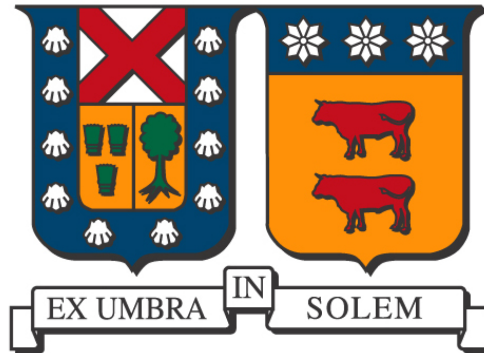
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



"PROPUESTA DE MEJORA TECNOLÓGICA UTILIZANDO UN JAVASCRIPT FRAMEWORK PARA LA EMPRESA SIDEKICK"

DANIEL ANDRÉS MÁRQUEZ LUTFY

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERÍA CIVIL EN INFORMÁTICA

Profesor Guía: Hubert Hoffmann
Profesor Correferente: Paulo Tarud

Agosto 2016

Le dedico esta memoria a mi familia y amigos que me apoyaron durante mis estudios y la realización de ésta.

Agradecimientos

Le doy las gracias a mis padres, Marta y Osvaldo; a mis hermanos, Tamara y Sebastián; a mis tíos y tías; a mis primos y primas; a mis amigos: Gabriel, Abdel, Nicolás, Pablo, Danilo, Eugenio, Cristian, Leonías, John, Víctor y Eduardo; a Don Hubert Hoffmann, profesor guía; a Don Paulo Tarud, correferente; a mis compañeros de trabajo y equipo de SIDEKICK, Javiera y Gustavo; y a todos los que me han acompañado durante mis años de estadía en la Universidad. Muchas gracias por todo.

Resumen

El presente documento, “PROPUESTA DE MEJORA TECNOLÓGICA UTILIZANDO UN JAVASCRIPT FRAMEWORK PARA LA EMPRESA SIDEKICK”, trata sobre: el proceso de selección del *Javascript Framework*; incorporación e integración de éste a una aplicación ya existente a través de una aplicación prototipo; comparaciones del desarrollo realizado con aplicaciones existentes dentro de la empresa SIDEKICK; y conclusiones obtenidas del aprendizaje logrado.

Abstract

This document, “PROPOSAL TO TECHNICAL IMPROVEMENTS USING A JAVASCRIPT FRAMEWORK FOR THE SIDEKICK COMPANY”, it’s about: the selection process of a Javascript Framework; its integration in an existent application using a application prototype; comparisons between the development and existent applications in the company SIDEKICK; and the conclusions and insights learnt during the process.

Índice general

Resumen **1**

Índice de figuras **5**

Índice de cuadros **6**

Glosario **7**

1. Introducción **8**

1.1. Definición del Problema 8

1.2. Objetivos 9

1.2.1. Objetivo General 9

1.2.2. Objetivos Específicos 9

1.3. Estructura del Documento 9

2. Estado del Arte **10**

2.1. Historia de HTML 10

2.1.1. HTML 10

2.1.2. HTML2 11

2.1.3. HTML3 12

2.1.4. HTML4 12

2.1.5. HTML5 13

2.2. Historia de JavaScript, ECMAScript y Ajax 13

2.3. Architectural JS Frameworks 15

3. Selección del Javascript Framework 17

3.1. Análisis de una aplicación transversal 17

3.1.1. Análisis cuantitativo 18

3.1.2. Análisis cualitativo 19

3.2. Revisión de aspectos generales y algunos específicos de cada Javascript Framework 20

3.2.1. Criterios o Aspectos Generales, escala y medición 21

3.2.2. Criterios o Aspectos Específicos 22

3.2.3. Compatibilidad Técnica con aplicaciones existentes en la empresa SIDEKICK y selección final del Javascript Framework 23

4. Incorporación e Integración del Javascript Framework 25

4.1. Incorporación de Emberjs 25

4.1.1. Incorporación de Emberjs utilizando Bundler 26

4.1.2. Incorporación directa de Emberjs como dependencia 26

4.1.3. Estudio de la documentación de Emberjs para la integración con la aplicación escogida 27

4.2. Condiciones adicionales para la incorporación e integración de un Javascript Framework 29

4.3. Incorporación de Marionette 30

4.3.1. Incorporación directa de Marionette como dependencia 30

4.3.2. Estudio de la documentación de Marionette para la integración con la aplicación escogida 32

5. C. de la A. P. utilizando el JS Framework escogido 33

5.1. Definición de la funcionalidad 34
5.1.1. Diagrama de secuencia del prototipo 36
5.2. Definición de la estructura y flujo de trabajo 36
5.2.1. Estructura de la aplicación prototipo 36
5.2.2. Flujo de la aplicación prototipo 37
5.3. Construcción de la aplicación prototipo 38
5.4. Comparación de la aplicación prototipo con desarrollos existentes en la empresa SIDEKICK 39

6. Conclusiones 41

Referencias 43

Índice de figuras

3.1. Interfaz de la aplicación Todos - Fuente: Imagen elaboración propia 18

4.1. Flujo de trabajo en una aplicación Emberjs - Fuente: Elaboración propia 28

5.1. Caso de uso para la creación de un nuevo Tipo de Tarea - Fuente: Elaboración propia 34

5.2. Caso de uso modificado usando Marionette - Fuente: Elaboración propia 35

5.3. Diagrama de secuencia del prototipo - Fuente: Elaboración propia 36

5.4. Flujo general de la aplicación prototipo - Fuente: Elaboración propia 38

5.5. Formulario Modal generado por la aplicación prototipo - Fuente: Elaboración propia 39

Índice de cuadros

3.1. Cantidad de Líneas de Código Fuente (SLOC) por Aplicación o JS Framework . 19

3.2. Respuestas a las tres preguntas planteadas en la sección 3.1.2 20

3.3. Resumen de notas por JS Framework para los criterios de la sección 3.2 22

3.4. Resumen de cumplimiento de requerimientos por cada JS Framework de la Sección Emberjs 23

Glosario

API Application Programming Interface.

DOM Document Object Model.

ERP Enterprise Resource Planning.

ERPS Enterprise Resource Planning SIDEKICK.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

JS Javascript.

MVC Model View Controller.

REST Representational State Transfer.

SLOC Source Lines of Code.

URL Uniform Resource Locator.

W3C World Wide Web Consortium.

WHATWG Web Hypertext Application Technology Working Group.

Capítulo 1

Introducción

1.1. Definición del Problema

El desarrollo de sistemas o aplicaciones web es un desafío cada vez más complejo. Por un lado, los clientes quieren que éstas sean rápidas, eficientes, confiables, escalables, modulares, distribuidas, mantenibles, etc. Por otra parte, los programadores y arquitectos de software poseen a su alcance una amplia gama de herramientas y tecnologías para dar solución a cada uno de estos puntos. ¿Qué herramientas se adecuarán mejor en la construcción de la solución o a determinados aspectos del problema?, es una pregunta que surge con frecuencia en el equipo de desarrollo de la empresa SIDEKICK y, muchas veces, no es fácil de responder.

Bajo este contexto, en la empresa SIDEKICK, se ha determinado que se necesita mejorar el cómo y el con qué se construyen los componentes *frontend* o vistas de las aplicaciones desarrolladas, para lograr una mejora en la modularidad y reusabilidad de ellos.

Para verificar que se ha logrado solucionar la problemática en la empresa SIDEKICK, a continuación se definen los objetivos que deben ser desarrollados y terminados al final de la memoria.

1.2. Objetivos

1.2.1. Objetivo General

Diseño y desarrollo de una aplicación prototipo, utilizando un *Javascript Framework*, que permita comprobar la potencialidad de la arquitectura REST y su uso en la construcción de aplicaciones *frontend* para la empresa SIDEKICK.

1.2.2. Objetivos Específicos

1. Investigar *Javascript Frameworks* existentes, compararlos brevemente y escoger uno para aplicarlo.
2. Estudiar documentación del *Javascript Framework* escogido para construir una aplicación prototipo.
3. Comparar la aplicación prototipo construida con desarrollos de software existentes en la empresa SIDEKICK.

1.3. Estructura del Documento

La estructura general de este documento se divide en:

- Introducción, Capítulo 1.
- Estado del Arte, Capítulo 2.
- Selección del *Javascript Framework*, Capítulo 3.
- Incorporación e Integración del *Javascript Framework*, Capítulo 4.
- Construcción de la aplicación prototipo utilizando el Javascript Framework escogido, Capítulo 5.
- Concluciones, Capítulo 6.

Capítulo 2

Estado del Arte

Es importante conocer cómo han ido cambiando los lenguajes y las tecnologías que serán utilizados para el desarrollo de este documento, ya que algunos problemas y desafíos surgen a partir de las adaptaciones y diversificaciones que van adquiriendo con el tiempo.

2.1. Historia de HTML

Las secciones 2.1.1, 2.1.2 y 2.1.3 son una traducción y resumen del texto [DR98]. En ellas se exponen los inicios del lenguaje HTML y el surgimiento de los primeros problemas relacionados al desarrollo de éste. Por otra parte, la sección 2.1.4 expone las fechas de aparición de la versión 4 del lenguaje. Finalmente, la sección 2.1.5 muestra el estado actual y los actores que participan en el desarrollo del lenguaje HTML5.

2.1.1. HTML

HyperText Mark-up Language, más conocido como HTML, es un lenguaje creado e inicialmente desarrollado por Tim Berners-Lee [DR98] en el año 1989 en el Laboratorio Europeo de Física de Partículas (CERN) ubicado en Geneva, Suiza. Éste fue concebido con el objetivo de enlazar y compartir documentos científicos a partir de *Global Hypertext Links*, más conocidos como *Hyperlinks*, y el protocolo de comunicación HyperText Transfer Protocol o HTTP.

La estructura base de HTML se origina de otro lenguaje llamado *Standard Generalized Markup Language* o SGML. Éste utiliza pares de etiquetas para definir bloques de contenido. Un

ejemplo de lo anterior es `<TITLE>Mi Título</TITLE>`, que expresa inequívocamente el título de un artículo o de cualquier tipo de documento definido con el lenguaje.

El primer prototipo de *browser* o navegador fue inventado por el mismo Tim Berners-Lee [DR98] en el año 1990 y permitía visualizar un documento escrito con el lenguaje HTML en una estación de trabajo NeXT.

Aunque la idea de etiquetas no era nueva, la introducción de los *Hyperlinks* de Tim Berners-Lee [DR98] y su disposición de compartir sus ideas abiertamente permitieron volver popular este nuevo lenguaje, lo que fomentó e impulsó su posterior desarrollo.

En el año 1991 se inicia de forma abierta la discusión sobre HTML y en el año 1992 la *National Center for Supercomputer Application*, o NCSA, inicia el desarrollo de su propio navegador llamado Mosaic [Gro16], el cual fue lanzado en el año 1993 para la estación de trabajo de Sun Microsystems Inc. [tho16] y extiende las funcionalidades de HTML, lo cual da origen a uno de los primeros problemas importantes de la Web: La interoperabilidad.

Discusiones aparentemente simples de cómo definir un par de etiquetas (`<IMG>` para compartir imágenes, por ejemplo) no estuvieron libres de controversia y aún lo están. Algunas decisiones de cómo se debían definir e implementar estas etiquetas se tomaron arbitrariamente o en base a desarrollos ya hechos.

Durante el año 1993 y a principios del año 1994 cada navegador definía trozos de HTML distintos. En un esfuerzo por eliminar el caos y las malas definiciones del lenguaje, Dan Connolly [DR98] y sus colegas recolectaron todas las etiquetas HTML posibles que fuesen altamente utilizadas. Este esfuerzo daría origen a lo que Tim Berners-Lee [DR98] luego llamaría HTML2.

2.1.2. HTML2

En el año 1994 se publica *HyperText Mark-up Language* 2, más conocido como HTML2, el cual es un esfuerzo de Dan Connolly de estandarizar el avance del lenguaje considerando las opiniones de todos los actores posibles [DR98]. Ese mismo año se forma Netscape [Gro16] y se crea el navegador del mismo nombre, el cual se volvió inmensamente popular y exitoso. Como sus predecesores, Netscape empezó a desarrollar sus propias etiquetas HTML y, curiosamente, comenzó a liderar el desarrollo del estándar, algo que el núcleo de la comunidad de desarrollo de HTML creyó necesario remediar.

A finales del año 1994 se consolida *The World Wide Web Consortium* [WWWC16], o W3C,

con el objetivo de potenciar la Web a través del desarrollo de estándares abiertos. Poseían un fuerte interés en HTML y reclutaron a muchas de las personas más conocidas de la comunidad Web.

2.1.3. HTML3

El borrador del *HyperText Mark-up Language* 3, o HTML3, se publicó en el año 1995 e incluyó soporte a clases y estilos que permitían alterar propiedades visuales del contenido en una página web. Ejemplos de estas alteraciones son el cambio del tamaño de las letras o el cambio de color de éstas.

Como en la versión pasada, los navegadores existentes y en desarrollo crearon sus propias mejoras del lenguaje y que poco se apegaban al estándar. Este problema, no menor, generó confusión y dio paso a las extensiones que añadían funcionalidades adicionales a cada uno de los navegadores. Evidentemente, estas mejoras realizadas no formaban parte del lenguaje estándar y rara vez se tenía consenso en ellas.

En el mismo año de la publicación del borrador de HTML3 se introduce la versión 3.2 del lenguaje [DR98]. Ésta agregó soporte a uno de los elementos más importantes y utilizados de HTML: Las tablas. Éstas permitían dar formato y orden al contenido presentado en una página web y se utilizan hasta el día de hoy.

En enero del año 1997 el W3C termina y publica oficialmente HTML 3.2 como una especificación para la industria [DR98]. El estándar incluía tablas, *applets*, flujo de texto alrededor de las imágenes, *subscripts* y *superscripts*, características que se podían encontrar en versiones intermedias entre HTML2 y HTML3.2 (IETF HTML2, HTML+ y HTML3).

2.1.4. HTML4

El borrador del *HyperText Mark-up Language* 4, o HTML4, es anunciado el 8 de julio de 1997 por el W3C [WWWC97]. Este borrador consideró e incorporó mejoras importantes en la internacionalización de caracteres, accesibilidad, formularios, objetos, entre otras cosas.

Al igual que en las versiones anteriores, HTML4 estuvo sujeto a revisiones y discusión hasta el año 1998 donde fue oficialmente liberado [WWWC98].

La versión 4.01 del lenguaje es publicada el 24 de diciembre del año 1999 [WWWC99] y es, hasta la fecha, la que se utilizó por más años hasta la publicación oficial (como recomendación)

de HTML5 por el W3C [WWWC14].

2.1.5. HTML5

HyperText Mark-up Language 5, o HTML5, es la versión más reciente del lenguaje y empezó a ser desarrollado por el *Web Hypertext Application Technology Working Group* [WHATWG16c], más conocido por su acrónimo WHATWG. El grupo fue formado en el año 2004 por Apple [AI16], Mozilla [Moz16] y Opera [OSA16], en respuesta a la dimisión, por parte de la W3C, de la propuesta realizada por Mozilla y Opera de agregar mejoras a los formularios de HTML.

Dentro de los principios clave que propone el WHATWG se encuentran: la necesidad de que las tecnologías fuesen retrocompatibles; las especificaciones e implementaciones necesitan calzar entre sí, incluso si eso significaba cambiar una especificación en vez de la implementación; y que las especificaciones fuesen lo suficientemente detalladas para que las implementaciones pudiesen lograr completa interoperabilidad sin tener que realizar ingeniería inversa entre estas.

En el año 2006 el W3C toma interés en HTML5 y en el año 2007 se forma un grupo de trabajo con el WHATWG para desarrollar las especificaciones de éste [WHATWG16a]. Sin embargo, en el año 2011, se decide separar los esfuerzos de ambos grupos debido a las visiones distintas que poseían. Por una parte, el W3C quiere crear un estándar sólido y duradero con pocas modificaciones entre las versiones. Por otro lado, el WHATWG desea crear un estándar adaptativo que se ajuste a las necesidades y solucione problemas a medida que estos surjan.

El 28 de octubre del año 2014 el W3C publicó la recomendación de HTML5 [WWWC14] y el 8 de octubre del 2015 se publica el borrador en progreso de HTML5.1 [WWWC15]. Por otra parte, debido a la naturaleza que le atribuye el WHATWG a HTML5, la versión que mantiene este grupo del lenguaje se actualiza constantemente, ver [WHATWG16b].

2.2. Historia de JavaScript, ECMAScript y Ajax

En mayo del año 1995, Brendan Eich [WWWC12], que en ese entonces trabajaba en Netscape, creó en 10 días el lenguaje de programación *Javascript*. El nombre original del lenguaje fue Mocha, decidido por Marc Andreessen [WWWC12], fundador de Netscape. En septiembre de ese año se pasó a llamar LiveScript y finalmente en diciembre del mismo se pasaría a llamar Javascript, luego de recibir una licencia de marca por parte de Sun [tho16]. Este lenguaje tiene poco que ver con Java [Ora16] y su nombre fue solamente una decisión comercial debido a la

popularidad de este último en ese tiempo.

Entre los años 1996 y 1997 Javascript fue llevado a ECMA [WWWC12] para crear las especificaciones de un estándar que pudiese ser implementado por otros navegadores (aparte de Netscape). El trabajo anterior dio origen a ECMAScript [WWWC12], nombre del estándar oficial, donde Javascript pasó a ser una de las más conocidas implementaciones.

En los años 1998 y 1999 se publican, respectivamente, las versiones 2 y 3 de ECMAScript, las cuales fueron la base para las versiones actuales de Javascript [WWWC12].

Luego de un período de estancamiento en los avances, en el año 2005 se retoma el desarrollo ECMAScript con el apoyo de Macromedia [fun16] para crear la versión 4 del lenguaje. Sin embargo, entre los años 2007 y 2008, debido a las grandes diferencias de las implementaciones realizadas con ECMAScript 4, Doug Crockford [WWWC12] une fuerzas con Microsoft [Mic16] para crear ECMAScript 3.1, a modo de oposición de la versión 4.

Mientras lo anterior ocurría, Jesse James Garret [WWWC12] publicó un white paper en el cual acuñó el término Ajax o Asynchronous Javascript and XML [WWWC12], un conjunto de tecnologías, donde Javascript era el esqueleto, que eran utilizadas para crear aplicaciones web donde los datos se obtenían de forma asincrónica o mientras se realizan otras operaciones.

La introducción de Ajax originó la creación de bibliotecas de código abierto como Prototype [PCT16], jQuery [TjF16], MooTools [TMDT16], Dojo [DF16], entre muchas otras. Cabe destacar la importancia de estas en la construcción de aplicaciones y sitios web que no necesitan cargar nuevamente todo su contenido para mostrar cambios al comunicarse con un servidor de datos, por ejemplo. Lo anteriormente mencionado impulsa la necesidad de consenso en el desarrollo de ECMAScript y Javascript.

En julio del año 2008 los distintos actores en el desarrollo de ECMAScript 3.1 y 4 se juntaron a discutir el futuro del lenguaje [WWWC12]. El hecho anterior llevó a que en el año 2009 se decidiera convertir a ECMAScript 3.1 en ECMAScript 5 y continuar su desarrollo a través de una agenda llamada Harmony [EC14].

En junio del año 2011 se libera la versión 5.1 de ECMAScript [EI11] y en junio del año 2015 aparece ECMAScript 6 o también conocido por ES 6 Harmony [EI15]. Este último ya es soportado por SpiderMonkey, el motor del navegador web Firefox [Moz15], y otras aplicaciones que utilizan el lenguaje [Kan16].

Por otra parte, Javascript, de ahora en adelante JS, ya no es tan sólo utilizado en aplicaciones web, también se utiliza, por ejemplo, en plataformas de servidor como Node.js [NF16] que

permiten la construcción de aplicaciones asincrónicas.

También, se pueden encontrar APIs para HTML5 escritas en JS que permiten el control de contenido multimedia (videos, fotos y audio), abrir puertos web para comunicación con servidores de datos, obtener datos georreferenciados, alterar la apariencia del contenido en una página web y mucho más. Algunas de estas características serán parte del tema principal de este documento.

2.3. Architectural JS Frameworks

El Modelo Vista Controlador, o MVC, es una arquitectura bastante utilizada en la construcción de aplicaciones web [AO12]. El Modelo y el Controlador se encuentran, normalmente, en el servidor. Por otro lado, la Vista es lo que el usuario observa en su navegador, en otras palabras, es la parte de la aplicación que se envía al navegador y la que incluye los *scripts*, las etiquetas HTML, los estilos y el contenido.

Las mejoras introducidas por Ajax ampliaron enormemente la cantidad de funcionalidades que se pueden crear con JS y que son incluidas en los *scripts* de las vistas. Sin embargo, más funcionalidades implican mayor complejidad en el desarrollo de éstas, más código que escribir, mantener y documentar, entre otras cosas.

Una posibilidad, para intentar mantener el orden y minimizar el trabajo, es crear métodos o funciones que abstraigan las tareas más comunes y que den solución a un problema determinado. Sobre la idea anterior surge la siguiente pregunta: ¿No se ha hecho esto antes?. La respuesta, en casi todos los casos, es sí.

El nacimiento de los *frameworks*, normalmente, se produce cuando existen tareas comunes que se pueden abstraer, implementar y estandarizar en un conjunto de funcionalidades concretas, las cuales se pueden aplicar a una gran variedad de situaciones y problemas de desarrollo de *software*.

En el caso de las páginas web una situación común puede ser algo como lo siguiente:

Una situación

Un usuario necesita, por ejemplo, borrar un mensaje de su lista de mensajes mientras visualiza ésta en una página web, el usuario presiona un botón y el mensaje se elimina. Antiguamente, y en muchos sitios web de hoy, este proceso se realizaba, o realiza, comunicando la página con el servidor, para luego refrescar ésta y mostrar que se borró el mensaje. La pregunta que surge es: ¿Es necesario refrescar la página completa para mostrar que se borró el mensaje?. Nuestra intuición, o al menos nuestra experiencia con software de escritorio, nos dice que no.

Existen muchas situaciones similares a la anterior, implementar cada una de ellas en JS (para que no se refresque la página completa) es muy costoso, tanto en tiempo como en esfuerzo. A eso hay que sumarle la generación de código HTML, ya que eso es lo que se le muestra indirectamente al usuario, y la manipulación del *Document Object Model* [WWWC05], más conocido como DOM, el cual da acceso a los *scripts* a realizar modificaciones en la página web. Es por esto que han surgido APIs, como jQuery [TjF16], que ayudan en esta tarea.

Estos APIs de apoyo no son suficientes para arquitectar bien las vistas, a pesar que ayudan en la creación de éstas, ya que crear páginas web se ha empezado a parecer, cada vez más, a la construcción de aplicaciones de escritorio.

Sin embargo, ambas situaciones se dan en contextos distintos. Lo que se ve en el navegador, la parte interactiva con el usuario o la vista, se encuentra separado y, muchas veces, muy lejano al servidor de datos, algo menos común en aplicaciones de escritorio.

Es por esto que surge la siguiente idea: Incluir una arquitectura adicional en la creación de las vistas, de tal forma que se comporten como una aplicación independiente y que sólo se comuniquen con el servidor cuando sea necesario.

Esta arquitectura adicional puede ir acompañada de un *framework* que la implemente, lo que da origen a los *Architectural JS Frameworks* [AO12]. Ejemplos de estos son: Angular [Goo16a]; Backbone [Ash16a]; React [Fac16]; entre otros. En [ANAS14] se puede ver un esquema donde se sitúan estos *frameworks* en la construcción de una aplicación.

Ya que los desafíos que estos *frameworks* deben enfrentar no son menores: eficiencia; rapidez; escalabilidad; mantenibilidad; interoperabilidad; retrocompatibilidad; entre otros, es importante compararlos y saber escoger el que más se ajuste a las necesidades de la aplicación que se quiera construir.

Capítulo 3

Selección del Javascript Framework

3.1. Análisis de una aplicación transversal

El proyecto TodoMVC [Osm16] ofrece una aplicación transversal, implementada en distintos *Javascript Frameworks*, que permite evaluar rápidamente algunas de las características principales de los *Javascript Frameworks* en estudio.

La aplicación consiste de una lista de tareas pendientes, definidas por el usuario, que pueden ser marcadas como completadas una vez sean terminadas.

La figura 3.1 muestra la interfaz de la aplicación, la cual es exactamente igual para todos los *Javascript Frameworks*.

Por otra parte, el proyecto incluye 65 implementaciones distintas de la aplicación descrita, una por cada *Javascript Framework*, de las cuales sólo serán consideradas 15 como objeto de estudio. Éstas corresponden a los *Javascript Frameworks* que no necesitan de compilación para poder funcionar. También, se descartaron los casos en donde las aplicaciones aún necesitan corrección o refinamiento, tal como aparece en [Osm16].

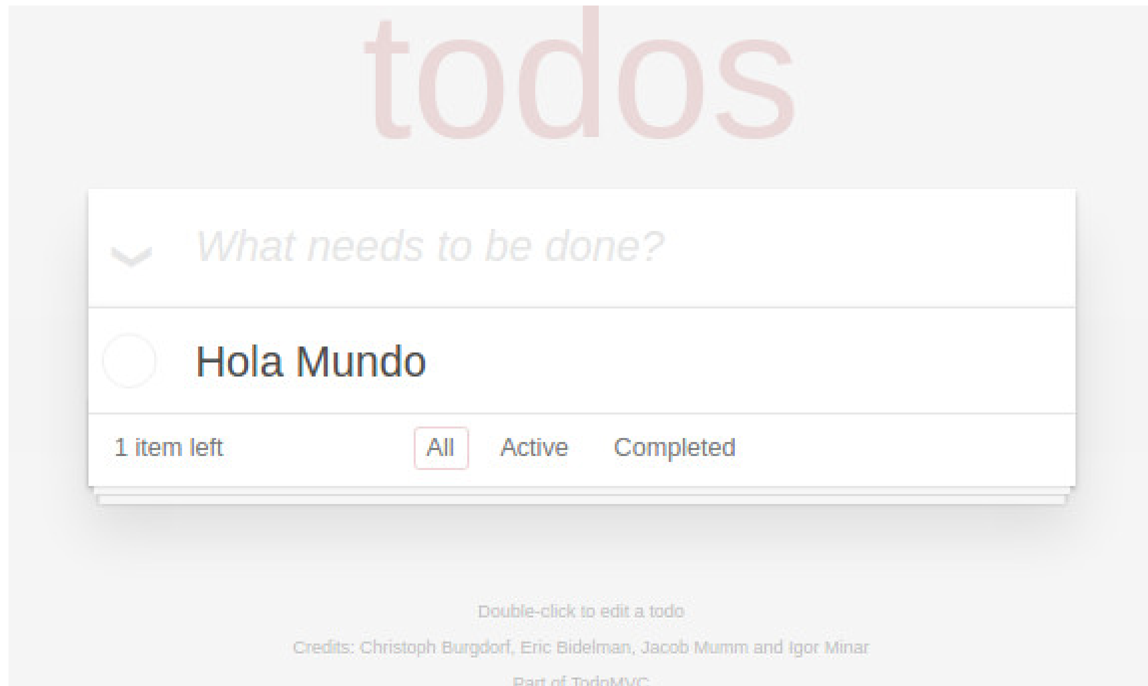


Figura 3.1: Interfaz de la aplicación Todos - Fuente: Imagen elaboración propia

3.1.1. Análisis cuantitativo

Para el análisis cuantitativo se considerará la Cantidad de Líneas de Código Fuente, *Source Lines of Code* o SLOC, de cada aplicación. Para ello se utilizará la herramienta escomplex [Boo16], que permite un conteo rápido y sencillo de éstas. Cabe mencionar que esta herramienta también permite obtener métricas como el Índice de Mantenibilidad [PO92], pero que en estudios recientes, como [DIKS12], han sido cuestionadas en su validez empírica, por lo que no se utilizarán en el análisis.

Se ha ordenado las aplicaciones de mayor a menor SLOC en el Cuadro 3.1. Sólo se ha considerado los archivos con código *Javascript* y se ha ignorado archivos con código HTML, ya que estos últimos son relativamente similares en todo los casos.

No se ha podido contar las Líneas de Código Fuente de la aplicación construida con *Polymer* [Goo16b], ya que éstas se encuentran incrustadas en los archivos HTML, por lo que resultan más difíciles de contar.

Cuadro 3.1: Cantidad de Líneas de Código Fuente (SLOC) por Aplicación o JS Framework

A. por JS Framework	**SLOC**	**A. por JS Framework**	**SLOC**
Troopjs	1515	Angularjs	430
React	1464	Backbone	383
Marionette	1392	Mithril	312
Dojo	916	Emberjs	223
Flight	760	Canjs	192
Knockoutjs	588	Knockback	168
Ampersand	512	Polymer	-
Vue	498		

Por otra parte, la razón por la que se están comparando las aplicaciones por SLOC es por la posible alta mantenibilidad y escalabilidad de éstas. Es natural suponer que aplicaciones con bajo SLOC son más fáciles de revisar y entender. En [DIKS12] se realizó un estudio empírico donde expertos concluyen que el sistema con menor SLOC resultó ser el más mantenible entre 4 opciones distintas.

Al inspeccionar visualmente las aplicaciones es fácil darse cuenta que la aplicación creada con Troopsjs [MK16] es mucho más difícil de entender que la construida con Knockback [KM16]. La misma conclusión se puede obtener realizando otras comparaciones.

A pesar de lo que se ha mencionado con anterioridad, no se descartarán *Javascript Frameworks* por esta única comparación de Cantidad de Líneas de Código Fuente. La aplicación transversal analizada es un caso particular y no es buena idea extrapolar las conclusiones obtenidas para posibles aplicaciones futuras. Sin embargo, queda claro que una mala decisión en la selección del *Javascript Framework* puede dar como resultado aplicaciones mucho más grandes y difíciles de mantener que otras realizadas con una buena elección.

3.1.2. Análisis cualitativo

El análisis cualitativo se realizará respondiendo las siguientes preguntas para cada una de las aplicaciones:

1. ¿Es fácil de entender el código fuente Javascript?
2. ¿Posee una estructura bien definida el código fuente Javascript?

3. ¿Sigue estándares generales, o buenas prácticas, acordes a los de la empresa SIDEKICK?

El Cuadro 3.2 muestra las respuesta a las preguntas planteadas en esta sección (3.1.2). Éstas fueron determinadas a través de inspección visual del código fuente, sin entrar en detalles del funcionamiento del mismo.

Cuadro 3.2: Respuestas a las tres preguntas planteadas en la sección 3.1.2

A. por JS Framework	**R.1**	**R.2**	**R.3**	**A. por JS Framework**	**R.1**	**R.2**	**R.3**
Troopjs	No	Sí	Sí	Angularjs	Sí	Sí	Sí
React	No	Sí	Sí	Backbone	Sí	Sí	Sí
Marionette	No	Sí	Sí	Mithril	Sí	Sí	Sí
Dojo	No	No	No	Emberjs	No	Sí	Sí
Flight	No	No	No	Canjs	Sí	No	No
Knockoutjs	No	No	No	Knockback	Sí	Sí	Sí
Ampersand	Sí	Sí	Sí	Polymer	No	No	No
Vue	Sí	No	No				

Aunque algunas aplicaciones no se entienden demasiado, debido a su carencia de estructura o por el estilo de programación, hay algunas bastante ordenadas y bien estructuradas.

En las aplicaciones que se detectó una estructura bien definida se pudo apreciar que el *Javascript Framework* asociado tiene algún grado de relación con ésta.

En este punto se descarta *Polymer* [Goo16b], debido a que es evidente la mezcla de código *Javascript* y código HTML, lo cual no sigue para nada los estándares de la empresa SIDEKICK.

Aunque la aplicación transversal analizada ha resultado de ayuda para obtener nociones generales de qué esperar de cada *Javascript Framework*, no es posible descartar más de éstos sin revisar otros aspectos generales y específicos de cada uno ellos, los cuales se analizarán en la siguiente sección.

3.2. Revisión de aspectos generales y algunos específicos de cada Javascript Framework

Para la selección del *Javascript Framework* se utilizará una mezcla de criterios generales y específicos, basados en lecturas de opiniones como [Sid09] y [Sym16], y los requerimientos de

la empresa SIDEKICK. Éstos serán evaluados según la escala que se adjunta a cada uno de ellos.

3.2.1. Criterios o Aspectos Generales, escala y medición

1. Popularidad y tamaño de la comunidad. Escala: Pequeña - Mediana - Grande.
2. Soporte y resolución de problemas. Escala: Malo - Intermedio - Bueno.
3. Seguridad. Escala: Escasa - Intermedia - Avanzada.
4. Documentación. Escala: Mala - Intermedia - Completa.
5. Licencia y restricciones de uso. Escala: Con restricciones - Con algunas restricciones - Sin restricciones.
6. Patrones y filosofía de diseño. Escala: No utiliza - Algunas veces utiliza - Utiliza frecuentemente.
7. Actualizaciones: Escala: Poco frecuentes - Frecuentes - Muy frecuentes.

Para facilitar la visualización de los resultados, se enumerará la primera opción de cada escala con el número 0, la segunda opción con el número 1 y la última opción con el número 2. Para cada *Javascript Framework* se sumarán estos números generando una nota que puede variar entre 0 y 14, donde 0 será considerado Malo y 14 se considerará Bueno.

Para determinar cada una de las notas del Cuadro 3.3 se revisó las estadísticas de cada *Javascript Framework* en GitHub [GI16]; las documentaciones oficiales de cada uno; y algunas reseñas como [JH15], [IH15], [Bui15] y [Mit16].

Ya que la nota promedio aproximada de los 14 *Javascript Frameworks* es 10, se procederá a descartar a todos los que se encuentren dentro o bajo el promedio. Esto quiere decir: Ampersand [and16], Flight [Twi16], Canjs [Can16], Knockoutjs [SS15], Knockback [KM16], Mithril [LH14] y Troopsjs [MK16], no se seguirán revisando.

También se eliminará a Dojo [DF16], ya que luego de revisar en detalle la documentación no se trataría de un *Javascript Framework*, si no más bien de un biblioteca de funcionalidades las cuales pueden ser incorporadas en casi cualquier tipo de proyecto *Javascript*.

Por otra parte, no se detectó problemas con las licencias de cada *Javascript Framework*. En general, la documentación de cada uno son bastante completas y se pueden encontrar ejemplos e instrucciones sin mucha búsqueda.

Cuadro 3.3: Resumen de notas por JS Framework para los criterios de la sección 3.2

JS Framework	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**Nota**
Angularjs	2	1	2	2	2	2	2	13
React	2	1	1	2	2	2	2	12
Emberjs	2	2	1	1	2	1	2	11
Marionette	1	2	1	2	2	2	1	11
Dojo	1	2	2	2	2	1	1	11
Vuejs	1	2	2	2	2	1	1	11
Backbone	2	2	0	2	2	2	1	11
Ampersand	0	1	2	2	2	2	1	10
Flight	0	1	2	2	2	1	1	9
Canjs	1	2	1	2	2	1	0	9
Knockoutjs	0	1	1	1	2	2	1	8
Knockback	0	2	1	1	2	2	0	8
Mithril	0	2	1	1	2	1	0	7
Troopjs	0	0	0	0	2	1	0	3

Finalmente, las mayores diferencias entre cada *Javascript Framework* se vieron en los tamaños de las comunidades que los avalan y en la cantidad de contribuidores de cada uno de ellos. Por ejemplo, Emberjs [INC15] posee 604 contribuidores en GitHub [GI16], pero Troopjs [MK16] sólo tiene 6, una gran diferencia, la cual demuestra el nivel de interés por parte de programadores o desarrolladores.

3.2.2. Criterios o Aspectos Específicos

Para los criterios específicos de selección se utilizará los requerimientos solicitados por la empresa SIDEKICK, los cuales son fundamentales para cumplir con el objetivo de presentar una verdadera mejora tecnológica.

1. Compatibilidad con *Representational State Transfer Architecture*, más conocida como REST Architecture, y en el caso de los *Web Services* como RESTful Web Services [tut16].

2. Compatibilidad con Pruebas Unitarias [MR16b].

Los criterios anteriormente mencionados son excluyentes, esto quiere decir que si no se cumple cualquiera de éstos el *Javascript Framework* será descartado.

El Cuadro 3.4 muestra que no se ha encontrado problemas con los requerimientos específicos, por lo que no se ha descartado *Javascript Frameworks* utilizando estos criterios.

Cuadro 3.4: Resumen de cumplimiento de requerimientos por cada JS Framework de la Sección Emberjs

JS Framework	**C1**	**C2**
Angularjs	Sí	Sí
React	Sí	Sí
Emberjs	Sí	Sí
Marionette	Sí	Sí
Vuejs	Sí	Sí
Backbone	Sí	Sí

3.2.3. Compatibilidad Técnica con aplicaciones existentes en la empresa SIDEKICK y selección final del Javascript Framework

Hasta este punto se ha obtenido una lista de *Javascript Frameworks* con características deseables por la empresa SIDEKICK, utilizando criterios de descarte o selección bastante globales y algunos específicos.

Para la toma de la decisión final se considerará la compatibilidad técnica con aplicaciones ya existentes dentro de la empresa SIDEKICK, donde una de las tecnologías principalmente utilizadas es el *Framework Ruby on Rails* [RCT16] para la construcción de *Enterprise Resource Planning Applications* o ERP [MR16a].

Luego de algunas lecturas como [BH15], [ES16], [RL16], [TT16] y principalmente [KP16], [PP16b] y [ZK15], se puede argumentar que Vuejs [EY16] no es muy popular entre los usuarios del *Framework Ruby on Rails* [RCT16]. Por otra parte, React [Fac16] parece estar adquiriendo fuerza en la comunidad, pero aún se encuentra en una etapa temprana de desarrollo en comparación con los otros *Javascript Frameworks*, tal vez para el año 2017 o 2018 sea un opción bastante interesante de utilizar.

En las lecturas realizadas se menciona con bastante énfasis que Emberjs [INC15] se apega bastante a la filosofía de desarrollo del *Framework Ruby on Rails* [RCT16], por lo que, al parecer, es la mejor opción para probar e incorporar en la propuesta de mejora tecnológica para la empresa SIDEKICK.

Finalmente, cabe mencionar que Angularjs [Goo16a], Marionette [mar16a] o Backbone [Ash16a] son opciones igualmente buenas de probar como parte de la mejora tecnológica, pero se priorizará Emberjs [INC15] en un primer intento de integración con alguna de las aplicaciones de la empresa SIDEKICK.

Capítulo 4

Incorporación e Integración del Javascript Framework

4.1. Incorporación de Emberjs

De las aplicaciones disponibles en la empresa SIDEKICK, se ha decido utilizar ERP-SIDEKICK para la incorporación e integración del *Javascript Framework*, de ahora en adelante ERPS, ya que se encuentra en constante desarrollo y mejora.

Para poder hacer uso de las funcionalidades de cualquier *Javascript Framework* dentro de ERPS, se deben incluir los archivos de código fuente en sus dependencias o *assets*.

Normalmente, las dependencias de una aplicación construida con el *Framework Ruby on Rails* se administran a través de un gestor de paquetes llamado Bundler[AA16]. Éste permite la rápida y fácil integración de nuevas funcionalidades a ERPS, junto a un sencillo control de versiones de éstas.

Una alternativa al gestor de paquetes es incluir de forma directa las nuevas dependencias dentro de la aplicación realizando los enlaces que correspondan. El problema de utilizar esta alternativa es que se debe actualizar constantemente las dependencias de forma manual, en otras palabras, sobrescribir archivos de forma parcial o completa cada vez que sea necesario.

A continuación se expondrá la incorporación de Emberjs a ERPS, utilizando el primer método mencionado, y los problemas encontrados.

4.1.1. Incorporación de Emberjs utilizando Bundler

Para la incorporación de Emberjs en una aplicación *Ruby on Rails*, como ERPS, se puede utilizar el paquete *ember-cli*, el cual se puede incorporar siguiendo la documentación oficial[PP16a].

Aunque el proceso no debería tomar más de unos minutos, rápidamente surge el primer inconveniente. La última versión disponible de *ember-cli* (0.8.0) requiere tener instalado, y en consecuencia como dependencia de la aplicación, la versión 2.2.0, o superior, del lenguaje de programación Ruby[Mat16]. ERPS utiliza la versión 2.1.5 del lenguaje Ruby, por lo cual no es compatible con la versión 0.8.0 de *ember-cli*. De todas formas, se puede integrar Emberjs utilizando una versión previa de *ember-cli* (0.7.4).

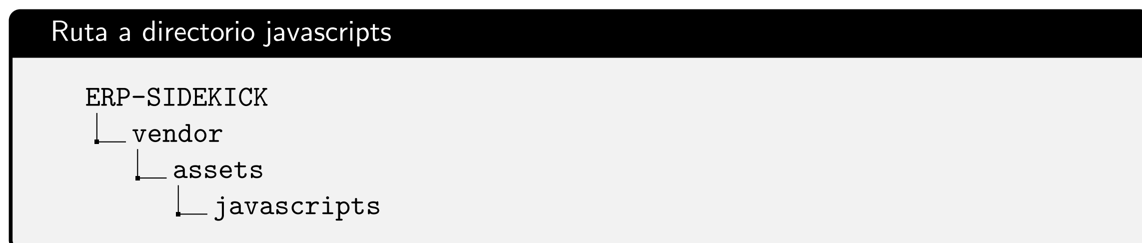
Continuando con la incorporación surge un segundo inconveniente: *ember-cli* crea una aplicación independiente dentro de ERPS asumiendo que la aplicación con la cual se está incorporando Emberjs es una API. Sin embargo, ERPS no es una API, por lo cual se dificulta la integración del flujo de trabajo propuesto por *ember-cli* con éste (ver [PP16a]), debido a que la interfaz *frontend* de ERPS ya se encuentra construida y su sustitución implicaría cambios importantes en la estructura de la aplicación.

Luego de algunos intentos fallidos de integración utilizando *ember-cli* se determinó que no se debe utilizar este paquete para incorporar Emberjs en ERPS, pero eventualmente se podría utilizar para aplicaciones que se estén construyendo desde cero.

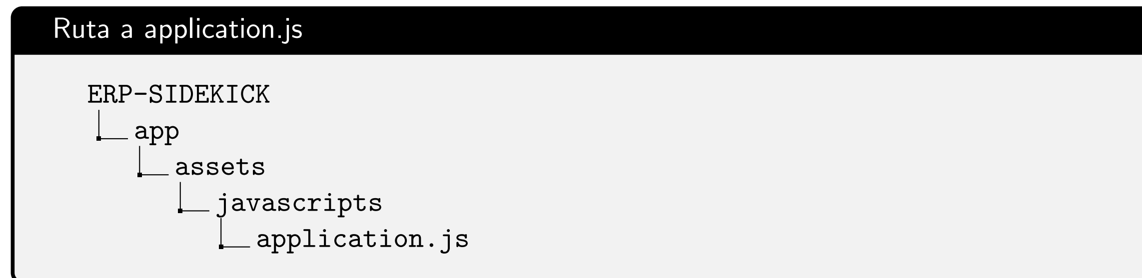
Finalmente, la mejor opción es incluir directamente los archivos de Emberjs en las dependencias de ERPS.

4.1.2. Incorporación directa de Emberjs como dependencia

Para incorporar Emberjs dentro de ERPS como una dependencia directa, o incluso cualquiera de los *Javascript Frameworks* ya mencionados, se deben copiar los archivos de código fuente, que se pueden descargar desde la página web oficial [INC15], en la siguiente ruta dentro de ERPS:



Una vez copiados los archivos, se debe especificar que efectivamente serán utilizados en la aplicación en el siguiente archivo:



Incluyendo las siguientes lineas de código:

```
//= require ember.js
//= require ember-data.js
```

Adicionalmente, Emberjs requiere de la dependencia Handlebars[Kat16] (descargable del sitio web oficial), como motor generador de plantillas:

```
//= require handlebars.js
```

Otra dependencia es jQuery[TjF16], pero ya se encuentra incorporada a ERPS.

Aunque el método descrito es bastante simple, se debe tener en consideración la posible aparición de nuevas versiones de Emberjs y de sus dependencias, para realizar actualizaciones de forma manual a los archivos correspondientes cuando sea necesario.

4.1.3. Estudio de la documentación de Emberjs para la integración con la aplicación escogida

Emberjs, siendo un *Architectural Javascript Framework*, ofrece un flujo de trabajo y arquitectura definidos para la construcción de aplicaciones *frontend*.

La figura 4.1 muestra el flujo de trabajo de una posible aplicación construida con Emberjs. Visitar la documentación oficial para información detallada de cada bloque expuesto en la figura [INC16].

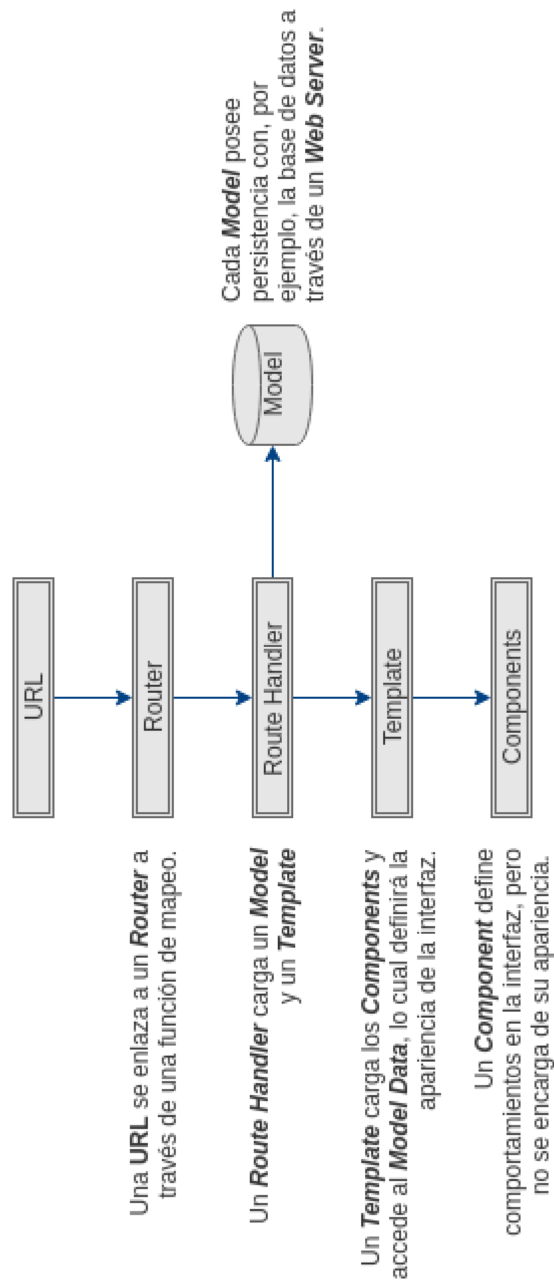


Figura 4.1: Flujo de trabajo en una aplicación Emberjs - Fuente: Elaboración propia

El flujo de trabajo funciona perfectamente si es que se está integrando Emberjs como única aplicación *frontend*, en otras palabras, toda la interfaz es construida utilizando Emberjs. Sin embargo, ERPS ya posee una interfaz *frontend* funcional y aunque es posible integrar Emberjs

haciendo ajustes a ERPS, la documentación no menciona cómo cambiar el flujo de trabajo o cómo adaptar Emberjs en el caso puntual de una aplicación que no necesite seguir el flujo de trabajo propuesto.

Otro inconveniente descubierto es que Emberjs utiliza Handlebars como único motor de plantillas. Este hecho implica una dependencia directa de un componente importante del *Javascript Framework*. En la documentación no se menciona si es posible cambiar Handlebars por otra opción, por ejemplo Underscore[Ash16c], lo cual podría perjudicar la futura integración de Emberjs en caso de perder el soporte de este motor.

Tampoco se documenta o explica el caso en que se quiera construir una aplicación híbrida (el caso de ERPS utilizando Emberjs) en la que se quiera incorporar plantillas sin usar la convención de *Uniform Resource Locator* o URL descrita en el flujo de trabajo propuesto por Emberjs.

Estos inconvenientes son causal suficiente para detener el estudio de Emberjs y volver a analizar la integración con otro de los *Javascript Frameworks* propuestos en la Sección 3.2.2.

4.2. Condiciones adicionales para la incorporación e integración de un Javascript Framework

A pesar de las revisiones realizadas en la Sección 3.2, los problemas expuestos en la Sección 4.1.3 dejan claro que se necesita flexibilidad adicional para integrar un *Javascript Framework* a una aplicación ya existente como ERPS.

Luego de revisar más en detalle las documentaciones de los *Javascript Frameworks* restantes de la Sección 3.2, tomando en consideración los problemas expuestos en la Sección 4.1.3, se ha determinado que Backbone y Marionette son los siguientes candidatos para un intento de integración con ERPS. Las razones son las siguientes:

1. En la documentación de Backbone [Ash16a] se menciona explícitamente que es posible realizar la integración con el *Framework Ruby on Rails*. En particular, los Modelos o Models[Ash16b], una abstracción de un objeto proveniente de la orientación a objetos, son perfectamente enlazables con cada URL que utiliza ERPS y que sigue los métodos *Create*(Crear), *Read*(Leer), *Update*(Actualizar) y *Delete*(Borrar), también conocido como CRUD, disponibles a través del protocolo HTTP por los métodos POST, GET, PUT y DELETE, respectivamente.

2. El punto anterior implica que Backbone sigue la arquitectura REST para su comunicación con otras aplicaciones y, en particular, permite desacoplar su funcionamiento de ERPS manteniendo una cantidad mínima de intervención en su código fuente, siempre y cuando ERPS siga la arquitectura REST para su comunicación. En caso de ser necesario se debe hacer ajustes para asegurar esto último.

3. Backbone utiliza Underscore como motor de plantillas, pero es posible utilizar otros motores en caso de ser necesario.

4. Aunque Backbone sigue la arquitectura MVC no entrega un flujo de trabajo definido como Emberjs, lo que ofrece flexibilidad en la definición de éste en la aplicación a desarrollar, algo necesario para integrar el *Javascript Framework* a ERPS.

5. Por otra parte, Marionette es un *Javascript Framework* que extiende a Backbone y agrega funcionalidades adicionales que facilitan la construcción de aplicaciones. Entre éstas está la definición de un módulo centralizado que da mejor estructura a la aplicación y la definición de regiones de trabajo dentro de las vistas.

En términos prácticos incluir Marionette en ERPS implica incluir Backbone también, por lo que el siguiente intento de incorporación e integración será realizado con Marionette.

4.3. Incorporación de Marionette

Tomando en consideración los problemas expuestos en la Sección 4.1.3, se procederá a incorporar Marionette como dependencia directa en ERPS para evitar nuevos inconvenientes.

4.3.1. Incorporación directa de Marionette como dependencia

De forma similar a la Sección 4.1.2, es posible descargar Marionette y sus dependencias desde la página web oficial de éste (ver [mar16a]). La estructura resultante de archivos es la siguiente:

Ruta a directorio javascripts y archivos agregados

```
ERP-SIDEKICK
└── vendor
    └── assets
        └── javascripts
            ├── backbone.js
            ├── backbone.babysitter.js
            ├── backbone.radio.js
            ├── backbone.wreqr.js
            ├── backbone.marionette.js
            ├── json2.js
            └── underscore.js
```

Cabe mencionar que no se incluye jQuery porque ya es parte de ERPS.

Una vez copiados los archivos, se debe especificar que efectivamente serán utilizados en la aplicación en el siguiente archivo:

Ruta a application.js

```
ERP-SIDEKICK
└── app
    └── assets
        └── javascripts
            └── application.js
```

Incluyendo las siguientes lineas de código:

```
//= require json2
//= require underscore
//= require backbone
//= require backbone.babysitter
//= require backbone.wreqr
//= require backbone.radio
//= require backbone.marionette
```

4.3.2. Estudio de la documentación de Marionette para la integración con la aplicación escogida

Durante el estudio de Marionette se determinó que no es necesario utilizar todas la funcionalidades y estructuras que ofrece el *Javascript Framework* para demostrar que se puede ofrecer una mejora tecnológica a la empresa SIDEKICK utilizando éste.

Particularmente, se estudió los siguientes conceptos de Marionette en la documentación oficial (ver [mar16b]):

1. Vistas(*Views*): Es un objeto (Colección o Modelo) que debe ser mostrado a través de una plantilla. En particular, las Vistas de Marionette incluyen el concepto de Vista Hijo que permite anidar de forma sencilla una vista con otra.
2. Eventos(*Events*): Permiten la comunicación entre objetos (Colecciones, Vistas y Modelos) a través de eventos. Éstos activan el accionar de los objetos que estén interesados en ellos.
3. Regiones(*Regions*): Permiten la administración de Vistas agrupándolas en un espacio de trabajo.
4. Aplicación(*Application*): Permite unificar la aplicación en una estructura común que a la vez ofrezca un punto de entrada a ésta.

Por otra parte, se estudió los siguientes conceptos de Backbone (debido a que es una dependencia directa) en la documentación oficial (ver [Ash16a]):

1. Modelos(*Models*): Son la abstracción de objetos del mundo real o de conceptos y poseen métodos para poder interactuar con los datos que se pueden almacenar dentro de ellos.
2. Vistas(*View*): Similar a las vistas de Marionette, pero con menor cantidad de funcionalidades.

Los aspectos correspondientes a la compatibilidad de la integración de estos elementos con ERPS fue revisada en la Sección 4.2, por lo que sólo basta proceder a la definición de una estructura de trabajo que permite crear una aplicación prototipo dentro de ERPS.

Capítulo 5

Construcción de la aplicación prototipo utilizando el Javascript Framework escogido

A partir de los conceptos estudiados, mencionados en la Sección 4.3.2, y las definiciones de funcionalidades relacionados a éstos, disponibles en las documentaciones oficiales [mar16b] y [Ash16a], se construyó una aplicación prototipo funcional simple que se acopla a ERPS.

Se tomó en consideración los siguientes aspectos para la construcción de la aplicación prototipo:

1. Simplicidad de la funcionalidad que integra o reemplaza.
2. Bajo impacto en la integración con ERPS, implicando que no se debe interrumpir el funcionamiento normal de las funcionalidades ya existentes.
3. La aplicación prototipo debe ser capaz de:
    a) Verificar que es posible implementar la arquitectura REST, tanto en una aplicación *frontend* como en ERPS sin perjudicar el desarrollo actual.
    b) Presentar una funcionalidad que se pueda comparar con el desarrollo actual de ERPS, esto implica que debe ser un cambio visible dentro de la interfaz *frontend*.
    c) Permitir detectar inconvenientes durante su desarrollo e integración con ERPS en caso de que existan.

5.1. Definición de la funcionalidad

Una de las funcionalidades más básicas de ERPS es la creación de nuevos Tipos de Tareas. La Figura 5.1 presenta los casos de uso pertinentes que involucran la intervención por parte del usuario.

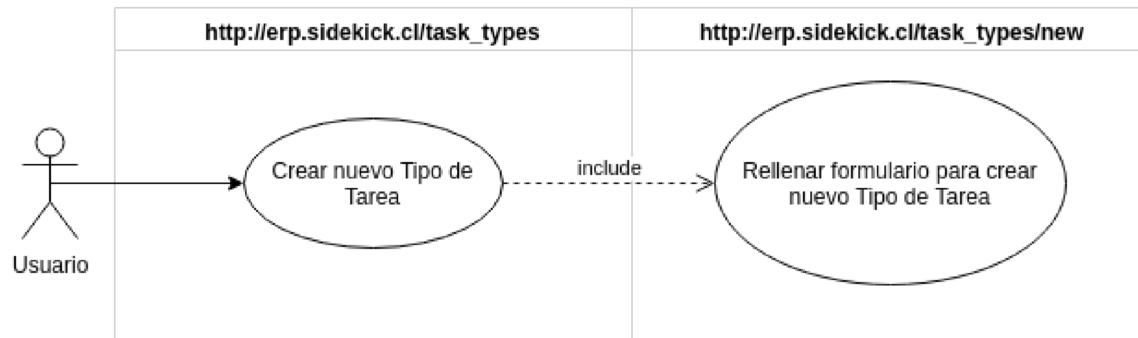


Figura 5.1: Caso de uso para la creación de un nuevo Tipo de Tarea - Fuente: Elaboración propia

En la Figura 5.1 se ha incluido la separación correspondiente al lugar donde ocurre cada caso de uso. El primero, *Crear nuevo Tipo de Tarea*, se inicia bajo del URL *http://erp.sidekick.cl/task_types* e incluye un segundo caso de uso, *Rellenar formulario para crear nuevo Tipo de Tarea*, el cual ocurre bajo del URL *http://erp.sidekick.cl/task_types/new*.

Normalmente, si en una página web dos casos de uso ocurren bajo dos URL distintos, esto conllevará a que se realizarán en dos páginas diferentes. Aunque lo anterior no representa grandes inconvenientes, implicará algunos percances:

1. Se debe cargar cada página web de forma completa cada vez que se accede a un URL distinto.
2. El tiempo que toma realizar tareas sencillas, como rellenar formularios, aumenta por los tiempos de espera y carga.

La Figura 5.2 presenta el cambio que realiza la aplicación prototipo, donde se agrupan ambos casos de uso bajo del URL *http://erp.sidekick.cl/task_types*.

Este cambio mitiga los inconvenientes anteriormente mencionados, pero implica un mayor tiempo de espera al entrar al URL *http://erp.sidekick.cl/task_types*, ya que junto a la página web se debe cargar Marionette y sus dependencias. Sin embargo, tareas repetitivas se ven enormemente beneficiadas por el cambio.

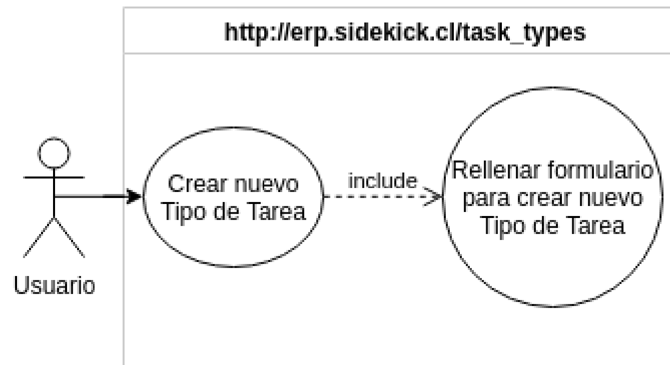


Figura 5.2: Caso de uso modificado usando Marionette - Fuente: Elaboración propia

5.1.1. Diagrama de secuencia del prototipo

La Figura 5.3 muestra el diagrama de secuencia del usuario con la interfaz *frontend* (que incluye la aplicación prototipo) y como ésta interactúa con la interfaz *backend* de ERPS, de forma resumida.

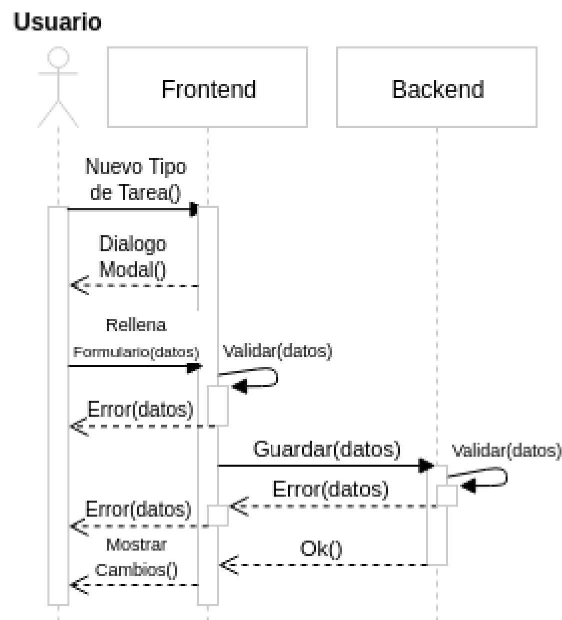


Figura 5.3: Diagrama de secuencia del prototipo - Fuente: Elaboración propia

5.2. Definición de la estructura y flujo de trabajo

5.2.1. Estructura de la aplicación prototipo

La estructura de archivos usada para ordenar la aplicación prototipo está basada en los conceptos mencionados en la Sección 4.3.2.

En la estructura se define directorios que agrupan a los Modelos(Models), Regiones(Regions), Plantillas(Templates), Utilidades(Utils) y Vistas(Views).

En particular el directorio *templates* posee un subdirectorio *task_types* (Tipos de Tareas) que agrupa las posibles plantillas del modelo Tipo de Tareas.

La lógica de agrupación de la aplicación prototipo es muy similar a la usada en ERPS, pero cabe mencionar que se puede idear otras estrategias de agrupación. Por ejemplo subestructuras por cada modelo o por región.

5.2.2. Flujo de la aplicación prototipo

La Figura 5.4 muestra, de forma general, el flujo de la aplicación prototipo con respecto a si misma, el usuario y ERPS.

Cabe mencionar que *DataTables*[Ltd16] es parte del *frontend* de ERPS y que permite la visualización de los Tipos de Tareas creados a través de la aplicación prototipo.

Por otra parte, la carga de datos realizada por el usuario y presentación de errores por parte de la aplicación prototipo, se realiza a través de un formulario modal, que es una alternativa

sencilla para presentar al usuario en reemplazo de la interfaz original de ERPS.

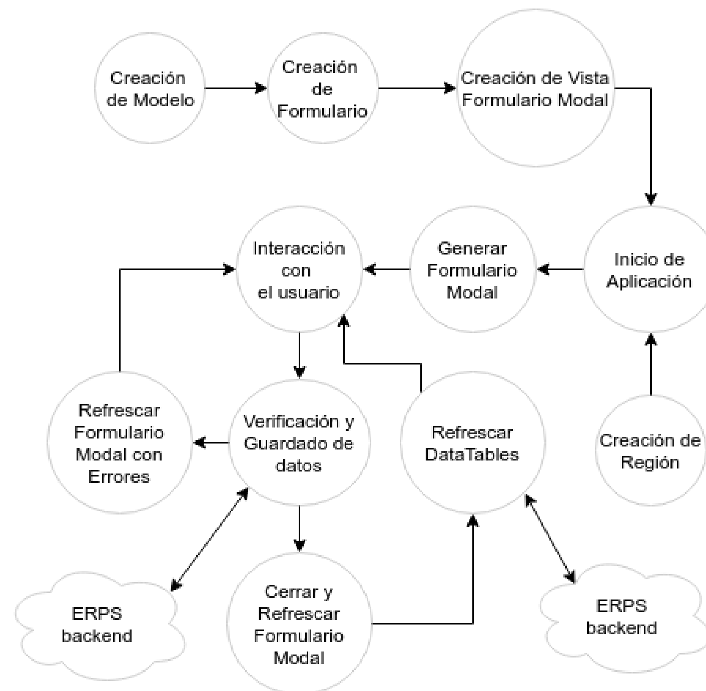


Figura 5.4: Flujo general de la aplicación prototipo - Fuente: Elaboración propia

5.3. Construcción de la aplicación prototipo

Para la construcción del formulario modal es necesario incluir algunas dependencias adicionales, disponibles en [Dav16], y son las siguientes:

- backbone-forms
- backbone-forms/list.js
- backbone-forms/templates/bootstrap3.js

Éstas son agregadas en el mismo directorio mencionado en la Sección 4.3, junto a la modificación del archivo pertinente (*application.js*) para terminar la incorporación a ERPS.

Utilizando las estructuras definidas, el flujo de trabajo propuesto y la funcionalidades de Backbone, Marionette, Underscore, jQuery y backbone-forms; es posible crear un formulario modal

funcional como aplicación independiente dentro del *frontend*, en otras palabras, la aplicación prototipo.

La Figura 5.5 presenta la visualización del formulario modal en la interfaz *frontend* de ERPS.

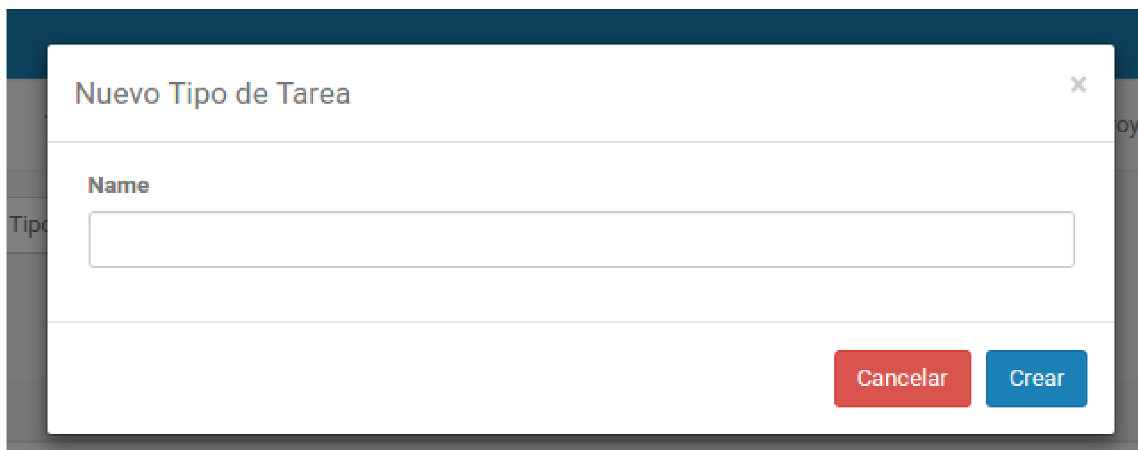


Figura 5.5: Formulario Modal generado por la aplicación prototipo - Fuente: Elaboración propia

5.4. Comparación de la aplicación prototipo con desarrollos existentes en la empresa SIDEKICK

Aunque la aplicación prototipo es bastante simple, ya que sólo es utilizada para validar la factibilidad técnica de la incorporación de un *Javascript Framework* y su integración en la interfaz *frontend* de ERPS, demuestra que es posible reemplazar y mejorar funcionalidades actuales de éste, sin interferir en el correcto funcionamiento de otras que no estén relacionadas.

Por otra parte, la aplicación prototipo desarrollada demuestra que es posible disminuir la cantidad de veces que se carga la página de tres veces a tan sólo una vez, en secuencias simples como la expuesta en la Sección 5.1, las cuales se pueden encontrar en muchas de las aplicaciones desarrolladas por la empresa SIDEKICK.

Implícitamente, el tiempo de respuesta de la interfaz *frontend* es mejor debido a que la comunicación con la interfaz *backend* de ERPS se limita a tan sólo verificar y guardar datos entregados por la aplicación prototipo, en vez de cargar páginas, extraer datos de éstas, verificarlos, guar-

darlos y finalmente mostrar los resultados, proceso actual de ERPS.

Sin embargo, aunque se pueden lograr más mejoras dentro de ERPS (en acciones simples como borrar elementos, por ejemplo) y otras aplicaciones de la empresa SIDEKICK, cabe mencionar que construirlas conllevará mayor tiempo de desarrollo de las interfaces de cada aplicación.

Capítulo 6

Conclusiones

Con respecto al proceso de selección e integración del *Javascript Framework* se puede concluir lo siguiente:

1. Existen muchos *Javascript Frameworks* distintos y se debe tener cuidado al seleccionar uno, ya que siguen arquitecturas y metodologías distintas que tal vez no se puedan adaptar fácilmente a la aplicación que se quiera desarrollar.
2. La lectura de código fuente y ejemplos puede ayudar a obtener nociones generales de cada *Javascript Framework*, pero de todas formas se debe revisar la documentación oficial para encontrar posibles problemas de compatibilidad.
3. Es conveniente seleccionar un *Javascript Framework* que se adapte a la tecnología que se utiliza, en vez de adaptarlo a ésta.

Por otra parte, con respecto a Marionette y la incorporación e integración a ERPS se puede concluir lo siguiente:

1. Se logró construir una aplicación prototipo funcional sobre la interfaz *frontend* de ERPS, capaz de demostrar que se pueden realizar mejoras funcionales de la interfaz utilizando la arquitectura REST y un *Javascript Framework*.
2. La aplicación prototipo también presenta mejoras de usabilidad y tiempo de respuesta en beneficio de los usuarios de ERPS.

Aunque existen beneficios por utilizar un *Javascript Framework* como Marionette, cabe mencionar que también existen algunos aspectos en contra:

1. Se necesita tiempo de desarrollo adicional para crear los modelos, vistas, plantillas, etc.

2. El tiempo de carga inicial de la aplicación aumenta, ya que se deben incluir las dependencias y código fuente adicional de las funcionalidades.

3. Durante el desarrollo de la aplicación prototipo se detectaron algunos problemas con las dependencias de Backbone, puntualmente la de formularios. Ya que ERPS es, principalmente, un gran conjunto de formularios, tener problemas con una dependencia relacionada a ellos puede significar grandes problemas de desarrollo y mantenibilidad de la aplicación.

Finalmente, durante el desarrollo de la memoria se ha logrado completar los objetivos propuestos en la Sección 1.2, se ha transmitido la investigación realizada, el prototipo y las conclusiones obtenidas a la empresa SIDEKICK, cumpliendo con el objetivo general de la memoria de proponer una mejora tecnológica utilizando un *Javascript Framework*.

Referencias

[AA16] Carl Lerche Yehuda Katz André Arko, Terence Lee. Bundler. https://rubygems.org/gems/bundler, 2016. Accedido: 3/9/2016.

[AI16] Apple Inc. Apple. http://www.apple.com, 2016. Accedido: 5-08-2016.

[ANAS14] André Nitze and Andreas Schmietendorf. Modularity of javascript libraries and frameworks in modern web applications. User Conference for Software Quality, Test and Innovation, At Klagenfurt, AT, Volume: 12, 2014.

[and16] andyet. Ampersandjs. https://ampersandjs.com/, 2016. Accedido: 3-08-2016.

[AO12] Addy Osmani. Understanding MVC And MVP (For Javascript And Backbone Developers). http://goo.gl/4b2iaH, 2012. Accedido: 25-12-2015.

[Ash16a] Jeremy Ashkenas. BackboneJS. http://backbonejs.org/, 2016. Accedido: 11-01-2016.

[Ash16b] Jeremy Ashkenas. BackboneJS. http://backbonejs.org/#Model, 2016. Accedido: 4/9/2016.

[Ash16c] Jeremy Ashkenas. Underscorejs. http://underscorejs.org/, 2016. Accedido: 4/9/2016.

[BH15] Blaine Hatab. 3 ways to integrate ruby on rails + react + flux. http://www.openmindedinnovations.com/blogs/3-ways-to-integrate-ruby-on-rails-react-flux, 2015. Accedido: 5-08-2016.

[Boo16] Phil Booth. escomplex. https://github.com/escomplex/escomplex, 2016. Accedido: 1/8/2016.

[Bui15] J. Buis. Security Comparison: AngularJS Vs Backbone.js vs Ember. https://softwaresecured.com/angularjs-backbone-js-and-ember-security-comparison/, 2015. Accedido: 3-08-2016.

[Can16] Canjs. Canjs. https://canjs.com/, 2016. Accedido: 3-08-2016.

[Dav16] Charles Davison. Datatables. https://github.com/powmedia/backbone-forms, 2016. Accedido: 4-09-2016.

[DF16] Dojo Foundation. Dojo. http://mootools.net/, 2016. Accedido: 2-08-2016.

[DIKS12] Audris Mockus Dag I. K. Sjoberg, Bente Anda. Questioning software maintenance metrics: A comparative case study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 107 – 110. IEEE, 11 2012.

[DR98] Dave Raggett. A history of HTML. https://www.w3.org/People/Raggett/book4/ch02.html, 1998. Accedido: 25-12-2015.

[EC14] ECMAScript Community. harmony:harmony. http://wiki.ecmascript.org/doku.php?id=harmony:harmony, 2014. Accedido: 25-12-2015.

[EI11] Ecma International. ECMAScript® Language Specification. http://www.ecma-international.org/ecma-262/5.1/, 2011. Accedido: 11-01-2016.

[EI15] Ecma International. ECMAScript® 2015 Language Specification. http://www.ecma-international.org/ecma-262/6.0/, 2015. Accedido: 11-01-2016.

[ES16] Eric Sipple. The Top Mistakes Developers Make Using Ember and Rails. https://www.airpair.com/ember.js/posts/top-mistakes-ember-rails, 2016. Accedido: 5-08-2016.

[EY16] Evan You. Vuejs. https://vuejs.org/, 2016. Accedido: 5-08-2016.

[Fac16] Facebook. React. https://facebook.github.io/react/, 2016. Accedido: 11-01-2016.

[fun16] fundinguniverse.com. Macromedia. http://www.fundinguniverse.com/company-histories/macromedia-inc-history/, 2016. Accedido: 2-08-2016.

[GI16] GitHub Inc. Github. https://github.com/, 2016. Accedido: 3-08-2016.

[Goo16a] Google. AngularJS. https://angularjs.org/, 2016. Accedido: 11-01-2016.

[Goo16b] Google. Polymer Project. https://www.polymer-project.org, 2016. Accedido: 2-08-2016.

[Gro16] Gregory Gromov. Roads and crossroads of the internet history. http://www.netvalley.com/cgi-bin/intval/net_history.pl?chapter=4, 2016. Accedido: 2/8/2016.

[IH15] Itay Herskovits. Does AngularJS Meet Enterprise Security Needs? http://blog.backand.com/angular-enterprise-dev/, 2015. Accedido: 3-08-2016.

[INC15] TILDE INC. Ember. http://emberjs.com/, 2015. Accedido: 11-01-2016.

[INC16] TILDE INC. Ember Guides. https://guides.emberjs.com, 2016. Accedido: 4/9/2016.

[JH15] John Hannah. Choosing the Right Javascript Framework for the Job. https://www.lullabot.com/articles/choosing-the-right-javascript-framework-for-the-job, 2015. Accedido: 3-08-2016.

[Kan16] Kangax. ECMAScript compatibility table. https://kangax.github.io/compat-table/es6/, 2016. Accedido: 11-01-2016.

[Kat16] Yehuda Katz. Handlebars. http://handlebarsjs.com/, 2016. Accedido: 3/9/2016.

[KM16] Kevin Malakoff. Knockbackjs. http://kmalakoff.github.io/knockback/index.html, 2016. Accedido: 2-08-2016.

[KP16] Katherine Pe. Among Angular, Backbone, Ember and React, which one is a better choice to be used with rails? http://goo.gl/6azIXE, 2016. Accedido: 5-08-2016.

[LH14] Leo Horie. Mithril. http://mithril.js.org/, 2014. Accedido: 3-08-2016.

[Ltd16] SpryMedia Ltd. Datatables. https://datatables.net/, 2016. Accedido: 4-09-2016.

[mar16a] marionettejs.com. Marionettejs. http://marionettejs.com/, 2016. Accedido: 5-08-2016.

[mar16b] marionettejs.com. Marionettejs. http://marionettejs.com/docs/current/, 2016. Accedido: 4-09-2016.

[Mat16] Yukihiro Matsumoto. ruby. https://www.ruby-lang.org, 2016. Accedido: 3/9/2016.

[Mic16] Microsoft. Microsoft. https://www.microsoft.com, 2016. Accedido: 5-08-2016.

[Mit16] Mithril. How is Mithril Different from Other Frameworks. http://mithril.js.org/comparison.html, 2016. Accedido: 3-08-2016.

[MK16] Mikael Karon. TroopJS. https://github.com/troopjs, 2016. Accedido: 2-08-2016.

[Moz15] Mozilla. ECMAScript 6 support in Mozilla. https://developer.mozilla.org/es/docs/Web/JavaScript/Novedades_en_JavaScript/ECMAScript_6_support_in_Mozilla, 2015. Accedido: 25-12-2015.

[Moz16] Mozilla. Mozilla. https://www.mozilla.org, 2016. Accedido: 5-08-2016.

[MR16a] Margaret Rouse. Erp (enterprise resource planning). http://searchsap.techtarget.com/definition/ERP, 2016. Accedido: 5-08-2016.

[MR16b] Margaret Rouse. unit testing. http://searchsoftwarequality.techtarget.com/definition/unit-testing, 2016. Accedido: 4-08-2016.

[NF16] Node.js Foundation. About Node.js®. https://nodejs.org/en/about/, 2016. Accedido: 11-01-2016.

[Ora16] Oracle. The History of Java Technology. http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, 2016. Accedido: 2-08-2016.

[OSA16] Opera Software ASA. Opera. http://www.opera.com/, 2016. Accedido: 5-08-2016.

[Osm16] Addy Osmani. Todomvc. http://todomvc.com/, 2016. Accedido: 1/8/2016.

[PCT16] Prototype Core Team. Prototype. http://prototypejs.org/, 2016. Accedido: 2-08-2016.

[PO92] J. Hagemeister P. Oman. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337 – 344. IEEE, 11 1992.

[PP16a] Sean Doyle Pavel Pravosud, Jonathan Jackson. ember-cli. https://github.com/thoughtbot/ember-cli-rails, 2016. Accedido: 3/9/2016.

[PP16b] Phips Peter. I am confused whether to use angular.js, underscore.js react.js, ember.js or backbone.js for my next project. can someone help me with material which states which type of web apps each library rocks at? http://goo.gl/h6n7e8, 2016. Accedido: 5-08-2016.

[RCT16] Rails Core Team. Ruby on Rails. http://rubyonrails.org/, 2016. Accedido: 5-08-2016.

[RL16] Richard LaFranchi. Vue.js and Rails. https://rlafranchi.github.io/2016/03/09/vuejs-and-rails/, 2016. Accedido: 5-08-2016.

[Sid09] Siddharth. 15 Most Important Considerations when Choosing a Web Development Framework. http://goo.gl/aeDZIU, 2009. Accedido: 3-08-2016.

[SS15] Steve Sanderson. knockoutjs. http://knockoutjs.com/, 2015. Accedido: 3-08-2016.

[Sym16] Symfony.com. 10 criteria for choosing the correct framework. http://symfony.com/ten-criteria, 2016. Accedido: 3-08-2016.

[tho16] thocp.net. Sun Microsystems. http://www.thocp.net/companies/sun_microsystems/sun_microsystems_company.htm, 2016. Accedido: 2-08-2016.

[TjF16] The jQuery Foundation. jQuery. https://jquery.com/, 2016. Accedido: 11-01-2016.

[TMDT16] The MooTools Dev Team. MooTools. http://mootools.net/, 2016. Accedido: 2-08-2016.

[TT16] Tim Tyrrell. Intro to Backbone.js with Rails. http://es.slideshare.net/timtyrrell/intro-to-backbonejs-with-rails, 2016. Accedido: 5-08-2016.

[tut16] tutorialspoint.com. Restful web services - introduction. http://www.tutorialspoint.com/restful/restful_introduction.htm, 2016. Accedido: 4-08-2016.

[Twi16] Twitter. Flight. https://flightjs.github.io/, 2016. Accedido: 3-08-2016.

[WHATWG16a] Web Hypertext Application Technology Working Group. HTML Introduction. https://html.spec.whatwg.org/multipage/introduction.html, 2016. Accedido: 11-01-2016.

[WHATWG16b] Web Hypertext Application Technology Working Group. HTML Living Standard. https://html.spec.whatwg.org/multipage/, 2016. Accedido: 11-01-2016.

[WHATWG16c] Web Hypertext Application Technology Working Group. Sitio web oficial. https://whatwg.org/, 2016. Accedido: 25-12-2015.

[WWWC97] World Wide Web Consortium. World Wide Web Consortium Publishes Public Draft of HTML 4.0. https://www.w3.org/Press/HTML4, 1997. Accedido: 25-12-2015.

[WWWC98] World Wide Web Consortium. Html 4.0 specification. https://www.w3.org/TR/1998/REC-html40-19980424/, 1998. Accedido: 11-01-2015.

[WWWC99] World Wide Web Consortium. Introduction to HTML 4.01. https://www.w3.org/TR/html4/, 1999. Accedido: 25-12-2015.

[WWWC05] World Wide Web Consortium. Document Object Model. http://www.w3.org/DOM/, 2005. Accedido: 25-12-2015.

[WWWC12] World Wide Web Consortium. A Short History of Javascript. https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript, 2012. Accedido: 25-12-2015.

[WWWC14] World Wide Web Consortium. HTML5 A vocabulary and associated APIs for HTML and XHTML. `http://www.w3.org/TR/html5/`, 2014. Accedido: 25-12-2015.

[WWWC15] World Wide Web Consortium. HTMl 5.1 W3C Working Draft. `http://www.w3.org/TR/html51/`, 2015. Accedido: 25-12-2015.

[WWWC16] World Wide Web Consortium. Sitio web oficial. `http://www.w3.org/`, 2016. Accedido: 25-12-2015.

[ZK15] Zach Kuhn. Choosing a front end framework: Angular vs. ember vs. react. `http://smashingboxes.com/blog/choosing-a-front-end-framework-angular-ember-react`, 2015. Accedido: 5-08-2016.