "index" a la plantilla handlebars, y se hace uso del helper {{#each}} para mostrar cada entrada del blog. Figura 32

```
App.IndexController = Ember.ArrayController.extend({
  contentBinding: Ember.Binding.oneWay('App.Post.FIXTURES')
});
```

Figura 31. Extendiendo ArrayController

```
<script type="text/x-handlebars" data-template-name="index">
  <ul>
    {{#each controller}}
      <li><h3>{{titulo}}</h3>
        <time {{bindAttr datetime="publicado"}}>{{publicado}}</time>
        {{{content}}}
      </li>
    {{/each}}
  </ul>
</script>
```

Figura 32. Vista parcial y uso de {{#each}}

4.3.5 Definición de un modelo

Los modelos en Ember.js contienen elementos que necesariamente se deben guardar en el servidor. Ember.js posee un *Store* como repositorio para contener los modelos cargados, el cual es el responsable de la recuperación de los modelos que todavía no se han cargado. Generalmente, la interactuar se realiza con los modelos directamente, no con el *Strore*. Sin embargo, es necesario indicarle a Ember.js que se desea incluir Ember data para gestionar los modelos. En este caso, basta con definir una subclase de *DS.Store* dentro de la aplicación. *Ver* Figura 33.

```
App.Store = DS.Store.extend({
    revision: 12,
});
```

Figura 33. Definición de un modelo en Ember.js

Si se desea personalizar un modelo para su posterior inclusión o no en el Store de Ember.js la sitaxis seria la reflejada en la Figura 34.

Figura 34. Definición de un modelo personalizado.

En un modelo, se pueden definir atributos, que bien pueden ser utilizados como una propiedad, esto se logra utilizando la instrucción "***DS.attr".*** *Ver* Figura 35.

Figura 35. Asignación de atributos para un modelo.

Nota: *Como se ha explicado anteriormente, la mayoría de los modelos en ember.js dependen de la biblioteca* ***ember-data*** *(una tienda de modelos para trabajar con ember.js). La cual se encuentra en proceso de desarrollo, por lo que se recomienda que hasta que la versión 1.0 de este framework no sea publicada, trabajar con modelos de datos inferior a la versión 13 de la biblioteca ember-data.*

4.4 Configuración del entorno de desarrollo

En este punto se describirá paso a paso como configurar el entorno de trabajo para el framework Ember.js. El cual está compuesto por un navegador web para visualizar y monitorear el avance del proyecto (recomendable *Chrome* y *Firefox* que actualmente involucran un entorno para desarrolladores), un editor de texto para la creación y edición del código como *Notepad++*, *Sublime Text*, *TextMate, Brackets* o si se desea, cualquier **IDE** con soporte para desarrollo web. Por último la figura principal en este caso, el framework ember.js, el cual viene organizado en una carpeta comprimida denominada "*starter-kit*" disponible para su descarga en desde la página oficial de este framework[11].

[11] http://emberjs.com/.

4.5 Componentes básicos del *"Kit de inicio"*

Una vez se haya descargado el archivo comprimido "*starter-kit*", el siguiente paso será descomprimirlo en cualquier lugar, generando de este modo la carpeta "*starter-kit*" (que bien se puede renombrar si se desea) la cual contiene los componentes básicos para iniciar un desarrollo con el framework Ember.js. Si todo va bien hasta ahora, se debe observar sin ningún problema dentro de esta carpeta un documento de texto con el nombre "*TODO*", un documento HTML denominado "*index*" como archivo índice donde se mostrará la información en el navegador del usuario, y dos carpetas: "*css*" y "*js*".*Ver* Figura 36.

Figura 36. Componentes básicos del kit de inicio de Ember.js.

Como podemos observar los nombres de las carpetas son lo bastante específico para referirse a su contenido, por lo que la carpeta css servirá para almacenar todos los archivos de hojas de estilos, que en un principio el *"kit de inicio"* incluye dos de muchos disponibles para el desarrollo de aplicaciones web, estos son "*normalize*" y "*style*". Por su parte en la carpeta "*js*" se encuentra principalmente el archivo "*app.js*" y una carpeta denominada "libs" destinada al almacenamiento de las librerías externas, en la cual el "*kit de inicio*" incorpora *"handlebars"*, "*ember*" y "*jQuery*".

- ***normalize.css:*** Una moderna herramienta incluida en HTML5 para restablecer y normalizar hojas de estilo css, permitiendo que los navegadores rendericen los elementos de forma más consiente de acuerdo a los estándares modernos. Dirigiéndose solo a los estilos que se necesiten normalizar.
- ***Style.css:*** Archivo destinado para alojar los estilos a utilizar en la aplicación.
- ***App.js:*** Archivo JavaScript que contiene la estructura de la aplicación en ember.js.
- ***handlebars.js:*** Es una biblioteca de plantillas integradas que se actualizan de forma automática cuando los datos subyacentes cambian.
- ***ember.js:***
- ***jquery.js:*** Biblioteca cuya principal característica es permitir cambiar el contenido de una página web, sin la necesidad de recargarla. Consiste en un único fichero JavaScript que contiene las funcionalidades comunes del DOM, eventos, efectos y AJAX.

5 Creando una aplicación con Ember.js

En esta sección se realiza una implementación básica del framework Ember.js para desarrollar una aplicación del lado del cliente, la cual consiste en la elaboración de un blog, en el cual un usuario podrá gestionar un blog.

Para esta práctica se hará uso de:

- El Editor de texto *Sublime Text2*[12].
- El navegador *Chrome,* debido a que ofrece a los desarrolladores una herramienta para la depuración de aplicaciones creadas en Ember.js que se puede incorporar a modo de extensión "*Ember inspector*".
- *starter-kit de Ember.js.*

Una vez definido y configurado el entorno de trabajo, se ejecuta el archivo *index* ubicado dentro del *starter-kit*, para comprobar que en efecto está listo para comenzar a trabajar, ofreciendo la siguiente vista en el navegador. *Figura 37.*

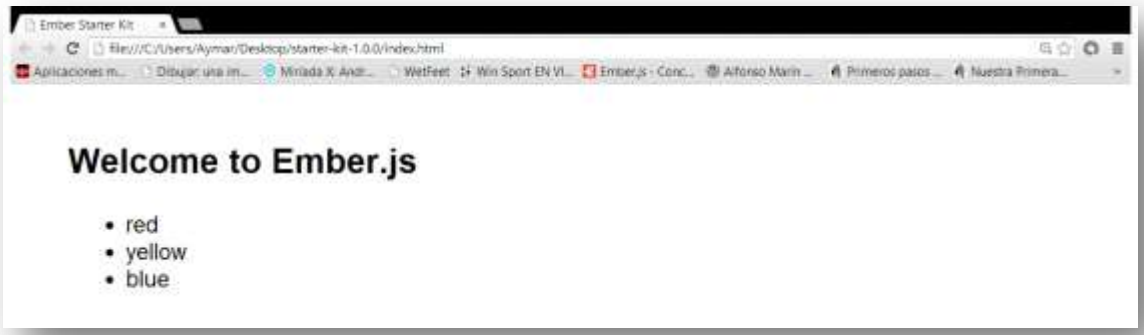

Figura 37. Vista index del Starter-Kit en el navegador.

Ahora bien, en primer lugar se debe definir el espacio de nombres con el que Ember.js va a trabajar, para ello es necesario hacer una instancia de "***Ember.Application***" y asignarle una variable global que por lo general suele ser "***App***", Aunque Ember.js no limita el nombre de esta, solo se debe cumplir que comience con una letra mayúscula.

Por consiguiente, ubicamos en el *kit de inicio*, el archivo *app.js*, que de forma predeterminada trae consigo funciones vacías para que se configuren las rutas a trabajar, pero con el objetivo de realizar esta aplicación guiada desde cero, se elimina esta estructura dejando solo la línea uno, que define el espacio de nombre de la aplicación.

[12] http://www.sublimetext.com/2

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

Luego se comienza a editar el archivo *index.html* con el fin de dar inicio a la construcción de la aplicación. Se borra el contenido de los <*script*> que vienen por defecto a excepción de los enlaces a las librerías. Una vez realizado esto, se procede a añadir en la sección <*head*> la hoja de estilo *bootstrap.css* y junto a las otras librerías *Showdown.js* y *Moments.js* las cuales permiten parsear texto en formato Markdown[13] y modificar la hora del sistema a un formato tradicional respectivamente.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Ember Starter Kit</title>
  <link rel="stylesheet" href="css/normalize.css">
  <link rel="stylesheet" href="css/style.css">
</head>
<body>

<!-- plantilla de la aplicación - ->

  <script type="text/x-handlebars">
    {{outlet}}
  </script>

<!-- plantilla index - ->

  <script type="text/x-handlebars" data-template-name="index">
  </script>

  <script src="js/libs/jquery-1.9.1.js"></script>
  <script src="js/libs/handlebars-1.0.0.js"></script>
  <script src="js/libs/ember-1.0.0.js"></script>
  <script src="js/libs/ember-data.js"></script>
  <script src="js/libs/showdown.js"></script>
  <script src="js/libs/moment.js"></script>
  <script src="js/app.js"></script>

</body>
</html>
```

aplicación, dentro de la plantilla simple *handlebars*, la barra está compuesta por cuatro secciones, "*Bloggr*", "*Mensajes*", "*Galería*" y "*Acerca de*".

[13] Texto de marcado ligero

```
<!-- plantilla principal - ->
  <script type="text/x-handlebars">
    <div class="navbar-inner">
      <a class="brand" href="#">Blog Ember.js</a>
        <ul class="nav">
          <li>{{#link-to 'posts'}}Mensajes{{/link-to}}</li>
          <li>{{#link-to 'gallery'}}Galería{{/link-to}}</li>
          <li>{{#link-to 'about'}}Acerca de{{/link-to}}</li>
        </ul>
      </div>
    </div>

    {{outlet}}
  </script>
```

A través de *class* (clases) se aplican hojas de estilo (*bootstrap*), para dar apariencia de barra de navegación a las secciones del blog, se hace uso del helper *{{link-to}}* para enlazar las rutas *posts, gallery* y *about*, a los elementos visuales de la barra de navegación que correspondan. La expresión *{{outlet}}* indica que el resto de plantillas a utilizar en el blog se visualizaran debajo de la barra de navegación.

A continuación se crean las rutas principales con las que debe trabajar la aplicación, que han sido enlazadas a los elementos que apunta el heler *{{link-To}}* de la figura anterior (*posts, gallery y about*). Para ello se ubica el archivo *app.js* y se definen en la función *App.Router.map.* **¡Error! No se encuentra el origen de la referencia.**

```
<!-- Rutas principales - ->
App.Router.map(function() {
  this.resource('about');
  this.resource('gallery');
  this.resource('posts')
  });
});
```

Seguidamente, se crean las plantillas *about, gallery* y *posts*, en el archivo *index* que son a las que su ruta respectiva ubicará al momento que el usuario realice una interacción en la barra de navegación. Por ahora vacías para verificar el comportamiento del blog.

```
            <!—-plantillas por nombres para el blog- ->

<!—- plantilla posts - ->
 <script type="text/x-handlebars" data-template-name="posts">

  </script>

<!—- plantilla gallery - ->
  <script type="text/x-handlebars" data-template-name="gallery">

  </script>

<!—- plantilla about - ->
  <script type="text/x-handlebars" data-template-name="about">

  </script>
```

Si todo va bien, se debe ver la barra del blog con las secciones “Mensajes” “Galería” y “Acerca de” al ejecutar el archivo index. Figura 38, y al hacer clic por una de ellas la barra de direcciones del navegador debe reflejar los cambios en la ruta.

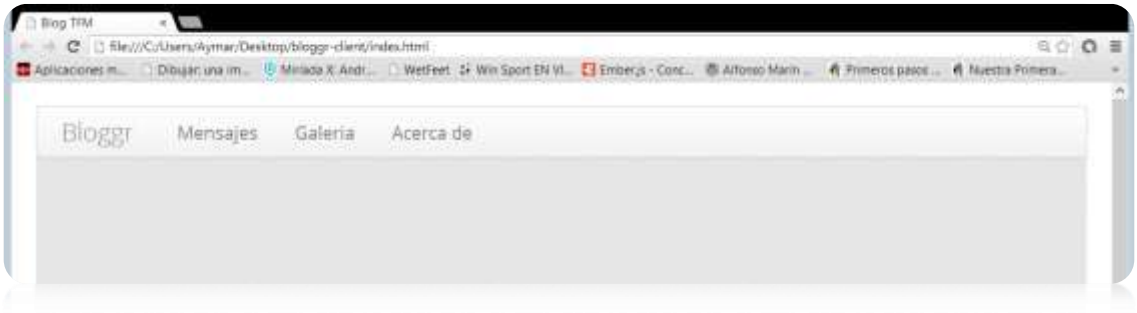

Figura 38. Barra de navegación de la aplicación.

Hasta ahora se tiene divido el blog en dos secciones, una superior con la barra de navegación y otra inferior donde se despliegan las plantillas creadas por nombres. La idea es que cuando el usuario acceda a la opción de “Mensajes” esta sección inferior se divida en dos partes: a mano izquierda una opción para crear un “Nuevo Mensaje”, seguido de una lista de mensajes anteriormente creados; al seleccionar alguno de ellos su contenido pueda ser redenrizado en el espacio restante a mano derecha, mostrando el contenido del mensaje seleccionado más las opciones de editar y eliminar Figura 39

Figura 39. Esquema de posición de plantillas para el posts.

Por su parte la opción la opción de "Galería", desplegará un conjunto de imágenes y "Acerca de" información estática sobre el blog.

Para continuar creando el blog, se debe definir un modelo y con él un Store de datos personalizado dentro del archivo *app.js*. Para ello se extiende el adaptador FixtureAdapter. **¡Error! No se encuentra el origen de la referencia.** en este caso, se indica que para cada tipo de modelo que se desea representar, se debe crear una nueva subclase de DS.FixtureAdapter.

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

<!—- Extensión del FixtureAdapter – ->
App.ApplicationAdapter = DS.FixtureAdapter.extend();
```

El uso del adaptador FIXTURES se debe a que en esta aplicación no cuenta con una API del lado del servidor, por lo que se simulará la carga de datos a través de este adaptador usando *DS*. Para conectar estos datos de prueba a los datos del modelo, así cuando la *App* esté lista para su puesta en producción, simplemente se cambia el adaptador sin daño alguno en el resto del código.

Seguidamente se define el modelo con los atributos para manejar los datos de prueba.

```
<!—- Definición del modelo con sus atributos – ->

App.Almacen = DS.Model.extend({
   Title:DS.attr('string'),
   author:DS.attr('string'),
   date:DS.attr('string'),
   body:DS.attr('string')
});
```

Ahora se deben llenar los datos del FIXTURES, la forma de hacerlo se muestran a continuación:

```
<!—- Cargando los datos modelo en FIXTURES – ->

App.Almacen.FIXTURES = [{
   Id:'0',
   Title:'Android KitKat la nueva versión de Google',
   author:'TicBeat',
   date: new Date('12-24-2012'),
   body:' La versión 4.4 de Android, la próxima del …'
},{
   Id:'1',
   Title:' ¿En qué estado está Ubuntu Touch para….',
   author:' Javier Pastor',
   date: new Date('12-24-2012'),
   body:' Ubuntu, una de las alternativas más …'
}];
```

Ya se tiene el modelo y la ruta *posts*, ahora se debe crear la plantilla en el archivo *index* de la aplicación, para que cuando el usuario seleccione la opción de "Mensajes" en la barra de navegación del blog, se despliegue esta plantilla debajo de dicha barra. El proceso interno que sigue Ember.js seria ubicar la ruta "posts" y enlazarla a esta plantilla para ofrecer esta vista al usuario. Esta plantilla debe ser configurada según el modo que se desee sea visualizada Figura 39.

Se crea entonces una tabla dividida en dos partes: una sección *span3*, que debe contener un enlace a la ruta "*newpost*", y un *{{#each}}* con un enlace a la ruta *"post"* para iterar y representar los atributos *{{titulo}}* y *{{autor}}* de esa ruta.

El uso de {{#each}} se debe a que en esta división se va a trabajar con el elemento serializado de la ruta post que se definirá más adelante.

La siguiente sección de la tabla (*span9*) solo contendrá una marca {{outlet}} que indica la posición donde se mostraran las plantillas de los mensajes desplegados en el blog.

```
                    <!—-plantilla por nombres para el posts- ->
<script type="text/x-handlebars" data-template-name="posts">
   <div class="container-fluid">
      <div class="row-fluid">
        <div class="span3">
          <table class='table’>
            <thead>
                <tr><th>{{#Link-to 'newpost'}}Nuevo Mensaje {{/Link-to}} </th></tr>
                <tr><th>Mensajes Recientes</th></tr>
             </thead>
            {{#each model}}
            <tr><td>
                {{#link-to 'post' this}}{{title}} 
                   <smallclass='muted'>por {{author}}</small>
                {{/link-to}}
            </td></tr>
            {{/each}}
          </table>
        </div>
        <div class="span9">
          {{outlet}}
       </div>
      </div>
    </div>
  </script>
```

Esto se debería ver de la siguiente manera en el navegador del siguiente modo:

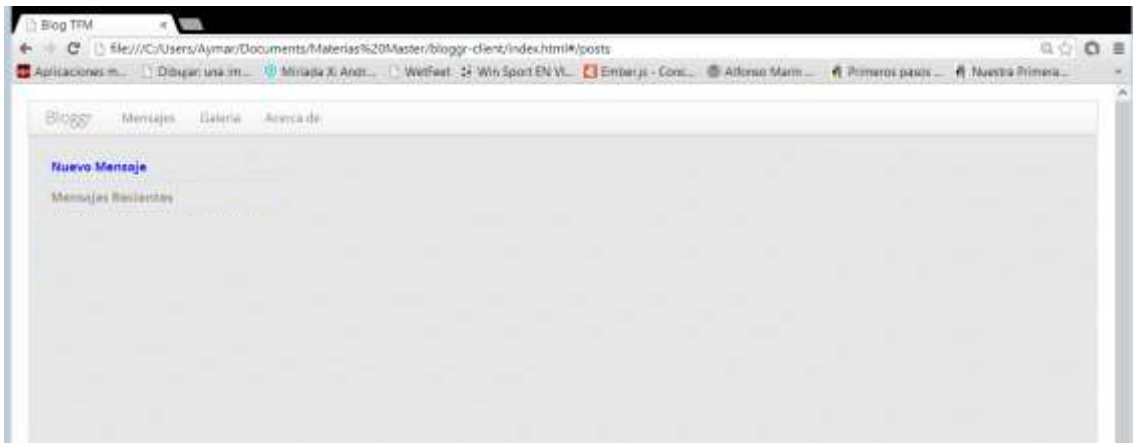

Figura 40. Tabla para la plantilla posts

Por el momento no se observa ningún mensaje reciente, debido a que no se ha indicado qué modelo o colección de modelos de datos se quiere mostrar cuando se visite la ruta “Posts.Route” (este modelo de datos es el que se definió anteriormente en FIXTURE el cual se va a identificar en lo sucesivo como “almacen”).

Para indicar el modelo con el que se va a trabajar se extiende la rutas “Post.Route” en el *archivo app.js*.

```
<!-- Aplicando el modelo de datos almacén a la rura posts - ->

App.PostsRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('almacen');
  }
});
```

De este modo la ruta podrá representar el modelo almacen junto a sus aributos. Se podra ver en el navegador de la siguiente manera: Figura 41. Mensajes recientes

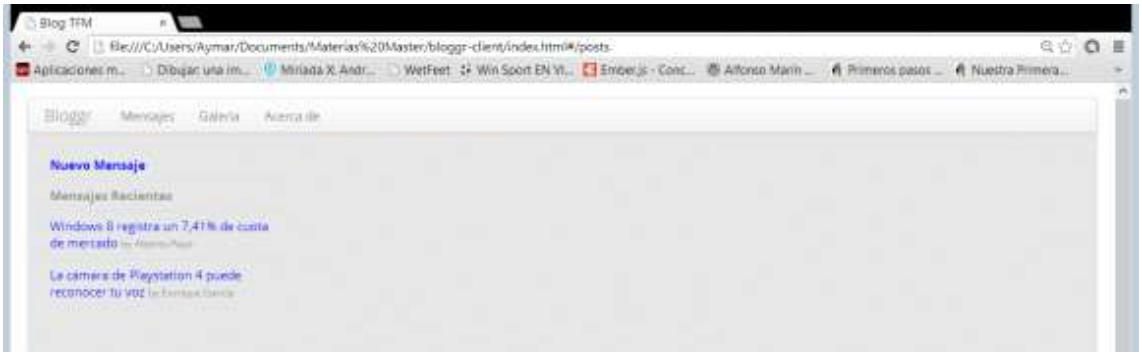

Figura 41. Mensajes recientes

Observe que se muestra solo los atributos del almacen a los que se ha apuntado anteriormente en la división span3 de la tabla para la ruta "posts".

Veamos ahora como crear un nuevo mensaje: para eso se debe anexar en la funcion *App.Router.map* ubicada en el archivo *app.js* (donde anteriormente se crearon las rutas principales) una subruta de "post" para *"newpost"*.

```
<!-- Ruta para newpost - ->

App.Router.map(function() {
  this.resource('about');
  this.resource('gallery');
  this.resource('posts');
  this.resource('newpost');
});
```

Una vez creada la ruta se puede crear la plantilla para esta ruta dentro del archivo index de la aplicación.

```
<!-- plantilla newpost - ->

  <script type="text/x-handlebars" data-template-name ="newpost">
    <h1>Nuevo Mensaje</h1>
    <p>Titulo {{input type="text" value=newTitle}}</p>
    <p>Autor {{input type="text" value=newAuthor}}</p>
    <p>Mensaje {{textarea value=newBody}}</p>
    <button {{action 'createPost'}}>Aceptar</button>
  </script>
```

Dentro de esta plantilla se crean dos campos de textos uno para que el usuario introduzca el nuevo título y otro para el autor, también se crea un área de texto para ingresar el contenido del nuevo mensaje y por último un objeto "botón" (Aceptar).

El botón "*Aceptar*" que se genera en esta plantilla hasta ahora no efectúa ninguna acción, debido a que se le debe asignar un controlador para soportar el evento generado por este *{{action 'createPost'}}*.

Este controlador se crea en el archivo *app.js* con el mismo nombre de la plantilla (*newpost*).

```
<!-- Controlador para newpost - ->

App.NewpostController = Ember.ArrayController.extend({

    createPost: function () {

      // Get the todo title set by the "New Todo" text field
      var title = this.get('newTitle');
      var author = this.get('newAuthor');
      var body = this.get('newBody');

      //fecha
      var f = new Date();

      // Create the new  model
      var todo = this.store.createRecord('almacen', {
        title: title,
        author: author,
        date: new Date((f.getMonth() +1) + "-" +f.getDate() + "-"
             + f.getFullYear()+":"+f.getHours()+":"+f.getMinutes()+":"+f.getSeconds()),
        body: body
      });

       // Clear the "New Todo" text field
      this.set('newTitle', '');
      this.set('newAuthor', '');
```

En este controlador se describe la función "*createPosts*" (es decir la acción a tomar cuando el usuario seleccione el botón Aceptar). En primer lugar se procede a capturar la información introducida por el usuario para los atributos "*title*", "*author*" y "*body*" que hacen parte del modelo y los almacene en "*newTitle*", "*newAuthor*" y "*newBody*" respectivamente. Lo siguiente que se hace, es crear una nueva variable para almacenar el valor de la fecha actual y por último crear un nuevo registro al almacén del modelo posts al que se le pasan los atributos. Las últimas líneas se crean con el fin de limpiar el formulario de nuevo mensaje. *Figura 42*

Figura 42. Vista del formulario Nuevo Mensaje

Ya se pueden crear nuevos mensajes, pero no se podrán mostrar hasta que no se redefina la ruta "*post*" en el archivo *app.js*.

```
<!-- Ruta para post - ->

App.Router.map(function() {
  this.resource('about');
  this.resource('gallery');
  this.resource('posts',function(){
    this.resource('post', {path: ':post_id'});
    this.resource('newpost');
});
});
```

La ruta "post" debe anidar la ruta de cada mensaje añadido (el elemento path: es utilizado para ello), por lo que se le asigna el método _id para poder serializar cada mensaje sin cambiar la ruta raíz "post", un ejemplo quedaría del siguiente modo: #/posts/1.

Se procede ahora a crear la plantilla para el *"post"* dentro del archivo *index.*.

```
<!-- Plantilla post - ->

  <script type="text/x-handlebars" data-template-name ="post">
    <h1>{{title}}</h1>
    <h2>by {{author}} <small class='muted'>({{format-date date}})</small></h2>
    <hr>

    <div align="justify" class='below-the-fold'>
      {{format-markdown body}}
    </div>
</script>
```

En esta plantilla se crea la forma de visualizar cada uno de los atributos de un mensaje. Para los atributos *{{titulo}}* y *{{autor}}* se hace uso de de la función *format-date* y para el *{{body}}* el formato establecido será el determinado en la función *format-markdown*. Estas funciones deben ser definidas en el archivo *app.js.*

```
    <!-- Definicion de funciones - ->

    var showdown = new Showdown.converter();

    Ember.Handlebars.helper('format-markdown', function(input) {
      return new Handlebars.SafeString(showdown.makeHtml(input));
    });

    Ember.Handlebars.helper('format-date', function(date) {
      return moment(date).fromNow();
    });
```

conjuntamene se crea la ruta que relacióne cada *post* con el segmento dinámico _id antes descrito.

```
        <!-- Relacionando cada post con su id - ->

        App.PostRoute = Ember.Route.extend({
          model: function(params) {
          return App.Almacen.FIXTURES.findBy('id', params.post_id);
          }
         });
```

Ahora se debe dotar de interactividad a la aplicación permitiendo al usuario la edición y la eliminación de los *post*.

En el archivo index se ubica la plantilla "post" creada anteriormente, para anexar el condicional *{{isEditing}}* que va a permitir controlar las acciones generadas por el botón editar.

```
<!—- Plantilla post – ->

  <script type="text/x-handlebars" data-template-name ="post">
    <h1>{{title}}</h1>
    <h2>by {{author}} <small class='muted'>({{format-date date}})</small></h2>
    <hr>

    <div align="justify" class='below-the-fold'>
      {{format-markdown body}}
    </div>

   {{#if isEditing}}
      {{partial 'post/edit'}}
      <button {{action 'doneEditing'}}>Aceptar</button>
    {{else}}
      <button {{action 'edit'}}>Editar</button>
      <button {{action 'delete' this}}>Eliminar</button>
    {{/if}}

</script>
```

Detalladamente se indica que si se está editando, se muestre como vista parcial la plantilla "*post/edit*" junto con el botón *Aceptar*, que permite guardar los cambios. De lo contrario mostrar junto con la información referente al post los botones "*editar*" y "*eliminar*" para que el usuario ejecute alguna de estas acciones.

Ahora se debe configurar en el archivo *app.js* un controlador para soportar el evento generado por los botones "*editar*" y "*eliminar*". Como se está trabajando sobre la plantilla "*post*", el controlador a generar debe ser *App.postController*.

```
<!-- controlador para los botones de la plantilla post - ->

App.PostController = Ember.ObjectController.extend({

  isEditing: false,
  edit: function() {
    this.set('isEditing', true);
  },
  doneEditing: function() {
    this.set('isEditing', false);
    this.get('model').commit();
  },
   delete: function () {
    this.set('isEditing', false);
    var almacen = this.get('model');
    almacen.deleteRecord();
    almacen.save();
  }
 });
```

Una vez configurado el controlador, se debe definir la plantilla *"post/edit"* en el archivo *index*, la cual será la que se va a desplegar cuando se invoque la vista parcial en el *post* al momento de presionar el botón *editar*. Desplegando dos cuadro de *text* (título y autor) y un área de texto para que se editar el mensaje.

```
    <!-- Plantillla post/_edit - ->

      <script type="text/x-handlebars" data-template-name="post/_edit">
        <p>{{input type="text" value=title}}</p>
        <p>{{input type="text" value=author}}</p>
        <p>{{textarea value=body}}</p>
      </script>
```

Se llena el contenido de la plantilla *gallery* con enlace a imágenes para mostrar y la plantilla *about* con contenido estático de información acerca del blog

```
<!-- Plantillla gallery - ->

    <script type="text/x-handlebars" data-template-name="gallery">
    <h2>Galeria de Imagenes</h2>
    <div id="container">
      <br>
      <div align="center" class="span6">
        {{#link-to 'gallery' rel="images/1.png" class="image"}}<img src="images/T1.png"
            class="thumb" border="0"/>{{/link-to}}
      </div>
      <div class="span6" id="image"><img src="images/1.png" border="0"/></div>
    </div>
  </script>
```

```
<!-- Plantillla about - ->

<script type="text/x-handlebars" data-template-name="about">
    <div class='about'>
      <h2>Acerca de</h2>
      <br>
      <div class="span3" align="center">
        <img src="images/photo.png"  border="0"/>
      </div>
      <div class="span9">
          <br><br><br>
          <p>Blog desarrollado como ejemplo del uso de EmberJS, ….</a>
          </p>
      </div>
    </div>
  </script>
```

Por último, se rellena la plantilla *index* con un texto para indicarle al usuario que hacer en la ruta *index* (creada automáticamente por Ember.js).

```
<!-- plantilla index - ->

  <script type="text/x-handlebars" data-template-name="index">
     <p>class="text-warnig"> Por favor seleccione un mensaje</p>
  </script>
```

A esta plantilla se le crea una ruta en el archivo *app.js* para que redireccione.

```
    <!-- Ruta para la plantilla index - ->
    App.IndexRoute = Ember.Route.extend({
      redirect: function(){
        this.transitionTo('posts');  }
    });
```

Como nota final, se indica que el código de esta aplicación está disponible para su descarga en https://github.com/AGMar/bloggr-client. Y blog puede ser visualizado a través de la siguiente dirección: www.blogemberjs.p.ht.

6 Conclusiones

- El desarrollo de aplicaciones web estuvo por muchos años limitado solo en servidores, al plantearse el paradigma de usar "el lado del cliente", la solución lógica fue el uso de JavaScript, naciendo con ello un nuevo enfoque para el uso de otras tecnologías involucradas en un desarrollo web. La idea era "no reinventar la rueda" pero si corregir su rumbo.

- El patrón de diseño MVC es el que en esencia da respuesta a la necesidad de que una aplicación web sea escalable, a través de la separación de funciones.

- La implementación del modelo MVC en frameworks para el desarrollo de aplicaciones web cliente, ofrece el mantenimiento de un código estructurado.

- Se elimina la figura del Web máster en el desarrollo web que todo lo hacía, actualmente se dividen las cargas en un desarrollo front-end y back-end.

- Existen diversidad de frameworks creados con JavaScript, que en un principio se pueden dividir en creados por terceros y oficiales. Inmersos en esta división se puede identificar a framework para el desarrollo front-end, back-end y frameworks intermedios.

- La identificación oportuna de la presencia de características como vistas parciales, filtrado de listas, elementos personalizados y un desarrollo totalmente del lado del cliente en un framework, agilizan la toma de decisión sobre cual framework utilizar.

- Ember.js se define finalmente como un framework MVC opensource, que permite el desarrollo de aplicaciones de una sola página con una curva de aprendizaje empinada.

- El uso de Ember.js implica que como desarrollador se implementen solo los elementos que se necesiten, del resto se encarga el mismo. Por ejemplo, si se crea un objeto "*NotasController*" este framework admite que está asociado a una vista y datos con el mismo nombre "*Notas*".

- El uso de Ember.js no solo ofrece beneficios para los desarrolladores, sino también para un usuario final, quienes actualmente se pueden favoreces de la característica "carga rápida" que ofrece.

- Ember.js, posee una comunidad de desarrollo que está en constante actividad para mantener al día las librerías JavaScript dentro de la tienda de

datos que este framework incorpora. Actualmente se recomienda hacer uso de la tienda ember-data de forma cautelosa mientras no se estabilice su versión 1.0.

Bibliografía

[1] B. Aumaille, «Componentes de una aplicación Web,» de *J2EE desarrollo de aplicaciones web*, Barcelona, Ediciones ENI. ISBN:2-7460-1912-4, Noviembre, 2002, pp. 26-28.

[2] P. Bermejo, «Conferencia: Arquitectura Web en Entornos Profesionales,» de *Arquitectura Web en Entornos Profesionales*, Asturias., 2012.

[3] E. Evans, «modelado la capa de presentación.,» de *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Massachusetts, ISBN 0-321-12-521-5, 2004, pp. 45-60.

[4] E. R. Zulian, «Implementación de un framework para el desarrollo de aplicaciones web utilizando patrones de diseño y arquitectura MVC/REST,» Universidad de Belgrano. Facultad de Tecnología Informática Ingeniería en Informática., Buenos Aires- Argentina, Febrero 2010.

[5] Á. Martínez Echevarría, «MANUAL PRÁCTICO DE HTML,» 1995. [En línea]. Available: http://www-app.etsit.upm.es/~alvaro/manual/manual.html#1. [Último acceso: 2 julio 2013].

[6] Google ©2010-2012, «Web oficial de Angular.js,» [En línea]. Available: http://angularjs.org/. [Último acceso: 20 Junio 2013].

[7] A. Osmani, Developing Backbone.js Applications, O'Reilly Vlg. Gmbh & Co. ISBN-13: 978-1449328252, Junio 2013.

[8] J. Haagen Skeie, «¿que es Ember.js?,» de *Ember.js in {{Action}}*, Manning Publications, 2013, p. 7.

[9] Copyright 2013 Tilde Inc. Design by HeroPixelCore Team. Project Sponsors, «Ember Conceptos Básicos,» [En línea]. Available: http://emberjs.com/guides/concepts/core-concepts/. [Último acceso: 26 julio 2013].

[10] Mac Caw, Alex; O'Reilly's JavaScript Web Applications;, «Web oficial del Framework Spine.js,» [En línea]. Available: http://spinejs.com/. [Último acceso: 02 julio 2013].

[11] «Web oficial del Framework Batman.js,» [En línea]. Available: http://batmanjs.org/. [Último acceso: 14 junio 2013].

[12] S. Vicencio, «Análisis de factores criticos de desempeño en aplicaciones web máviles basadas en framework MV*,» Mayo-2013.

[13] J. Haagen Skeie, Ember.js in Action, Manning Publications. V6, 2013.