

Switch IDE

Borrador — Universidad Técnica Federico Santa María

Ian Murray Schlegel

*A los que creyeron en mí, a los que me apoyaron,
a todos los que hicieron esto posible.*

Índice

1. Introducción	1
2. Estado del Arte	3
2.1. Frameworks Actuales	3
2.1.1. Backbone	3
2.1.2. Cappuccino	4
2.1.3. Ext JS	5
2.2. Herramientas Actuales	6
2.2.1. Sencha Architect	6
2.2.2. Divshot	6
2.2.3. eXo Cloud IDE	8
2.2.4. Wavemaker	11
2.2.5. Zoho Creator	11
3. Propuesta de Solución	12
3.1. Elección de Herramientas para el Backend	12
3.2. Elección de Herramientas para el Frontend	13
4. Construcción de la Solución	15
4.1. Diseño de la Solución	15
4.1.1. Backend	15
4.1.2. Frontend	16
4.1.2.1. Definición de Objetos	17
4.1.2.1.1. Modelos y Colecciones	17
4.1.2.1.2. Vistas	18
4.1.2.2. Diseño del Editor de Templates	19
4.2. Primera Etapa de Construcción	20
4.2.1. Creación del Entorno de Trabajo	20
4.2.1.1. Entorno de Trabajo para el Backend	20
4.2.1.2. Entorno de Trabajo para el Frontend	22

4.2.2.	Prototipado de la Interfaz	23
4.2.3.	Creación de Servicios en el Backend	23
4.2.3.1.	Autenticación de Usuarios	25
4.2.3.2.	Creación de Proyectos	26
4.2.3.3.	Manipulación de Archivos	30
4.2.3.4.	Ensamblado y Servidor de Pruebas	32
4.2.4.	Agregado de Funcionalidad al Prototipo del Frontend	32
4.3.	Segunda Etapa de Construcción	35
5.	Resultados	39
6.	Conclusiones	40
6.1.	Sobre la Solución	40
6.2.	Sobre la Metodología	40
6.3.	Sobre la Elección de Herramientas	41
6.3.1.	Herramientas del Backend	41
6.3.2.	Herramientas del Frontend	41
6.4.	Características que Faltaron	41
6.5.	Trabajo Futuro	41
7.	Referencias	44

1. Introducción

En el contexto del desarrollo de aplicaciones web, existen dos grandes “corrientes”. Por una parte, es posible (y se utiliza muchísimo) desarrollar aplicaciones completamente de lado de servidor. Esto significa que la aplicación procesa todos los datos y genera todo lo que el usuario ve de forma remota. Esta forma de desarrollar ha sido así durante muchísimos años y sigue siendo una forma muy utilizada. Por otra parte, últimamente, con el crecimiento de la comunidad de Javascript, y la constante mejora en rendimiento de los navegadores modernos, se ha popularizado la idea de llevar gran parte de la lógica de negocio y el procesamiento de datos al cliente. Los navegadores modernos tienen cada vez más capacidad de ejecutar procesos rápida y eficientemente, lo que tiene una gran cantidad de ventajas: se alivia la carga en los servidores, lo que permite poder soportar a muchos más usuarios simultáneos, y las aplicaciones se desempeñan mucho mejor, dado que se disminuye el retardo que hay en transmitir datos entre el cliente y el servidor. Esto último es muy importante al momento de crear aplicaciones web. Si bien toda página web, sin importar su naturaleza, debería ser lo más rápida y responsiva posible, las aplicaciones web deben serlo por sobre todo. Al fin y al cabo, están intentando imitar el comportamiento de aplicaciones nativas, pero a su vez aliviando la carga a la que se somete un desarrollador al momento de crear aplicaciones compatibles con una infinidad de dispositivos y sistemas operativos distintos.

Desarrollar aplicaciones web versus desarrollar aplicaciones nativas (que son ejecutables directamente en el computador, sin necesidad de un navegador) tiene varias ventajas, siendo quizás la más importante que no es necesario escribir el programa para diferentes plataformas, dado que la mayoría de los navegadores modernos funcionan en una gran variedad de dispositivos. Además, con el crecimiento del estándar HTML5 (que al momento de escribir el presente documento aun se encuentra en proceso de convertirse en un estándar final), es posible aprovechar muchas características de los dispositivos (en algunos casos incluso es posible usar los acelerómetros de los teléfonos móviles inteligentes). Es más, muchos juegos han sido portados a la web, utilizando WebGL y tecnologías similares.

Ahora bien, desarrollar aplicaciones web también tiene sus desventajas. Es cosa de ver Xcode o Visual Studio, donde ambas herramientas son un entorno completamente integrado para desarrollar aplicaciones. Desde escribir código a crear formularios y diferentes tipos de vistas, ambas herramientas (y otras similares) entregan una experiencia casi inigualable al desarrollador. Al desarrollar aplicaciones y sitios web en general, no es posible encontrar herramientas que se asemejen lo suficiente a las mencionadas (y que sean de código abierto) como para considerarse una alternativa viable. La mayoría de los entornos de desarrollo para web permiten previsualizar lo que el desarrollador codifica, pero no le permiten ahorrar tiempo al momento de realizar tareas tan necesarias como codificar la interfaz de una aplicación.

Es este último aspecto el que se considera como un problema actualmente en el mundo del desarrollo web. Si bien no es difícil codificar interfaces de usuario al momento de crear aplicaciones web, es una tarea que consume mucho tiempo y para la cual sí existen

herramientas muy buenas en el mundo del desarrollo de aplicaciones nativas (como Xcode o Visual Studio). Es por esto que en este trabajo se propone la creación de un entorno de desarrollo integrado que permita la creación de aplicaciones web facilitando la creación de interfaces de manera similar a como lo hacen las herramientas ya mencionadas.

Este trabajo se estructurará como sigue:

- Se revisará primero el *estado del arte*, analizando las herramientas que actualmente intentan dar solución al problema identificado, además de frameworks y otro tipo de utilidades que mitigan de cierta forma el problema pero sin darle una completa solución.
- Luego se propondrá una solución al problema identificado, junto con métricas que permitirán cuantificar la efectividad de la solución creada.
- Se construirá y documentará la creación de la solución planteada, comentando en el proceso la efectividad de las herramientas escogidas.
- **NO SE AUN** Se mostrarán casos de uso de la aplicación creada de manera de mostrar su funcionamiento
- Se analizarán los resultados utilizando las métricas¹ previamente propuestas.
- Finalmente se presentarán conclusiones del trabajo junto con ideas para posible trabajo futuro.

¹Definí estas métricas??

2. Estado del Arte

La metodología para el desarrollo de aplicaciones web está cambiando. Ha pasado de estar enfocada casi completamente de desarrollar de lado de servidor, a desarrollar parcial o totalmente de lado de cliente. Frameworks como Backbone han revolucionado lo que se piensa sobre desarrollar aplicaciones completamente usando Javascript, y la aparición de muchísimos frameworks nuevos en este joven sub-mundo de aplicaciones muestra claramente una tendencia hacia este “paradigma”.

Ahora bien, el hecho de que periódicamente aparezcan nuevos frameworks no es necesariamente bueno. Es fácil perderse, no se puede saber por dónde empezar, y lo peor de todo, cada framework hace lo suyo de formas diferentes, incluso utilizando paradigmas de desarrollo distintos (ya sea MVC [1], MVP [2] u otro de los que normalmente se utilizan).

En este capítulo, se revisarán las diferentes herramientas que existen en el mundo del desarrollo de aplicaciones Javascript, además de programas y utilidades que funcionan de manera similar a lo que se quiere lograr con Switch IDE y que están actualmente en el mercado. Se revisarán primero diferentes frameworks disponibles hoy en día, analizando sus ventajas y desventajas, para luego mostrar herramientas que facilitan el uso de frameworks y otro tipo de soluciones online.

2.1. Frameworks Actuales

Existe una variedad enorme de frameworks para desarrollo web de lado de cliente, y, como se dijo anteriormente, día a día aparecen nuevos competidores, lo que pasó de ser algo bueno a algo que aumenta las barreras de entrada. El hecho de que haya tantas opciones para desarrolladores (incluso experimentados) hace que elegir uno sea muy difícil y que finalmente se opte por la solución incorrecta. Muchos frameworks tienen varios puntos fuertes, y no siempre un framework es la mejor solución para un tipo determinado de problema.

Ahora bien, sí existen buenos frameworks y varios de ellos son relativamente fáciles de entender y dominar. La mayoría de ellos llevan buen tiempo en el mercado y por ende tienen una comunidad fuerte y activa, junto con una base de código robusta.

2.1.1. Backbone

[Backbone](#) [3] es uno de los frameworks más populares. Basta con ver la gran cantidad de sitios que lo utilizan actualmente [4]. Es simple, extensible y muy poderoso, lo que lo hace una muy buena opción para desarrollar aplicaciones responsivas. Además, sus pocas dependencias hacen que las aplicaciones desarrolladas con él sean livianas.

El objetivo principal de Backbone es facilitar y dar estructura a aplicaciones que se basan fuertemente en funcionar del lado del cliente (es decir, en el navegador mismo). Normalmente,

escribir aplicaciones de este estilo es posible utilizando sólo Javascript y sin usar algún framework, pero ello resulta tedioso, y lleva a aplicaciones difíciles de mantener. Backbone (y la mayoría de los frameworks que se nombrarán en este capítulo) intentan evitar esto último dándole una estructura a las aplicaciones, separando vistas de controladores y modelos, y dejando las cosas en su lugar. Trae consigo facilidades para guardar información en servidores (en cierta forma proveyendo un “backend” a las aplicaciones que se creen). Además, facilitan la interacción con el usuario, a la larga ahorrando tiempo al desarrollador.

Es utilizado por un sinnúmero de proyectos, algunos muy populares, tales como [Groupon Now!](#), [Trello](#), entre otros [4]. Las aplicaciones nombradas no son proyectos pequeños y simples, sino que son aplicaciones muy poderosas que se benefician muy bien de lo que Backbone provee.

2.1.2. Cappuccino

[Cappuccino](#) [5] es un framework de desarrollo web enfocado en llevar “Cocoa” de Apple a la web, aunque no está en forma alguna afiliado con esta empresa. Abstrae completamente el desarrollo web a un único lenguaje: Objective-J, un superconjunto de Javascript (de la misma forma que Objective-C es un superconjunto de C). No cuenta con demasiados adeptos, dado que no muchos sitios lo utilizan, pero el framework sigue en constante desarrollo, aunque con una comunidad menor que la de Backbone, eso sí.

Las ventajas de este framework son varias. Al estar imitando bastante fuertemente a Apple, sigue varios estándares ya conocidos, y lo hace bastante fácil de aprender para una persona con experiencia en desarrollo iOS o Mac OS X. Además, todo se desarrolla con el mismo lenguaje, y trae integrados varios controles (botones, tablas, ventanas, menús), lo que le permite al desarrollador enfocarse sólo en código y no en el diseño (ver Figura 1). A diferencia de Backbone, en donde el desarrollador debe trabajar por un lado con el código y por otro con el diseño y los estilos de las aplicaciones, Cappuccino trae todo eso en un solo framework, de la misma forma que Visual C# trae sus controles y el diseño incorporados, por ejemplo.

Una de las características interesantes de este framework (aunque no es la que más se publicita), es la capacidad de usar el constructor de interfaces de [Xcode](#) [6] — Interface Builder — para crear las interfaces. Eso sí, no todos los componentes que están disponibles en Xcode están implementados en Cappuccino, y agregar elementos inexistentes no siempre arroja errores fáciles de descubrir. Además, desarrollar usando esta técnica, requiere instalar varios componentes (entre ellos, Xcode, que no es liviano) y se necesita un computador Mac, dado que Xcode no funciona en otras plataformas. Por si eso no fuera poco, el ir y venir entre Xcode y el editor que se use para Cappuccino hace que la experiencia sea todo menos placentera para el desarrollador.

Además de lo anterior, tiene otras desventajas. Por un lado, es necesario aprender un lenguaje nuevo (Objective-J) y utilizar un framework completamente distinto a todos los conocidos (casi todos están programados en Javascript directamente). Por otro lado, hay varios controles esenciales que no están implementados, como por ejemplo, un control de entrada de texto



Figura 1: Un ejemplo de los diferentes controles que vienen incluidos en Cappuccino. Puede apreciarse el diseño estilo OS X que trae.

multilínea no está soportado actualmente por el framework (aunque existen herramientas de terceros).

2.1.3. Ext JS

[Ext JS](#) [7] es un framework con bastante tiempo en el mercado. Tiene soporte para una gran variedad de componentes y es muy poderoso. Una de las ventajas importantes de este framework, es que existe una empresa bastante importante detrás: [Sencha Inc.](#) Esto asegura que el framework tiene soporte detrás, y que existe mucha gente preocupada constantemente de su desarrollo. Incluso, esta empresa ofrece soporte técnico pagado para este framework.

Este es un framework bastante poderoso, a la par e incluso por sobre Cappuccino y otros similares. Al llevar bastante tiempo circulando (desde el 2007 [8]), es un framework con una gran cantidad de componentes y características disponibles que lo hacen muy poderoso. Se diferencia en casi las mismas cosas con Backbone que Cappuccino, al tener integrados muchísimos componentes como botones, tablas, ventanas, entre otros, pero con la diferencia de que este framework sí está escrito en Javascript directamente, por lo que no hace falta aprender un lenguaje nuevo.

Ahora bien, tiene sus desventajas. Es un framework muy completo y dominarlo toma más tiempo que otros. No es un framework fácil de usar y no existe una gran comunidad detrás (no existen muchos sitios desarrollados con esta plataforma, al menos no sitios públicos), por lo que encontrar tutoriales e información al respecto no es tarea fácil. Además, una de las mayores desventajas, es que este framework no es gratuito para desarrollo comercial. Si se desea desarrollar una aplicación web y mantener el código propietario, se deben cancelar (por

lo bajo) \$329 dólares americanos [9], valor a veces prohibitivo considerando que la mayoría de los otros frameworks son gratuitos.

2.2. Herramientas Actuales

Ahora que se han visto una variedad de frameworks disponibles para desarrollar aplicaciones web, se procederá a analizar el mundo de las herramientas para el desarrollo de éstas. El enfoque de esta sección es analizar diferentes programas y servicios que se ofrecen, que son en alguna forma similares a lo que se quiere lograr con Switch IDE.

2.2.1. Sencha Architect

Si se tuviera que elegir una herramienta para describir lo que se quiere lograr con Switch, sería [Sencha Architect](#) [10]. Sencha Architect es una herramienta de los mismos creadores del framework Ext JS, y es lo que más se asemeja a lo que se quiere lograr con esta memoria. Es bastante poderosa, y permite desarrollar aplicaciones web y móviles (basadas en web, no nativas) de manera visual, de una forma bastante similar a lo que se quiere lograr con Switch IDE.

Sencha Architect es la herramienta que más se asemeja a un IDE para desarrollo de aplicaciones nativas (como Xcode o Microsoft Visual Studio). Posee una barra lateral con componentes (ver Figura 2, el número 1), un área central de trabajo (número 2) y propiedades de los diferentes componentes que existan en el área de trabajo (número 3).

Las ventajas de esta herramienta son claras: es posible crear las vistas de las aplicaciones directamente, arrastrando componentes. De la misma forma que herramientas para desarrollo nativo, esto ahorra tiempo al momento de desarrollar.

Las desventajas son varias, eso sí. Por un lado, el desarrollador está obligado a trabajar con Ext JS como framework (que, como ya se dijo antes, no es fácil de aprender y usar), y por otro lado, la licencia de uso de este software no deja de ser considerable: \$399 dólares americanos [11] al momento de escribir este documento. Esto es un monto realmente alto, pues si se compara esto con el precio de un editor de código relativamente bueno (como lo son TextMate o Sublime Text 2, ambos cercanos a los \$50 dólares), es una inversión de consideración. Además, a ese valor hay que agregarle otros \$329 por la licencia de uso de Ext JS [9], si es que se quiere para uso comercial.

2.2.2. Divshot

[Divshot](#) [12] es una de las herramientas que inspiró la presente memoria. Es una aplicación de prototipado rápido basado en web. Fue creada en abril de 2012, por lo que es una herramienta relativamente nueva y, de hecho, no está abierta al público aún. Se logró conseguir una

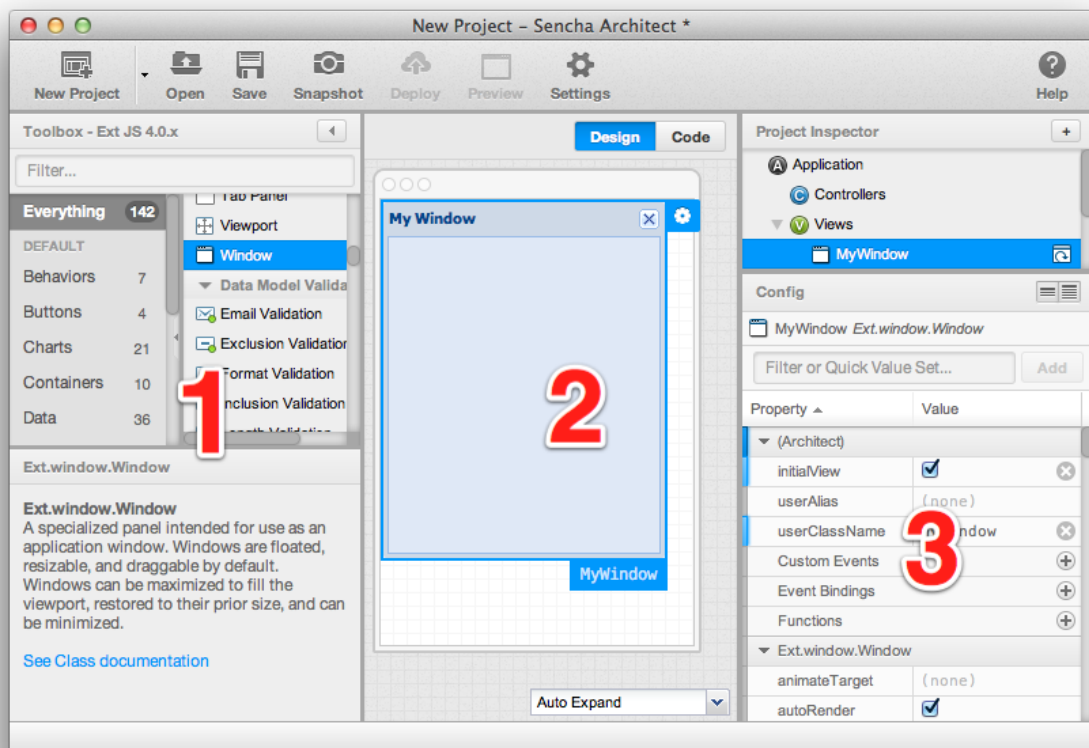


Figura 2: Sencha Architect: La ventana de trabajo

licencia de uso en esta fase para poder estudiar el poder de esta herramienta, y se llegó a la conclusión de que realmente tiene mucho potencial.

Permite, utilizando Twitter Bootstrap [13], crear prototipos de sitios y aplicaciones web arrastrando componentes, de la misma forma que IDEs como Xcode o Microsoft Visual Studio. En la presente memoria, se quiere lograr un comportamiento muy parecido para la creación de vistas, por lo que se podría considerar que Divshot es una de las inspiraciones más grandes para esta memoria.

Entre las ventajas que presenta la herramienta, está la facilidad con la que se pueden crear prototipos de vistas. Sólo arrastrando componentes, se puede llegar a una vista en pocos minutos. Además, es posible previsualizar los resultados fácilmente, e incluso exportar a HTML² con un sólo click. Lo mejor de todo, es que el HTML generado está muy bien ordenado y formateado.

Desventajas no tiene muchas. Es una herramienta muy puntual y bien diseñada, y, como aún está en fases de desarrollo, en constante mejora. Ahora bien, no es un “competidor” directo de Switch IDE, dado que no es una herramienta para programar. Simplemente permite crear vistas, que es sólo un componente de Switch.

2.2.3. eXo Cloud IDE

[eXo Cloud IDE](#) [15] es un entorno de desarrollo integrado, en la nube. Tiene varias características que lo hacen una buena opción al momento de querer colaborar o mantener el código alojado en internet. Soporta una gran variedad de lenguajes (entre ellos Java, Ruby y Python) y frameworks (como Ruby on Rails, Spring o incluso Google App Engine).

Además de lo anterior, soporta plataformas como Heroku para subir cambios a servidores directo desde el navegador, e incluso tiene soporte para versionamiento con Git [16].

Como cualquier IDE nativa estilo Xcode o Microsoft Visual Studio, tiene un visor de los archivos actualmente abiertos (ver Figura 4, número 1) y el editor de código mismo (número 2).

Las ventajas que provee esta herramienta son el no tener que depender de un sólo equipo para desarrollar. Al mantener el código en la nube, sólo es necesario conectarse al sitio web y empezar a trabajar. Por esto último, tampoco es necesario instalar las diferentes herramientas necesarias para desarrollar (como puede ser instalar Ruby o Python, que a veces es engorroso).

Ahora bien, tiene sus desventajas, siendo la más importante el hecho de que no permite desarrollar vistas de forma visual (que es lo que uno espera de una herramienta integrada de este estilo). Además, al no estar enfocado en un framework de desarrollo específico, no es muy poderoso al momento de elegir alguno (en términos de autocompletado de código por

²**Hypertext Markup Language:** es el lenguaje de etiquetas utilizado para las páginas web. Todas ellas están hechas con esto, más una combinación de otros lenguajes. [14]

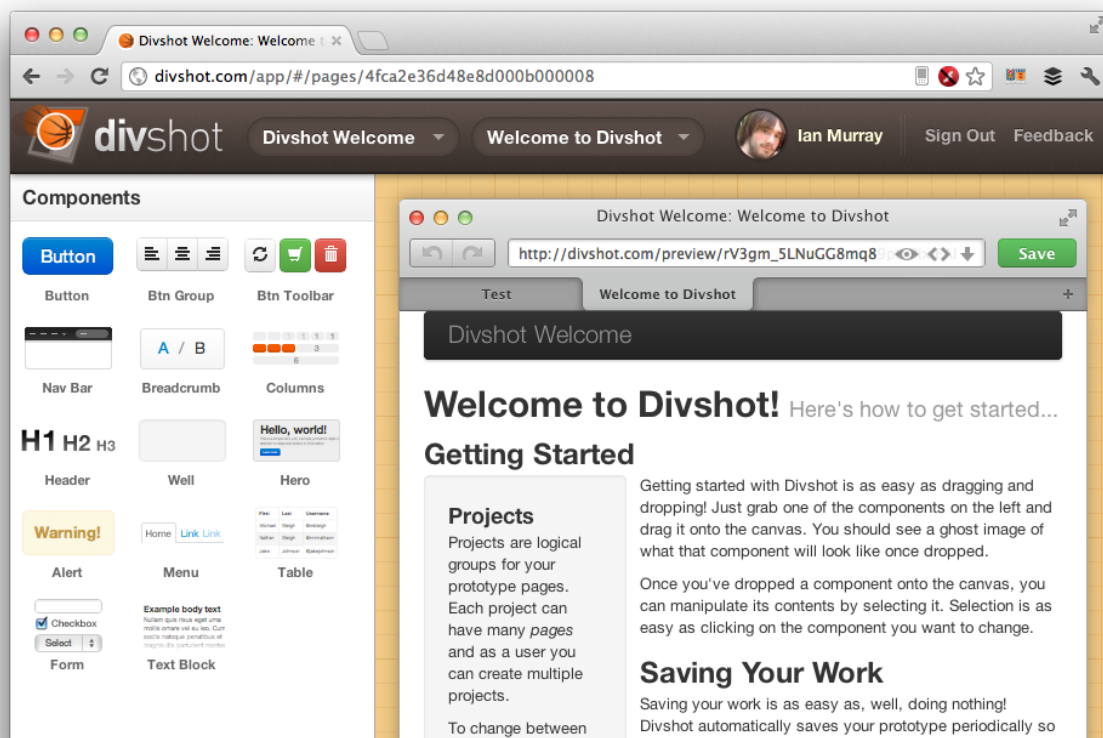


Figura 3: La ventana principal de Divshot. Se pueden apreciar los componentes en la barra lateral izquierda, con la vista a la derecha, donde se arrojan los componentes.

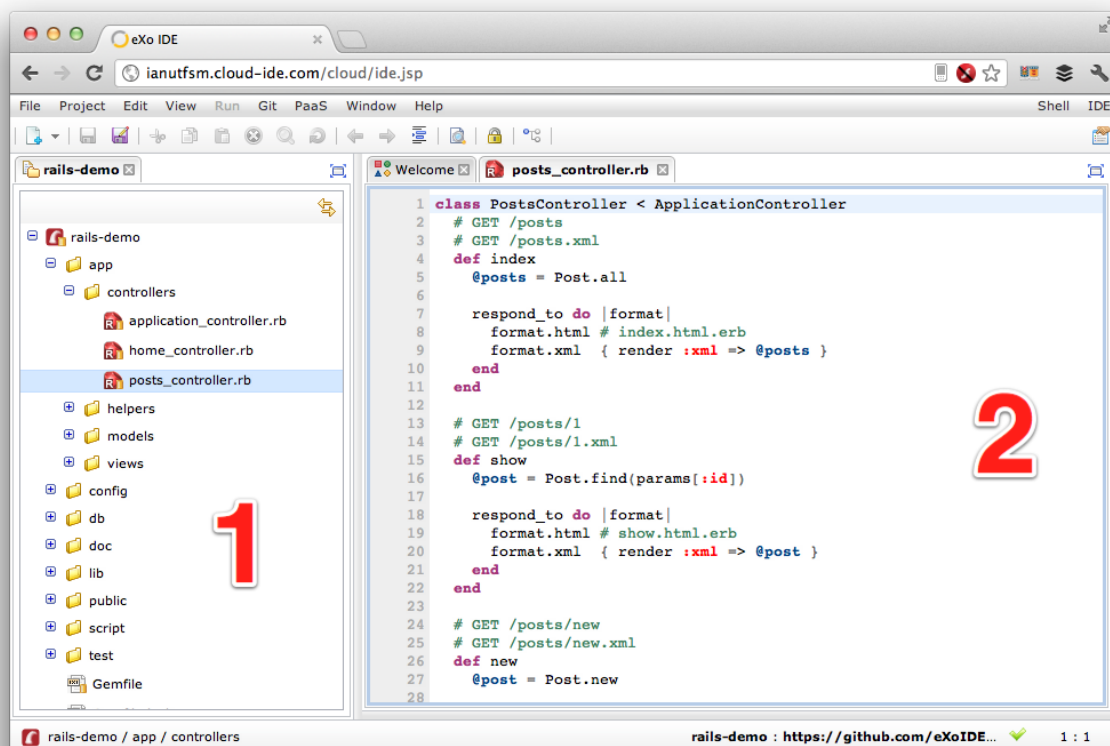


Figura 4: eXo Cloud IDE: un entorno de desarrollo integrado, en la nube

ejemplo). Relacionado con esto último, no permite desarrollar aplicaciones de lado de cliente, sólo de lado de servidor, por lo que también está limitado por ese lado. Además, desarrollar aplicaciones con los frameworks que soporta por lo general implica utilizar mucho el terminal de comandos para realizar pruebas, y en un ambiente web, eso no es fácilmente replicable.

2.2.4. Wavemaker

[Wavemaker](#) [17] es una herramienta que permite generar “bases de datos” de manera visual. El objetivo principal de esta aplicación es poder crear sistemas de manejo de datos de manera fácil, escribiendo la menor cantidad de código posible.

Esta herramienta se asemeja más a lo que es Microsoft Access. La idea es poder crear interfaces para guardar y organizar información rápidamente y sin programar. Es una aplicación enfocada más a gente que no tiene conocimientos de programación, aunque según los creadores de Wavemaker, las aplicaciones que se generan son aplicaciones Java completas, que pueden ser extendidas más adelante programando Java directamente.

Las ventajas dependen realmente de lo que se quiera hacer con la herramienta. Si el usuario no tiene conocimientos en el área de desarrollo de software, entonces la herramienta es bastante útil, aunque los resultados no son muy complejos. Por otro lado, si se le mira desde el punto de vista de un desarrollador, la herramienta no ofrece mayores beneficios a las anteriormente mencionadas.

2.2.5. Zoho Creator

[Zoho Creator](#) [18] es un servicio web que tiene más o menos el mismo enfoque que Wavemaker: crear bases de datos simples sin programar. Ahora bien, se diferencia un poco con Wavemaker en el sentido de que es un servicio, pagado, que permite crear aplicaciones en la nube y mantenerlas ahí, mientras que la herramienta anterior permite crearlas y exportarlas, para luego incluso extenderlas si es que se tienen conocimientos.

Si bien Zoho Creator es una herramienta relativamente poderosa y fácil de usar, no es una herramienta que utilizaría un desarrollador. Hay varias razones para esto último. Primero, no permite exportar lo creado para alojarlo en otro servidor; segundo, no es una herramienta de desarrollo que permita crear aplicaciones de lado de cliente o de servidor, sino que permite crear simples bases de datos; y tercero, no es una herramienta gratuita o de código abierto, lo que hace que lo creado con ella tenga ciertas limitaciones en cuanto a licencias.

3. Propuesta de Solución

Para solucionar el problema identificado, se propone una herramienta web, basada en *Backbone* y *Ruby*, que permita editar una aplicación web completa en el mismo navegador. Sus principales características serían:

- Dar una estructura a las aplicaciones que se desarrollen con Backbone
- Facilitar la creación de vistas mediante un editor que permita arrastrar los diferentes componentes y ahorrar el tiempo gastado en ello
- Implementar atajos de teclado de manera similar a cómo lo haría una IDE de escritorio
- Permitir ensamblar y probar la aplicación de manera similar a cómo lo haría un programa nativo

La herramienta no puede funcionar sólo de lado de cliente (exclusivamente en el navegador), dado que algunas funciones deben ser ejecutadas en el servidor, como por ejemplo la compilación de los archivos Javascript y levantar una instancia de un servidor para probar lo que el usuario esté desarrollando. Por esto, es necesario implementar este proyecto en dos partes distintas, pero dependientes.

Por un lado, se tiene el servidor, que tendrá la tarea de manipular los archivos de cada proyecto que el usuario cree y desarrolle. Además, tendrá la tarea de ejecutar ciertos comandos necesarios para la creación y prueba de los proyectos, que simplemente no es posible ejecutar en el navegador. El hecho de que exista un “backend” (como se le llamará de ahora en adelante), permite además dar lugar a futuras características imposibles de llevar a cabo de otra forma, como por ejemplo soporte para repositorios Git, que deben ser manejados en el servidor exclusivamente.

Por otro lado, está el cliente. El cliente es básicamente el “frontend” (como se le llamará de ahora en adelante). Es la interfaz gráfica y es lo que interactúa con el usuario, en el navegador. La mayor parte del trabajo recaerá en esta parte, dado que es lo que el usuario ve y utiliza para trabajar. Además, es la parte del proyecto que contendrá el editor de vistas, lo que requerirá un esfuerzo no mínimo para funcionar.

3.1. Elección de Herramientas para el Backend

Para desarrollar el backend se escogió el lenguaje de programación *Ruby*. Las razones para la elección de este lenguaje son varias: la familiaridad que tiene el autor con éste; su simplicidad para desarrollar tareas relativamente complejas en otros lenguajes; la infinidad de librerías

disponibles para facilitar diferentes tareas (como por ejemplo *Grit*, una librería para manejar repositorios Git directo desde Ruby)³.

Para desarrollar el backend no se utilizará Ruby puro. Lo que se quiere desarrollar en el lado del backend es básicamente una API (Application Programming Interface), que el frontend utilizará para funcionar correctamente. Existen varias formas de programar APIs en Ruby, donde las más populares son Ruby on Rails y Sinatra.

La primera — Ruby on Rails — ha comenzado a ser muy popular en estos últimos años por ser un framework extremadamente completo. Facilita enormemente una infinidad de tareas que lo hacen una herramienta ideal para todo tipo de proyectos web. Sin embargo, al ser una herramienta tan completa, agrega bastante *overhead*. Además, es una herramienta ideal para crear proyectos grandes y muy complejos, y dado que el backend de el presente trabajo no requerirá tanta complejidad, se convierte en una alternativa no tan ideal para este trabajo.

La segunda — Sinatra — podría considerarse el hermano menor de Ruby on Rails. Es un framework muchísimo más simple, y, por ende, mucho más liviano. Inicia más rápido y, en general, se desempeña mejor que su contraparte. Tiene la desventaja de no poseer tantas facilidades para desarrollar sitios complejos y grandes, pero dado que el backend de este proyecto no requerirá tanto trabajo, es la opción ideal.

3.2. Elección de Herramientas para el Frontend

Dado que la solución que se propone es una herramienta para desarrollar aplicaciones en Backbone utilizando Twitter Bootstrap, lo lógico es desarrollar esta solución usando las mismas herramientas. Se escogió Backbone dada su alta popularidad. Esto asegura que el framework es bastante sólido y que existen variadas librerías y herramientas estables para él. Por otro lado, si bien Backbone es un framework poderoso, es bien simple, lo que da más libertad sobre como afrontar diferentes problemas.

Ahora, Backbone es un framework que no da una estructura a las aplicaciones que se desarrollan con él. A diferencia de Ruby on Rails, por ejemplo, la tarea de estructurar la aplicación en carpetas o módulos depende completamente del desarrollador. En este punto se asemeja mucho más a Sinatra que a Ruby on Rails. Esto puede considerarse una desventaja, dado que sistemas complejos tienden a crecer y desordenarse bastante si no se aplica una estructura desde un principio. Por esto, es que se decidió utilizar Brunch⁴. Brunch es un ensamblador de aplicaciones (“application assembler” en inglés). Básicamente, basándose en un esqueleto, organiza aplicaciones en carpetas. Soporta diferentes frameworks y lenguajes, desde Backbone a Knockout⁵, usando Javascript o Coffeescript⁶. Además de entregar una estructura, las ensambla, es decir, toma todos los archivos y los junta en uno sólo de manera

³Cita?

⁴Poner referencia a esto!

⁵Referencia!

⁶Referencia.

de optimizar la aplicación cuando esté en producción. Es más, incluye un servidor web de desarrollo, que detecta cambios en los archivos y reensambla todo el sitio de manera de poder hacer pruebas más rápida y fácilmente.

En lo que respecta la decisión de lenguaje de programación, no hay muchas alternativas. Es posible desarrollar la solución usando Javascript o Coffeescript. Existen otras alternativas, pero al momento de escribir este documento no se encuentran en etapas estables de desarrollo ni madurez. Javascript es un lenguaje poderoso pero a la vez de relativo bajo nivel. Ciertas cosas son un tanto tediosas de programar, como recorrer arreglos por ejemplo. En cambio, Coffeescript, un lenguaje de programación escrito por Jeremy Ashkenas que apareció en el 2009⁷ que compila a Javascript, hace que desarrollar en Javascript sea mucho más cómodo y simple, sin perder desempeño ni funcionalidad (pues compila directamente a Javascript). Además, hace muchísimo más fácil programar “orientado a objetos”. Si bien ECMAScript (la base de Javascript) está definido como orientado a objetos,⁸ su estilo es diferente al resto de los lenguajes. Coffeescript agrega palabras clave como `class` y `extends` de manera de facilitar escribir clases y manipular objetos en este lenguaje.

Por último, para el diseño y estructura de la interfaz, se decidió utilizar *Twitter Bootstrap*. Bootstrap es un conjunto de componentes y un sistema de estructurado para páginas web, que hace la tarea de crear y diseñar un sitio muy simple. Trae consigo una enorme cantidad de componentes (botones, barras de progreso, etc.), además de facilitar el diseño de formularios y componentes similares muy comunes al momento de crear aplicaciones web. No sólo se utilizará Twitter Bootstrap para crear el frontend, sino que también se utilizará para el diseñador de interfaces, dado que tiene todas las ventajas descritas anteriormente.

⁷<https://github.com/jashkenas/coffee-script/commit/8e9d637985d2dc9b44922076ad54ffef7fa8e9c2>.

⁸<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf> página 1.

4. Construcción de la Solución

Este capítulo tiene por objetivo detallar todo el proceso del diseño y desarrollo de la solución propuesta anteriormente. Se dividirá en los siguiente subcapítulos:

- a. Diseño de la solución: básicamente se explicará cómo ambas componentes (frontend y backend) interactuarán entre sí. Además, cómo funcionarán ambas partes en términos de manipulación de archivos y el proyecto completo. Por último, se explicará cómo se diseñó el componente principal de la solución (el editor de interfaces).
- b. Flujo del usuario: acá se explicarán algunos de los casos de uso de la aplicación, como el inicio de sesión, o el ensamblado de los proyectos.
- c. Primera etapa de construcción: el desarrollo de la solución se dividió en dos etapas principalmente. Primero, se desarrolló lo que se denominó una “base” del programa. Esta etapa contempló el desarrollo de gran parte del backend y, en el frontend, una herramienta que permitiera crear proyectos nuevos, crear, editar y eliminar archivos, compilar y correr el proyecto.
- d. Segunda etapa de construcción: la segunda parte del desarrollo se enfocó en desarrollar y perfeccionar el editor de interfaces. Dado que este componente es el grueso de la solución, se decidió dedicar una etapa completa a él.

4.1. Diseño de la Solución

4.1.1. Backend

La solución es una aplicación mayoritariamente de lado de cliente, por lo que la mayor cantidad de lógica debe ir en este lado. Por esta razón, se diseñó el servidor de la forma más simple posible. Las tareas principales que tiene el servidor o backend de la aplicación son:

- Autenticar usuarios
- Generar proyectos utilizando Brunch
- Manipular los archivos. Esto incluye crear, eliminar, renombrar y actualizar archivos (o sea, recibir el contenido de ellos y guardarlo a disco)
- Ensamblar el proyecto
- Levantar un servidor estático que permita al usuario probar su proyecto

Algunas de estas tareas son realizadas por la utilidad Brunch, por lo que sólo es necesario hacer que el servidor ejecute un comando en la consola para llevarlas a cabo. El resto de las tareas son básicamente manipulación de archivos, para lo cual cualquier lenguaje de programación trae funciones o librerías (y Ruby no es ninguna excepción).

Se decidió utilizar una aproximación a lo que es REST⁹ para los servicios que proveerá el backend. En REST, la idea es representar objetos, y exponer diferentes métodos para tales objetos. En este caso, se decidió que deben existir 3 objetos diferentes: usuarios, proyectos y archivos.

- Usuarios: dado que la idea es que varios usuarios puedan utilizar el sistema a la vez, debe existir esta entidad en el servidor (y por ende en la base de datos).
- Proyectos: es casi la base de todo. Cada proyecto contendrá los diferentes archivos y carpetas, y pertenecerá a un usuario.
- Archivos: esto se refiere a archivos y carpetas. Por la forma en la que se manipulan los archivos en el servidor y en el cliente, es más conveniente manejarlos de (casi) la misma forma. Esto último se refiere a la forma en la que se obtienen los archivos en el servidor, y las acciones que se realizan en ellos. Ambos archivos y carpetas se crean y eliminan, como también se renombran. La única diferencia substancial es que las carpetas no tienen contenido y no se actualizan como el resto de los archivos.

El servidor autentificará a los usuarios utilizando sus cuentas de GitHub y OAuth. OAuth es un protocolo de autenticación y autorización que permite al usuario registrarse e ingresar a la aplicación con un sólo click (dos si ingresa por primera vez). Dado que esta es una herramienta enfocada a programadores, y considerando que GitHub es una plataforma conocida por cualquier desarrollador que se mantenga al día, utilizar este sistema para autentificar a los usuarios es simple y conveniente. Por el lado de servidor, basta con incluir una librería que redirige a los usuarios a las URLs específicas y crear un registro en la base de datos si es que el usuario está ingresando por primera vez.

4.1.2. Frontend

El frontend tendrá dos tareas principalmente. Primero, deberá permitir a los usuarios autenticarse. Para esto, y para efectos del presente trabajo, será una simple página web que redirija al usuario a GitHub para autenticarse usando este sistema. Segundo, deberá proveer al usuario con la IDE que se quiere construir en este documento.

EXPLICAR DE QUÉ TRATA LA PÁGINA DE AUTENTIFICACIÓN A GRANDES RASGOS, SCREENSHOTS Y TODO

⁹Explicar esto.

La IDE misma se dividirá en tres componentes principales: la barra lateral izquierda para explorar los archivos del proyecto, la barra lateral derecha con componentes visuales (como botones, formularios, etc.), y la sección central que contendrá el código del archivo actualmente seleccionado o la vista previa en caso de estar editando una vista.

Contará además con una barra superior con un menú (al igual que cualquier aplicación de escritorio), pero dado que proveerá simples accesos directos a funciones que se explicarán más adelante no se detallará su diseño ni implementación.

4.1.2.1. Definición de Objetos

En esta sección se pretende explicar qué objetos existirán en el frontend, sus responsabilidades y cómo interactuarán entre ellos. Primero se definirán algunos conceptos necesarios para entender de qué tipos de objetos se estará hablando.

Modelo Representa un objeto en un proyecto, como por ejemplo un archivo, una carpeta o el proyecto mismo. Cada modelo es responsable de persistir su estado de alguna forma (comunicándose con un servidor o almacenando datos en el mismo navegador).

Colección Es básicamente una lista de instancias de un tipo de modelo. Por ejemplo, una carpeta podría considerarse una colección de archivos (siendo cada archivo una instancia de un modelo).

Vista Una vista en Backbone es un archivo que se encarga de presentar información al usuario, y además de interactuar con él, por ejemplo ejecutando funciones cuando se haga un click en un botón. Una vista por lo general presenta un modelo (o una colección). Por ejemplo, se puede tener una vista para cada instancia de un archivo, o bien se pueden tener vistas que no presenten a ningún modelo en particular.

Template Un template es un trozo de HTML que una vista utiliza para generar lo que el usuario ve. Si bien no son enteramente necesarias y una vista podría generar todo lo que necesita con Javascript, hacen la tarea algo más fácil. Contrario a lo que pueda suponerse, el editor visual en el que se trabajará en este documento editará los templates, y no las vistas.

4.1.2.1.1. Modelos y Colecciones

A continuación se explicará a grandes rasgos los modelos y colecciones que existirán en el frontend. Se tendrán modelos para los proyectos y los archivos. Cada uno se encargará de comunicarse con el backend para obtener los datos que le sean necesarios o bien para guardar los cambios.

El modelo de proyecto Guardará el nombre de éste y una referencia a una colección de los archivos que se encuentren en la raíz de su carpeta. Tendrá como responsabilidades

crear archivos y carpetas, ensamblar el proyecto e iniciar el servidor de pruebas. Estas acciones se complementan con llamadas al backend que realizan las tareas mismas.

El modelo de archivo Se encargará de guardar el nombre y el contenido (en caso de que corresponda) del archivo o carpeta al que representa, además de guardar una referencia al proyecto al que pertenece y . Tendrá como responsabilidades pedir su contenido, actualizarlo, renombrar y eliminar el archivo del sistema. Todas estas acciones se complementan además con llamadas al backend.

La colección de archivos Tendrá como responsabilidad ordenar las listas de archivos una vez que la haya obtenido (además de guardar una referencia a cada modelo de archivo que le corresponda). Ordenar las listas de archivos es importante pues el backend arroja una lista de archivos ordenada alfabéticamente, pero, dado que archivos y directorios son considerados de la misma forma en el backend, es necesario ordenar la lista de manera que los directorios queden arriba. Esto facilita encontrar archivos para el usuario.

4.1.2.1.2. Vistas

Cada una de las siguientes vistas considera un template asociado.

Explorador de Archivos se colocará en la barra lateral izquierda, y tendrá dos listas de archivos. Una lista de archivos actualmente abiertos y la lista de archivos y directorios en el proyecto. Tendrá entre sus responsabilidades mantener una lista de archivos que se encuentran actualmente abiertos para que el usuario pueda navegar entre ellos.

Archivo Esta vista representará a un archivo en la vista anterior (explorador de archivos). Mostrará su nombre y un icono que represente si es un directorio, un archivo o una vista editable con el editor que se construirá. Entre sus responsabilidades están abrir los archivos (o sea, abrir el archivo en el editor de código o en el editor de vistas en caso que corresponda), embeber listas de archivos en caso de que se clickee un directorio y permitir al usuario renombrar archivos, mostrando un menú contextual.

Editor de Texto Esta vista contendrá el editor de archivos de texto (editor de código). Sus responsabilidades serán mostrar un editor con resaltado de sintaxis y modificar el modelo de archivo que corresponda para guardar cambios.

Editor de Vistas esta vista mostrará templates y, en una barra lateral derecha, diferentes componentes para que el usuario los arrastre y agregue. Contará además con un editor de código HTML, en caso de que el usuario quiera editar la vista o realizar cambios que el editor no permita directamente. Tiene las mismas responsabilidades que el editor de texto.

4.1.2.2. Diseño del Editor de Templates

El editor de interfaces se construirá de manera que el usuario pueda arrastrar componentes como botones o campos de texto directamente en una vista previa del template que esté editando. El contenido de los archivos de templates es simplemente HTML, por lo que es posible presentarlos directamente en la aplicación. Este contenedor o vista previa del template se le llamará “canvas” de ahora en adelante.

El objetivo es que el usuario arrastre elementos hacia el canvas de la misma forma en la que se hace en Xcode o Visual Studio. El sistema debe proveerle retroalimentación visual mostrando el objeto que está arrastrando y además mostrar en qué lugar quedararía el elemento una vez que el usuario lo suelte.

Para implementar este concepto de arrastrar y soltar, se utilizará jQuery UI. esta librería provee, entre otras cosas, métodos para habilitar el arrastrado de elementos en una página. El usuario arrastrará un elemento, y, mediante las llamadas de jQuery, se colocará el elemento en donde el usuario tenga su cursor en el momento, a manera de proveer retroalimentación visual. En cuanto el usuario suelte el elemento, se agregará su correspondiente fragmento de HTML en el template, lo que se reflejará en el canvas en tiempo real.

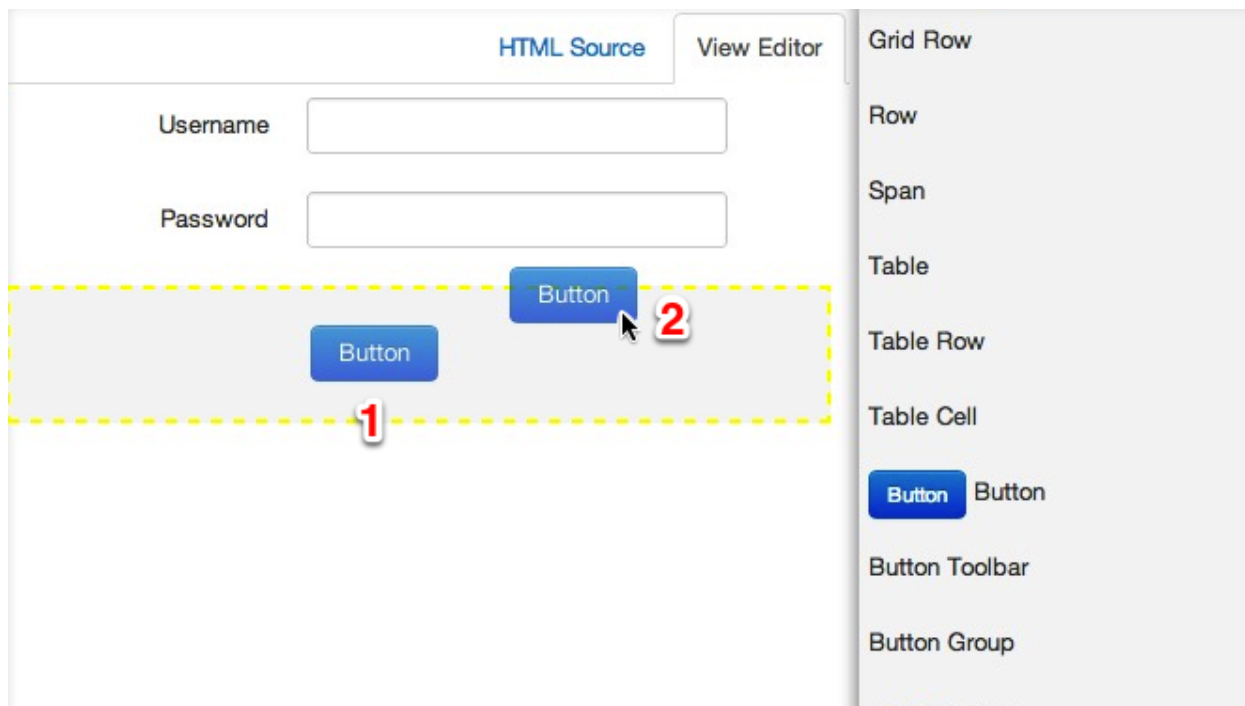


Figura 5: Los diferentes elementos de retroalimentación visual que se presentan al arrastrar un componente.

En la Figura 5 se pueden apreciar los diferentes elementos de retroalimentación al momento de arrastrar un elemento. En el punto 1 se pueden apreciar, primero, un borde amarillo al

rededor del elemento en donde “caería” el componente. Además, en el mismo número, puede visualizarse cómo se vería el componente (en este caso un botón) una vez que el usuario lo deje ahí. En el número 2, puede verse el mismo componente, que sigue al cursor mientras el usuario esté arrastrando. Esto le muestra al usuario qué es lo que está arrastrando.

De esta forma, el desarrollador puede ver, por un lado, qué componente estaría agregando al canvas y, por otro lado, cómo quedaría éste una vez que lo agregue.

4.2. Primera Etapa de Construcción

4.2.1. Creación del Entorno de Trabajo

En esta sección se detallarán cómo se crearon y estructuraron los dos entornos de trabajo, tanto para el backend como para el frontend.

Ambos entornos de trabajo se crearon en carpetas independientes (dada su naturaleza) y se inicializaron repositorios Git en cada una, a manera de mantener un control de versiones en cada una.

Para el control de versiones no se utilizó ninguna estrategia en especial. Dado que sólo el autor estará desarrollando, no valdría la pena implementar alguna estrategia de ramas o algo parecido para el control de versiones. Simplemente se utilizó la rama principal (**master** en Git), y se fueron creando “commits” cuando se considerara necesario.

4.2.1.1. Entorno de Trabajo para el Backend

El backend utilizará Ruby con Sinatra. Sinatra se diferencia de, por ejemplo, Rails, en que es un framework mucho más simple. Por esto, es que no posee utilidades para crear directorios de trabajo. La idea detrás de Sinatra es crear todo en un sólo archivo. Si bien esto es posible, e incluso aconsejable para algunas aplicaciones, no lo es para ésta, en donde se tendrán diferentes modelos y controladores. Por lo tanto se definió la siguiente estructura para el backend:

- **api**
 - **v1**
 - **config**: contiene diferentes archivos de configuración
 - **controllers**: los diferentes controladores
 - **models**: los modelos de usuario y proyecto
 - **app.rb**: este archivo es la base de la aplicación Sinatra, pues inicializa ciertas configuraciones y contiene métodos compartidos por los controladores

- `boot.rb`: este archivo es cargado inicialmente y se encarga de incluir las diferentes librerías y archivos para inicializar el servidor
- `projects`: será el contenedor de los diferentes proyectos que crearán los usuarios
- `public`: esta carpeta sirve archivos estáticos directamente, como imágenes
- `Gemfile`: archivo utilizado por Bundler (una librería de Ruby) para definir qué librerías y en qué versiones utilizará el backend
- `config.ru`: archivo utilizado para levantar el servidor

Se decidió estructurar el backend en carpetas de versiones. La idea detrás de esto es poder dividir cada versión de la API de manera de no perder compatibilidad con posibles clientes que estén basados en una versión de la API. Por ejemplo, de llegar a crearse un cliente para tablets, cada cliente estaría atado a una versión específica. Si se cambiara algún método (o se descontinuara), el cliente automáticamente fallaría. En cambio, teniendo diferentes versiones, se puede mantener esta compatibilidad.

Se creó además una carpeta llamada `projects`. Esta carpeta (que no es accesible directamente), guarda cada uno de los proyectos de cada usuario. Cada vez que se inicialice uno nuevo, se creará una subcarpeta en este directorio con la estructura determinada.

El archivo `Gemfile` es un archivo que utiliza la librería Bundler¹⁰. Esta utilidad permite especificar librerías externas que se quieren incluir en un proyecto (en este caso el backend) y especificar sus versiones. Por ejemplo, un extracto de un archivo `Gemfile` podría verse así:

```
gem 'sinatra', '1.3.3' # Especifica que se utilizará la versión 1.3.3

gem 'activerecord', '~> 3.2' # Especifica que se utilizarán
                             # versiones 3.2.X

gem 'haml' # Especifica que se utilizará la última versión
```

La gran utilidad de esta herramienta es que, al momento de ejecutar en la consola `bundle install`, las versiones especificadas son descargadas y se crea un archivo llamado `Gemfile.lock` que guarda las versiones que están siendo utilizadas. De esta forma, cuando otro desarrollador descargue el repositorio y ejecute nuevamente `bundle install` para instalar las dependencias, se descargarán exactamente esas versiones y ambos desarrolladores tendrán el mismo entorno de desarrollo.

Finalmente, el archivo `config.ru` especifica cómo debe levantarse el servidor. Este archivo detalla diferentes rutas que deben ser “montadas” y a las cuales el servidor debe responder

¹⁰Citar blablabla.

de diferentes maneras. En este caso, se tendrán dos rutas (inicialmente). Una, en la que se montará el backend mismo, o sea, `/api/v1`, y la otra, en la que se montará un servidor estático que sirva los archivos en la carpeta `public`.

4.2.1.2. Entorno de Trabajo para el Frontend

De la misma forma en que Switch utilizará Brunch para crear y administrar proyectos, se decidió utilizar la misma solución para construir la IDE. Brunch utiliza un sistema de esqueletos (“skeletons”, en inglés), los cuales utiliza para crear la estructura de los proyectos. Existe una gran variedad, utilizando diferentes lenguajes y frameworks. Se encontró uno que utiliza Backbone y CoffeeScript y se utilizó para crear la estructura de archivos inicial.

De no existir esta estructura, habría que escribir todo dentro de un sólo archivo, lo que para aplicaciones muy pequeñas puede ser práctico, pero no lo es en este caso. La estructura consiste en la siguiente:

- **app**
 - **assets**: imágenes y el archivo HTML principal de la aplicación
 - **models**: los modelos y colecciones
 - **routers**: definen los diferentes estados de la aplicación e inicializan lo necesario para funcionar en cada uno
 - **styles**: archivos de estilo (CSS) modularizados para cada sección de la aplicación
 - **views**: las vistas (o controladores)
 - **templates**: los archivos con HTML de cada vista
- **vendor**: librerías (Javascript o CSS) externas, como jQuery, Bootstrap, etc.

Existen otras carpetas que acá no se mencionan dado que no son relevantes a la aplicación. La funcionalidad de cada uno de estos tipos de archivos se explicaron en la Sección 4.1.2.1.

Para crear el entorno de trabajo, simplemente se ejecuta el siguiente comando:

```
brunch new -s git://github.com/meleyal/brunch-crumbs.git
```

Este comando utiliza el esqueleto presente en github.com/meleyal/brunch-crumbs para crear un directorio con la estructura ya mencionada.

4.2.2. Prototipado de la Interfaz

Se comenzó por prototipar la interfaz principal. Como ya se ha dicho, se utilizó Twitter Bootstrap, lo que permitió simplificar considerablemente esta etapa. Para realizar el prototipado se requirió realizar un poco de programación, pues hubo que crear vistas y rutas para ir testeando los casos de uso definidos anteriormente. La programación fue mínima de todas formas, enfocando esta etapa en prototipado y no en funcionalidad.

Se prototipó un menú superior con diferentes opciones de manera similar a los menú que se ven en diferentes IDE y programas de escritorio. Se incluyeron opciones como crear un nuevo proyecto, un nuevo archivo, ensamblar y ejecutar el proyecto, etc.

Se agregó la barra lateral izquierda, en la que se muestra una lista de los archivos abiertos y una lista de los archivos en el proyecto. Las carpetas cuentan con un ícono que las muestra como tal, mientras que archivos de vistas tienen un ícono que las diferencia de las demás. **PONER ACA QUE EN LA FIGURA X SE VE BLABLABLA**

Para el editor de código central se utilizó CodeMirror¹¹. Esto permitió embeber un editor de código muy extensible y completo en muy poco tiempo. El editor utiliza gran parte de la pantalla pues, siendo lo más esencial, debe dársele más espacio. **SUENA MUY MAL ESTO :(**

En el caso del editor de vistas, se agregó un “canvas” (básicamente un espacio en el cual arrastrar los componentes que se mencionarán más adelante), y una barra lateral derecha, que sólo es visible al estar editando un archivo que la requiriera. Además del canvas, en la parte superior se agregaron dos “tabs”, que permitirán al usuario cambiar entre el canvas y un editor de código para la vista. En la barra lateral estarán los diferentes componentes en una lista que mostrará una pequeña vista previa del componente y su nombre.

Otras vistas corresponden a el selector de proyectos, que se creó usando una ventana modal (que Twitter Bootstrap trae consigo). Ésta se mostraría en el momento que el usuario ingrese al programa. Ahí, podrá elegir algún proyecto en el que haya estado trabajando o crear uno nuevo directamente.

4.2.3. Creación de Servicios en el Backend

Para poder crear las funcionalidades necesarias en el frontend, se decidió crear primero los servicios en el Backend. Éstos son los siguientes:

- Autenticación de Usuarios
- Creación de proyectos
- Obtención de una lista de proyectos

¹¹Citar esto.

- Obtención de lista de archivos
- Operaciones CRUD en cada archivo y carpeta (incluyendo actualizar el contenido de archivos)
- Ensamblar y correr el proyecto para pruebas

Se consideraron dos acercamientos para la creación del backend. Primero, crear un modelo para los proyectos y modelos para los archivos (que podrían o no almacenarse en una base de datos). Segundo, crear un único modelo para el proyecto y dejar que éste maneje cada archivo utilizando su ruta en disco.

La primera alternativa se descartó dado que agregaba un cierto grado de complejidad sin agregar ningún beneficio. La segunda alternativa es más simple, pues considerando que los archivos y carpetas son literalmente archivos y carpetas en disco, no es necesario agregar una capa de abstracción dado que no simplificaría su manejo. Es por esto que cada proyecto tiene un modelo (y su información sí se guarda en la base de datos), y es posible manipular archivos y carpetas en el proyecto utilizando métodos en cada instancia (junto con la ruta al archivo o directorio).

Ahora, si bien sólo existirá un modelo para el proyecto, en términos de la interfaz que proveerá el backend, si se reflejará una diferencia entre archivos y proyectos. Por ejemplo, las siguientes son algunas de las rutas para realizar diferentes acciones:

- `GET /projects`: entregará una lista de proyectos existentes
- `GET /projects/:id/files`: entregará una lista de archivos en la raíz del proyecto
- `GET /projects/:id/files?path=/ruta/al/directorio`: lista los archivos en el directorio especificado

Aún cuando proyectos y archivos se manejarán en el mismo modelo, se dividirán en dos controladores, a manera de encapsular en cierta medida su funcionalidad.

En las subsecciones siguientes se discutirán algunas de las implementaciones de funcionalidades en el backend. Sin embargo no se discutirán los métodos que se publican y son accesibles a través de la API. Éstos últimos simplemente arrojan respuestas con objetos JSON como el que sigue para comunicarse con el frontend, por lo que no se considera necesario entrar en mucho detalle.

```
[
  {
    "name": "app",
    "parent": "",
```

```

        "type": "directory"
    },
    {
        "name": "Cakefile",
        "parent": "",
        "type": "file"
    },
    {
        "name": "conf",
        "parent": "",
        "type": "directory"
    },
    {
        "name": "config.coffee",
        "parent": "",
        "type": "file"
    }
    // etc...
]

```

4.2.3.1. Autenticación de Usuarios

Para autenticar usuarios se utilizó el sistema OAuth de GitHub. OAuth es un protocolo de autenticación y autorización que utiliza un proveedor. El proveedor en este caso es GitHub. La ventaja de este protocolo es que permite registrar y autenticar usuarios sin manejar un sistema de usuarios interno, quitando la necesidad de almacenar contraseñas y requerir al usuario registrarse en el sitio.

Funciona de la siguiente manera:

1. El consumidor (o sea, el backend en este caso) pide un token inicial al proveedor (en este caso GitHub).
2. GitHub genera este token único y lo devuelve.
3. Con este token, se genera una URL a la que el usuario es redireccionado.
4. Esta URL pertenece al proveedor, y es acá donde el usuario se autentifica contra el proveedor (no contra el consumidor). Al usuario se le da la opción de autorizar o denegar el acceso.
5. Si el usuario autoriza el acceso, es redireccionado al consumidor.
6. El consumidor ahora puede hacer una nueva petición al proveedor utilizando el mismo token inicial.

7. El proveedor, sabiendo que el usuario autorizó el acceso, entrega un *token de acceso*.

Este token de acceso permite al consumidor acceder a toda la información que el cliente haya autorizado. En el caso de este trabajo, sólo se necesitan el nombre y el correo electrónico. En la Figura 6 (en inglés) se puede apreciar el protocolo de mejor manera.¹²

Como puede apreciarse, implementar el protocolo OAuth manualmente resulta algo tedioso. Son bastantes los casos que deben considerarse y atenerse al protocolo podría tomar tiempo. Afortunadamente, existen diferentes librerías que permiten implementar este sistema de autenticación en muy poco tiempo. La librería Omniauth¹³ provee mecanismos de autenticación mediante OAuth y OpenID para diferentes proveedores (desde GitHub hasta Google). Junto con la librería omniauth-github, es posible implementar autenticación en muy poco tiempo.

El primer paso es crear una “aplicación” en el portal de desarrolladores de GitHub. Con esto, el proveedor entrega dos identificadores que permiten inicializar comunicaciones con ellos y autenticar a usuarios. En la Figura 7 se puede ver el proceso de creación de aplicaciones, y en la Figura 8 pueden verse ambos identificadores. Es simplemente entregar un nombre, una URL principal (que permite a GitHub asegurarse que las peticiones vienen del servidor que corresponde) y una URL a la cual redirigir al usuario (que puede sobreescribirse en cada petición).

4.2.3.2. Creación de Proyectos

La creación de un proyecto nuevo es relativamente simple. Consiste en crear un nuevo proyecto en la base de datos con el nombre que provee el usuario, y luego crear el proyecto mismo utilizando Brunch.

Al crear un proyecto nuevo, el modelo guarda en la base de datos una nueva entrada que está asociada al usuario que hace la llamada a la API. Guarda un nombre para el proyecto, la ruta en la que fue guardado el esqueleto y un puerto único. Antes de guardar la entrada en la base de datos, un “callback” es gatillado en el mismo modelo (en Ruby) que ejecuta el comando que crea la carpeta para el proyecto, como se detalla a continuación:

```
def create_project
  # Create a random path and the project
  self.path = "#{self.name}-#{SecureRandom.hex(10)}"
  system "brunch new #{self.full_path} \
    -s git://github.com/ianmurrays/brunch-crumbs.git"
end
```

¹²Poner referencia! <http://oauth.googlecode.com/>.

¹³Url!

OAuth Authentication Flow

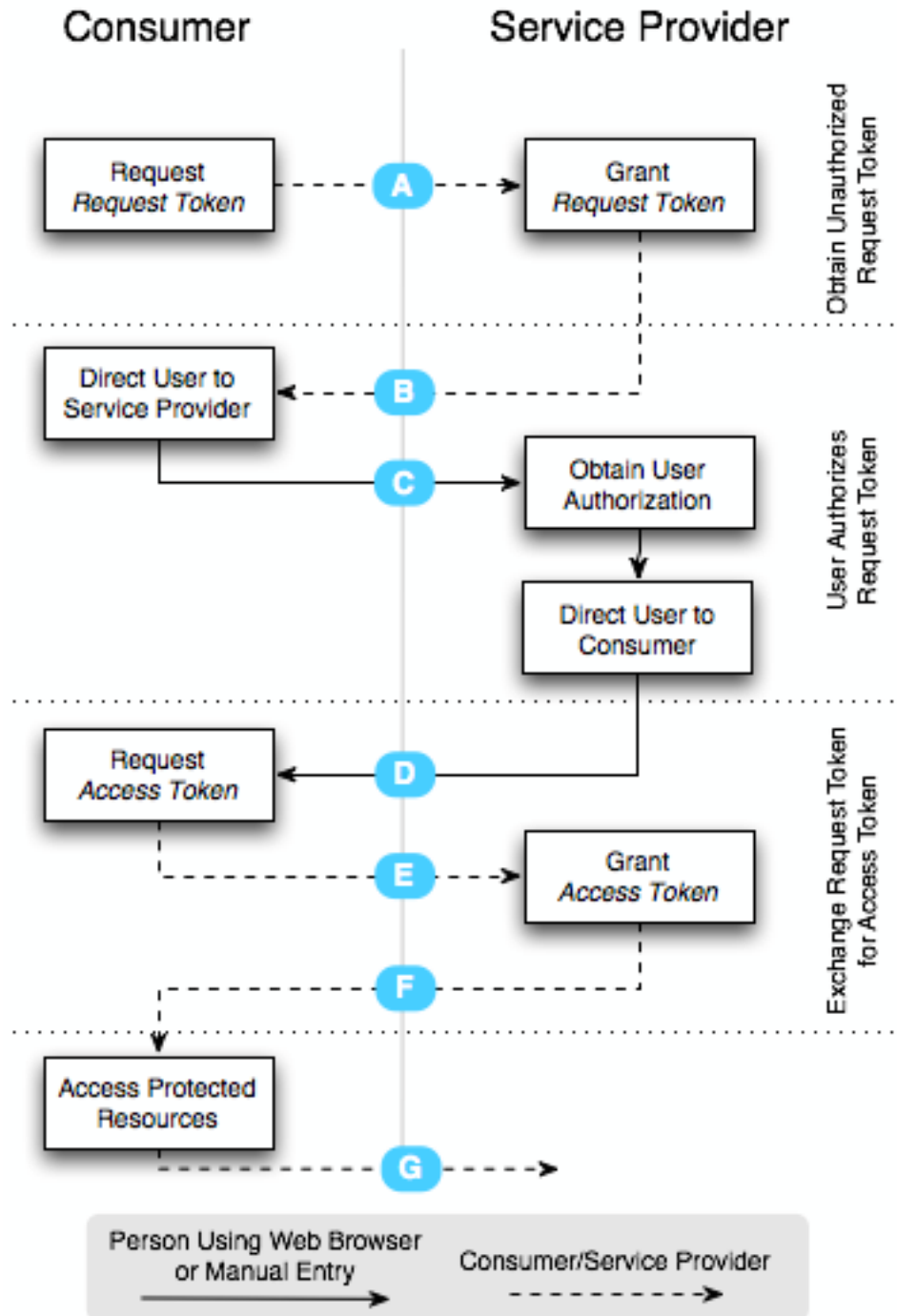


Figura 6: Fragmento del diagrama de flujo de autenticación de OAuth

Applications / **Register a new OAuth application**

Application Name


Main URL

Callback URL

Register application

Figura 7: Proceso de creado de aplicaciones en GitHub

Applications / **Switch IDE – Development**

 **ianmurrays** owns this application. [Transfer ownership.](#)

1 user

Client ID

fb319448e455

Client Secret

8413267392254f8d2abc44c72099bbde

Reset client secret

Name

Switch IDE – Development

URL



http://localhost:9292

Callback URL

http://localhost:9292/api/v1/auth/github/callback

Update application

Delete application

**OAUTH**
Authenticating with the GitHub API


[OAuth Documentation](#)

Figura 8: Acá pueden apreciarse ambos identificadores que entrega GitHub

Se crea un sufijo aleatorio para evitar colisiones con proyectos que tengan el mismo nombre, y se llama a un comando de sistema para crear el esqueleto. Esta implementación no es perfecta, dado que podría darse el caso de que `SecureRandom.hex(10)` arroje una cadena aleatoria idéntica a alguna anterior. Para efectos del desarrollo del presente trabajo, se decidió no darle demasiada importancia a este defecto, dado que las probabilidades de que ello ocurra son demasiado bajas.

Después de que finalice este comando, se llama a un segundo “callback” que crea el número de puerto único para esta aplicación:

```
def randomize_port
  begin
    self.port = 8000 + rand(2000)
  end until Project.where(port: self.port).count == 0
end
```

Básicamente asigna un puerto entre 8000 y 10000 aleatorio (y único) a cada proyecto. De esta forma, si se están editando dos proyectos simultáneamente, pueden levantarse dos instancias para hacer pruebas sin colisionar. Esta implementación, si bien es suficiente para el desarrollo de la herramienta y para este trabajo, no sería una implementación ideal en producción, dado que alguno de esos puertos podría estar siendo ocupado por algún servicio en el sistema. En una futura implementación podría agregarse algún mecanismo que verifique la disponibilidad del puerto al momento de asignarlo, o bien, verifique y asigne puertos cada vez que se ensamble y corra el proyecto.

4.2.3.3. Manipulación de Archivos

La manipulación de archivos se hace directamente sobre ellos usando librerías estándar de Ruby. Sólo se discutirán algunas de las implementaciones presentes en el backend.

Para listar los archivos presentes en una determinada ruta, se utilizó la siguiente implementación:

```
def files_in(folder = "")
  # Remove leading and trailing slashes
  folder.gsub! /\^\/, ""
  folder.gsub! /\$/ , ""

  files = Dir["#{self.full_path}/#{folder}/*"].collect do |entry|
    next if %w{server.js node_modules}.include? File.basename(entry)
    self.file_to_hash entry, folder
  end
end
```

El método lista todos los archivos presentes en la ruta especificada. De no especificarse, lista los archivos en la raíz del proyecto. Primero se eliminan las barras (/) al principio y final de la ruta que se especifique, y luego se recorre el directorio usando la clase `Dir`¹⁴. La llamada a `Dir[/ruta/a/directorio"]` retorna un array que se recorre utilizando `collect`. El método `collect` en los arreglos genera un nuevo arreglo con los elementos que retorne el bloque que se le pase. `collect` pasa cada elemento del arreglo original al bloque como argumento para su manipulación. Por ejemplo, si el siguiente fragmento de código retorna un nuevo arreglo con sus elementos elevados al cuadrado:

```
[1,2,3,4,5].collect do |num|
  num * num
end

# => [1,4,9,16,25]
```

Por lo tanto, la variable `files` en la implementación de `files_in` contendrá la representación en un Hash¹⁵ (básicamente un diccionario) de cada archivo (o directorio). Además, se excluirán el directorio `node_modules` y el archivo `server.js` del listado, dado que son un directorio y un archivo que no deben ser manipulados por el desarrollador.

La implementación del método `file_to_hash` es relativamente simple, y genera un diccionario con el nombre, padre y tipo de archivo para la ruta que se le pase como parámetro:

```
def file_to_hash(file, folder)
  {
    :name => File.basename(file),
    :parent => folder,
    :type => if File.directory?(file)
      :directory
    else
      :file
    end
  }
end
```

Para obtener y actualizar el contenido de archivos se utilizó una implementación relativamente simple:

```
def file_content(path)
  if FileTest.exists? self.full_path(path) \
```

¹⁴[Http://ruby-doc.org/core-1.9.3/Dir.html](http://ruby-doc.org/core-1.9.3/Dir.html).

¹⁵[Http://www.ruby-doc.org/core-1.9.3/Hash.html](http://www.ruby-doc.org/core-1.9.3/Hash.html).

```

    and ! File.directory? self.full_path(path)
  {
    :content => File.read(self.full_path(path))
  }
end
end

```

Básicamente, se verifica que el archivo indicado exista y que no sea un directorio, y se retorna un Hash indicando el contenido del archivo. Se decidió retornar un Hash pues facilita su lectura en el frontend, retornando un objeto JSON en vez del contenido directamente.

Para la escritura de archivos se utilizó una implementación similar:

```

def update_file(path, content)
  if FileTest.exists? self.full_path(path) \
    and ! File.directory? self.full_path(path)
    File.open(self.full_path(path), 'w') do |file|
      file.write content
    end
  end
end

```

Se realiza la misma verificación que al momento de obtener el contenido. Si el archivo existe y no es una carpeta, se reemplaza todo el contenido directamente.

4.2.3.4. Ensamblado y Servidor de Pruebas

4.2.4. Agregado de Funcionalidad al Prototipo del Frontend

Se comenzó por implementar el selector de proyectos. Para esto fue necesario implementar el modelo y colección de proyectos. En esta etapa fueron implementados de la forma más simple posible. Básicamente, se crearon como dos clases que extienden a `Backbone.Model` y `Backbone.Collection` respectivamente. Dado que Backbone está diseñado para interactuar con APIs REST, no fue necesario configurar más que la URL del backend y especificar que la colección corresponde a una colección de Proyectos.

PONER EJEMPLOS DE CÓDIGO ACÁ

Luego de configurar el modelo y la colección, se agregó una ruta al enrutador de Backbone. El enrutador lee la URL del navegador e interpreta qué método llamar del enrutador. La idea es que se especifiquen las URL necesarias para la navegación de la aplicación. En el caso de esta solución, sólo existirán dos rutas principalmente. La primera será la ruta base, donde se cargará el selector de proyectos del cual se habla, y la segunda será la que tendrá

cada proyecto. Por lo tanto, se configuró la URL base, y, en el método que es llamado, se inicializa la colección de proyectos.

En este punto, la colección es pasada a una vista. Las vistas, como se explicó antes, son las encargadas de presentar datos al usuario (por medio de templates). Esta vista, básicamente, presenta una lista de proyectos y un formulario para crear uno nuevo (esta última funcionalidad se creará en la segunda etapa).

AGREGAR SCREENSHOT DE CREAR UN PROYECTO

Al usuario hacer click en un proyecto existente, se llama a la segunda ruta ya mencionada. Esta ruta, toma el identificador de proyecto de la url (que tienen un formato estilo `projects/IDENTIFICADOR`) e instancia un proyecto usando ese identificador. Una vez que se haya obtenido toda su información desde el servidor, se le asigna el modelo al proyecto (de manera que su instancia quede compartida) y se inicializa el visor de archivos (que es una vista) con éste.

En este paso, se instancia el visor de archivos. El visor de archivos toma el modelo de proyecto y “pide” su carpeta raíz. El modelo hace una petición al backend, y éste contesta con una representación de cada archivo (o carpeta) del directorio raíz. A continuación un ejemplo de la respuesta del servidor:

JSON ACA OH SI

El visor de archivos, instancia una vista de archivo por cada uno de los documentos que responde el backend. La vista de archivo se encarga de mostrar el nombre y tipo de archivo, y permitir al usuario clickearlo para abrirlo o ver su contenido. En caso de que el archivo se trate de un directorio, la vista de archivo se encarga de mostrar sus contenidos. Esta parte resultó ser un poco complicada de implementar, dado que lo que se necesita es básicamente mostrar el mismo tipo de vista que la raíz del proyecto pero para un subproyecto. Lo que se hizo en este caso es lo siguiente: cuando el usuario hace click en un directorio, se instancia una colección de archivos con la ruta a ese directorio. Se hace una petición al servidor para que entregue la lista de archivos en esa carpeta y se instancian vistas de archivo para cada uno, embebiéndolas en la misma vista del directorio que se acaba de clickear. En la Figura 9 se puede apreciar el concepto. La carpeta `app` es una vista de archivo, y contiene todos los archivos y carpetas dentro de su recuadro. Lo mismo pasa con el directorio `models`.

En caso de que el usuario haga click en un archivo, se le pasa la instancia del modelo al editor de código, el cual indica al modelo que debe pedir el contenido del archivo al servidor para mostrarlo. El editor de código es simplemente una librería llamada CodeMirror que permite al usuario editar el contenido de los archivos. El editor de código se encarga además de indicarle al modelo del archivo que guarde su contenido en el backend si el usuario lo solicita.

NO SE SI EXPLICAR MÁS DE LO ANTERIOR

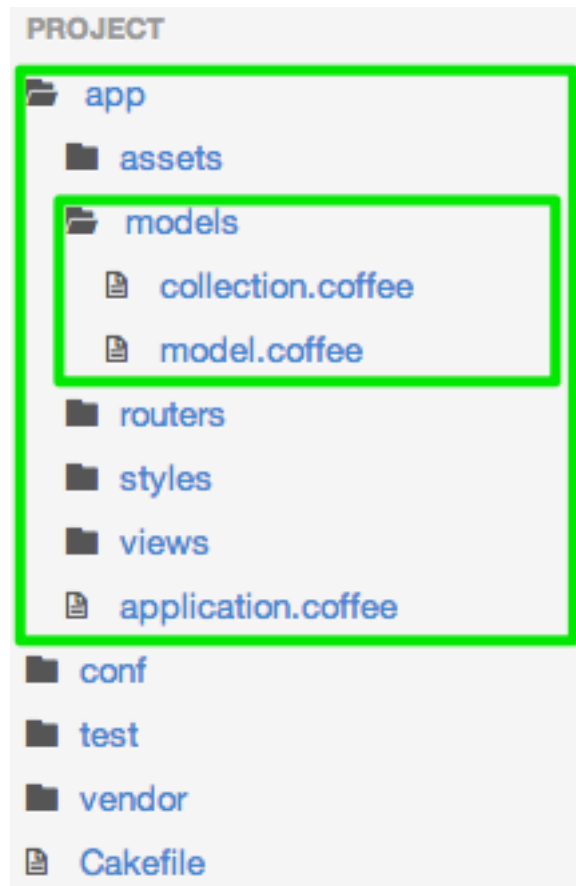


Figura 9: El visor de archivos: cada vista de directorio contiene más vistas de archivos de ser necesarias.

Para esta etapa de la construcción, se incluyó además la posibilidad de ensamblar y ejecutar el proyecto. Para esto, se agregaron métodos en el modelo de proyectos que hace llamadas al backend para ensamblar y ejecutar el servidor de pruebas. El evento es manejado por la barra de navegación, que incluye varios elementos de menú que por esta etapa se mantuvieron inactivos. A la derecha de la barra de navegación se encuentra un botón que permite ensamblar y ejecutar el proyecto con un sólo click, y otros dos que permiten ejecutarlo y ensamblarlo por separado.

4.3. Segunda Etapa de Construcción

La segunda etapa de construcción del proyecto se dedicó principalmente a la implementación del editor de templates. Dado que esta es la parte más importante del proyecto se dedicó una etapa completa a ella.

El editor se diseñó de manera que en el centro se tuviera una vista en vivo de lo que se estaba construyendo, mientras que a la derecha se listaran todos los componentes disponibles para agregar al template. Dado que los templates son básicamente HTML, es el navegador el que se encarga de mostrar cómo se vería finalmente. Por esto, lo que se hizo fue agregar elementos a la lista de componentes de manera que al arrastrarlos hacia el centro (el editor), simplemente se agregue su representación en HTML y el navegador se encargaría de mostrar su “vista previa”.

Entonces, en la lista de componentes se decidió agregar botones, tablas, formularios, campos de texto, entre otros, y dentro de ellos (en código, no visible para el usuario) agregar un fragmento de HTML que se agregaría al template. Entonces, utilizando jQuery UI¹⁶, cada componente se convierte en un elemento arrastrable. Con jQuery UI, se necesita convertir elementos en “arrastrables” y además, crear elementos en donde “soltar” lo que el usuario está arrastrando. En este sentido, y, en un primer intento, se convierten todos los elementos en la vista previa en “soltables”.

```
# Con la siguiente llamada, se convierte cada elemento en el editor  
# de vistas en "soltable".  
@$("#*").droppable()
```

Con este primer acercamiento, ya se podía arrastrar y soltar componentes. El problema es que se podían arrastrar componentes como botones y otras cosas dentro de elementos HTML que no correspondía, como imágenes, menús, etc. Para esto, se incluyeron ciertas excepciones a la llamada anterior, como sigue:

```
# Seleccionar todos los elementos, excepto los que están en la  
# llamada .not()  
@$("#*").not('img, button, input, select, option, optgroup').droppable()
```

¹⁶Definir esto!

Con esto, se simplificó un tanto el arrastrado de componentes, evitando que algunos quedaran dentro de elementos que no correspondía. Ahora, se notó que era difícil saber dónde realmente se estaba dejando el componente que el usuario estaba arrastrando, por lo que se incluyó retroalimentación visual al momento de arrastrar, es decir, cuando el usuario esté arrastrando el elemento, el elemento en donde “caería” el componente se rodea con un borde amarillo, como muestra la Figura ??.

PONER FIGURA! Esto se logra usando propiedades de jQuery UI:

```
exceptions = 'img, button, input, select, option, optgroup'
@$("#*").not(exceptions).droppable
    hoverClass: "hovering" # Esto agrega una clase CSS con un borde.
```

La propiedad `hoverClass` agrega una clase CSS al elemento donde se estaría arrastrando el componente y la remueve al salir. Con esto se agrega un borde que facilite al usuario saber dónde caerá el componente.

Se decidió además agregar componentes que sólo sirven si se arrastran dentro de un formulario. En este punto, el usuario puede arrastrar estos componentes a cualquier parte, lo que hace de su uso algo complicado. Para solucionar esto, se implementó un sistema en el cual cada componente tiene especificado dónde puede ser arrastrado. Por ejemplo, los botones pueden ser arrastrados a cualquier parte:

```
<div class="switch-component" data-component-type="button">
  <div class="payload">
    <button type="button" class="btn btn-primary">Button</button>
  </div>

  <span class="name">Button</span>
</div>
```

En cambio, los elementos de un formulario sólo pueden arrastrarse a un formulario previamente colocado. En el siguiente fragmento se puede notar la propiedad `data-component-drop-only` que contiene una cadena de texto con selectores CSS en dónde puede ser agregado.

```
<div class="switch-component" data-component-type="label-button"
  data-component-drop-only="form">
  <div class="payload">
    <div class="control-group">
      <div class="control-label"><label for="new_input">Label</label></div>
      <div class="controls">
        <input type="text" name="new_input" id="new_input">
      </div>
    </div>
  </div>
```



```

    </div>
  </div>

  <span class="name">Form Input</span>
</div>

```

Lo anterior, junto con el siguiente Coffeescript, permite que los componentes puedan ser arrastrados sólo a ciertos elementos, si es que lo especifican:

```

makeDroppable: (only) ->
  # Si se especificó only, entonces usarlo, de lo contrario
  # permitir cualquier elemento.
  if only
    only = "#view_container #{only}"
  else
    only = "#view_container, #view_container *"

  @$(only).not(exceptions).droppable
    hoverClass: "hovering"

```

Hasta ahora, el editor de templates agrega los componentes anexándolos al final de la posición en la que el usuario las deja. Esto lo imposibilita de agregar componentes al principio de una lista por ejemplo. **FIGURA EXPLICANDO ESTO?** Por lo tanto, se agrego la posibilidad de cambiar ese comportamiento. Al momento de arrastrar un componente, el usuario puede presionar (y mantener presionada) la tecla **SHIFT**, de manera que al dejar un componente, éste se anexe al principio en vez de al final, permitiendo al usuario agregar cosas al principio de listas o formularios, por ejemplo.

Por último, para facilitar aún más el arrastrado de componentes, se agregó una vista previa de cómo quedaría el componente que se está arrastrando una vez que se suelte. La idea es que al estar arrastrando un elemento, éste aparece en el editor con una ligera opacidad. Esto se logró usando algunas llamadas de jQuery UI:

```

# ...
@$(only).not(exceptions).droppable
  hoverClass: "hovering"
  greedy: yes
  drop: (e, u) ->
    self.putComponent(self, $(this), u, no)
  over: (e, u) ->
    self.putComponent(self, $(this), u, yes)
  out: (e, u) ->
    self.removeComponent()

```

Con estas llamadas, al momento de que el usuario esté sobre un elemento (“over”), literalmente se agrega el componente al editor, para luego eliminarlo en caso de salirse (“out”), o bien dejarlo definitivamente al soltarlo (“drop”). **FOTOOO!!!**

Por último, el editor de templates también debería permitir al usuario editar el código directamente, en caso de que no exista algún componente o bien se necesite agregar cierta lógica más allá de HTML. Para esto, se utilizó el mismo editor de código que para los archivos normales. Se agregaron dos pestañas en la parte superior del editor de templates que permiten cambiar entre el editor visual y el código fuente (ver Figura 10).

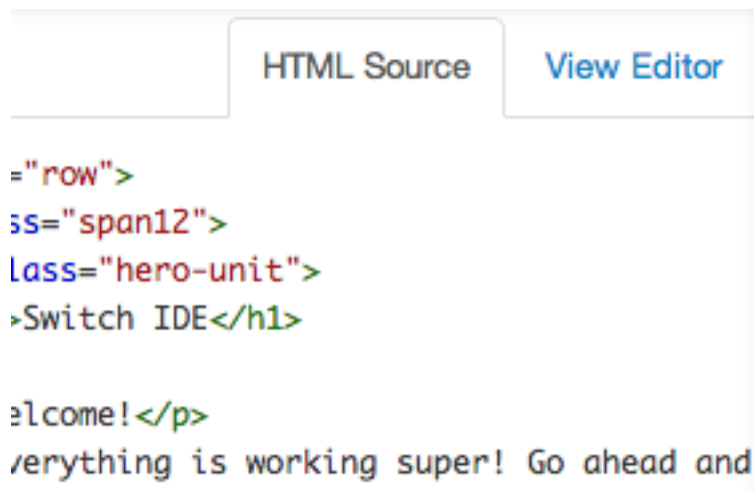


Figura 10: Estas pestañas permiten al usuario cambiar entre el modo visual y el editor HTML

5. Resultados

6. Conclusiones

A continuación se presentarán conclusiones del presente trabajo, en respecto a la solución lograda, su efectividad, la metodología empleada, características que no se pudieron agregar (y las respectivas razones), y conclusiones para un trabajo futuro.

6.1. Sobre la Solución

La solución que se presenta en este documento permite disminuir los tiempos de desarrollo de soluciones web, facilitando la tarea que más tiempo consume. El prototipado y construcción de vistas en aplicaciones de cualquier tipo siempre es la más tediosa, sobre todo en el ambiente web donde todo debe hacerse por código. Usando Switch IDE, se facilita gran parte de esta tarea proveyendo al desarrollador con componentes utilizados comúnmente en el desarrollo de aplicaciones, como botones, formularios, tablas, entre otras.

Sin embargo, posee algunas desventajas. Cada vez que se quiere probar algún cambio realizado en el programa, hay que reensamblar el proyecto, lo que toma tiempo. Esto, en todo caso, puede solucionarse y se discute en la Sección 6.5 del presente capítulo. Otra desventaja que presenta la solución es que agregar lógica a los templates (como se hace normalmente en desarrollo web), es difícil. Por ejemplo, si se desea mostrar un botón sólo en caso de que alguna variable sea verdadera en el controlador, se debe agregar código a la vista y eso hace que el editor visual muestre este código. Soluciones a este problema se discuten también en la Sección 6.5.

6.2. Sobre la Metodología

El haber dividido el proceso de construcción en dos fases permitió evaluar la efectividad de las herramientas escogidas antes de comenzar con la programación del aspecto más importante de la solución. La primera fase involucró la creación del backend de manera casi completa, el prototipado de la interfaz y la creación de una base para el desarrollo de la segunda etapa.

La división en dos fases fue acertada, y el orden del desarrollo de los diferentes componentes fue correcto, dado que de haber prototipado el editor de interfaces antes de comenzar con la base, se podría haber creado (accidentalmente) un editor completamente incompatible con la solución final. Ahora bien, el haber prototipado un editor de interfaces antes de comenzar con el desarrollo podría haber revelado dificultades que se podrían haber encontrado en la segunda etapa (aunque de todas formas se sabía que era posible desarrollar algo de esa índole dado que existían herramientas similares).

6.3. Sobre la Elección de Herramientas

6.3.1. Herramientas del Backend

Haber elegido Ruby y Sinatra para construir el backend fue una buena decisión. Permitió crear un backend simple, liviano y mantenible en poco tiempo.

YA Y?

6.3.2. Herramientas del Frontend

En un principio se escogió Backbone para construir el frontend por su simplicidad, popularidad y por ser un framework liviano. Hubo que considerar que el autor no poseía conocimientos con ninguno de estos frameworks inicialmente, por lo que la elección se vio sesgada hacia una herramienta fácil de aprender y con buena documentación y soporte (o sea, una comunidad activa).

Backbone probó ser un framework fácil de dominar y muy flexible. La presencia de varias librerías y extensa documentación en línea permitió dominar la herramienta en poco tiempo y crear una solución mantenible y legible.

6.4. Características que Faltaron

NO SÉ SI AGREGAR ESTO EN LA MEMORIA

La principal característica que se omitió fue la de poder “unir” elementos con acciones en el editor de vistas. Por ejemplo, al agregar un botón, la idea sería conectarlo con el controlador de manera que al hacer click se ejecutara una determinada acción. En Xcode es posible presionar CTRL y arrastrar el componente hacia el código, y automáticamente se genera una acción que se ejecutaría al presionar el componente. Ver Figura 11.

Otro feature que no se agregó es el de archivar los proyectos. Archivar significa básicamente ensamblar el proyecto y dejarlo listo para su implementación en un servidor. La idea de esta característica es ensamblar y comprimir el proyecto en un archivo zip para su posterior descarga.

6.5. Trabajo Futuro

Una de las características que sería ideal agregar a una futura versión es la de control de versiones con Git. Agregar esta característica no sería trivial pero tampoco sería extremadamente difícil dado que existen librerías que lo facilitarían. Agregar esta funcionalidad permitiría a los desarrolladores mantener su código versionado y además permitiría la colaboración, por medio de repositorios compartidos.

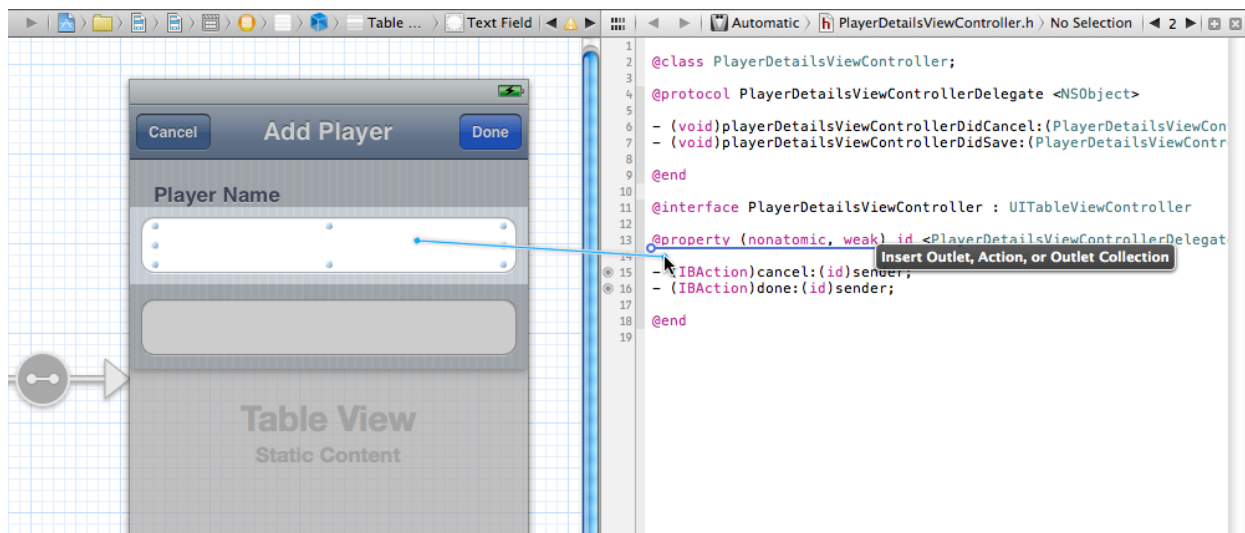


Figura 11: Al presionar la tecla CTRL y arrastrar, es posible crear métodos directamente.

Agregar esta funcionalidad requeriría de interfaces que permitan al desarrollador seleccionar archivos para agregar al repositorio, crear nuevos “commits” (como se le conoce a los estados de código en versionamiento), descargar y subir cambios a repositorios remotos y poder (al menos) ver el historial del repositorio.

Otra característica importante que se discutió antes es la de no necesitar ensamblar el proyecto constantemente. Brunch (el ensamblador que se utilizó en este trabajo) permite levantar un proceso que observa cambios en los archivos y ensambla el proyecto bajo demanda. Eso hace que el proceso de programar y probar los cambios sea mucho más continuo y simple. ***Tal vez hablar más de esto?***

Una adición que podría ser de importancia en proyectos con archivos grandes sería la de no enviar el archivo completo cada vez que se guarden cambios. Este es el comportamiento actual y, si bien no se nota para archivos livianos, sí se sentiría al momento de guardar archivos de 100 o más líneas de código. Una posible solución sería enviar parches, es decir, guardar en el frontend el estado en el que se encuentra el archivo en el servidor, y, al momento de guardar cambios, enviar sólo las diferencias de manera de minimizar la cantidad de información enviada. De esta forma, si se tuviera un archivo con 200 líneas de código y sólo se quisiera agregar 2 líneas con comentarios, se ahorraría un 99% de cantidad de datos que se transferían.

Un cambio que no sería menor, y que está relacionado con la edición de vistas, es la de cambiar el framework que se utiliza para el desarrollo por una conocida como Knockback¹⁷. Knockback es una combinación de dos frameworks: Backbone (la que se utiliza actualmente) y Knockout. Knockout es conocido por ofrecer “bindings”, es decir, permite agregar atributos

¹⁷Referencia.

HTML a las vistas de manera que se actualicen automáticamente, sin necesidad de escribir código dentro de ellas. Como por ejemplo:

```
<p>First name: <strong data-bind="text: firstName"></strong></p>
<p>Last name: <strong data-bind="text: lastName"></strong></p>
```

El fragmento anterior une las etiquetas `` con los atributos `firstName` y `lastName` del modelo asociado. Lo ventajoso es que es simple HTML y además las uniones son en tiempo real, lo que significa que cualquier cambio al modelo ocurrirá en la vista sin requerir ningún esfuerzo extra por parte del desarrollador.

Esto mejoraría la experiencia del usuario al momento de diseñar vistas dado que no necesitaría escribir código (sólo HTML). Además, y más importante, esto permitiría mejorar el editor de interfaces de manera de que el usuario agregue “bindings” visualmente, con menús contextuales, sin editar el código fuente de la vista.

Agregar este framework requeriría cierto esfuerzo, aunque Brunch facilita esta tarea permitiendo crear “esqueletos” para proyectos con este framework. Exportar proyectos ya existentes no sería trivial, pero tampoco sería muy complejo, dado que Knockback simplemente combina ambos frameworks.

7. Referencias

- [1] Steve Burbeck, “Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)” Disponible: <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>. Última Revisión: 09/07/2012.
- [2] Mike Potel, “MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java” Disponible: <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>. Última Revisión: 09/07/2012.
- [3] DocumentCloud, “Backbone” Disponible: <http://backbonejs.org>. Última Revisión: 09/07/2012.
- [4] Backbone, “Projects and Companies using Backbone” Disponible: <https://github.com/documentcloud/backbone-and-companies-using-backbone>. Última Revisión: 09/07/2012.
- [5] Cappuccino Project, “Cappuccino Framework” Disponible: <http://www.cappuccino-project.org/>. Última Revisión: 09/07/2012.
- [6] Apple Inc., “Xcode” Disponible: <https://developer.apple.com/xcode/>. Última Revisión: 09/07/2012.
- [7] Sencha Inc., “Ext JS” Disponible: <http://www.sencha.com/products/extjs/>. Última Revisión: 09/07/2012.
- [8] Sencha Inc., “Ext JS Releases” Disponible: <http://www.sencha.com/products/releases/>. Última Revisión: 09/07/2012.
- [9] Sencha Inc., “Buy Ext JS 4 Licenses and Support” Disponible: <http://www.sencha.com/store/extjs/>. Última Revisión: 09/07/2012.
- [10] Sencha Inc., “Sencha Architect” Disponible: <http://www.sencha.com/products/architect/>. Última Revisión: 09/07/2012.
- [11] Sencha Inc., “Buy Sencha Architect” Disponible: <http://www.sencha.com/store/architect/>. Última Revisión: 09/07/2012.
- [12] Divshot, “Divshot” Disponible: <http://www.divshot.com>. Última Revisión: 09/07/2012.
- [13] Twitter, “Twitter Bootstrap” Disponible: <http://getbootstrap.com>. Última Revisión: 09/07/2012.
- [14] Ian Hickson (editor), “HTML5 Specification” Disponible: <http://www.w3.org/TR/2011/WD-html5-20110525/>. Última Revisión: 09/07/2012.
- [15] eXo Platform SAS, “eXo Cloud IDE” Disponible: <http://cloud-ide.com>. Última Revisión: 09/07/2012.
- [16] Git Project, “Git” Disponible: <http://git-scm.com>. Última Revisión: 09/07/2012.
- [17] VMware Inc., “Wavemaker” Disponible: <http://wavemaker.com>. Última Revisión: 09/07/2012.

[18] Zoho Corp., “Zoho Creator” Disponible: <http://www.zoho.com/creator/>. Última Revisión: 09/07/2012.