Implementación de aplicaciones isomórficas con Javascript

Hernán Alejandro Quintana Cruz
DEVOS INC SAC. Lima, Perú.

Recibido: 14 de octubre de 2015 / Aprobado: 30 de octubre de 2015

Resumen

Con la evolución de los navegadores y la implementación de los nuevos estándares web (HTML5), aparecieron nuevas herramientas que nos han permitido crear aplicaciones web más complejas, con una mejor experiencia de usuario. Con el tiempo, este tipo de aplicaciones han resultado útiles, pero sacrificando la experiencia de usuario debido a los tiempos de carga de los datos en cada página. Fue por este inconveniente que se evidenció la necesidad de la interacción desde el servidor para la generación de página con aplicaciones híbridas. Una de estas técnicas era permitir la utilización (creación y modificación) de componentes de interfaz gráfica, tanto en el cliente como en el servidor, por lo cual estas aplicaciones son llamadas isomórficas. El presente artículo muestra el proceso que se llevó a cabo hasta llegar al concepto de aplicaciones isomórficas, describe las características que una aplicación isomórfica debe tener y, por último, plantea un modelo de arquitectura de aplicación con ejemplos de código.

Palabras clave: aplicaciones web / aplicaciones isomórficas / arquitectura web / interfaces gráficas / patrones de diseño web

Implementation Isomorphic JavaScript Applications

Summary

With the evolution of browsers and the implementation of new Web standards (HTML5), new tools became available that have enabled us to create more complex web applications with better user experiences. Over time, these types of applications have proven useful, but deteriorate the user experience as extended lengths of time are used load the data on each page. For this same reason, it is proven that there is a need for interaction from the server to the generation page with the hybrid applications. One such technique was to allow the use (creation and modification) of graphical user interface elements, both in the client just like the server, which is why these applications are called isomorphic. This article shows the process that was undertaken to arrive at the concept of isomorphic applications, which describes the characteristics that must have an isomorphic implementation and, finally, presents a model of application architecture with code examples.

Key words: web applications / isomorphic applications / web architecture / graphical interfaces / web design patterns

Introducción

En los inicios de la World Wide Web (WWW), la totalidad de las páginas web, en formato HTML, eran generadas por servidores web. Es decir, al ingresar una URL por un navegador nos conectábamos a un servidor web que generaba una página web en tiempo real**,** tomando los parámetros que se enviaban en la petición. Esta retornaba al navegador para ser renderizada y mostrada por el servidor al usuario.

Esta arquitectura resultó eficiente, ya que en ese momento los navegadores no eran muy poderosos, de manera tal que al trasladar el procesamiento de la página web al servidor, se dinamizaba el flujo de la información. Gracias a este enfoque se crearon diversos *frameworks* y librerías que optimizaban y facilitaban la generación e implementación de páginas web dinámicas en el servidor (*backend*). Tecnologías como Java Server Pages (JSP) y sus extensiones como Java Server Faces (JSF), Struts y Spring en el mundo JAVA fueron utilizadas por muchos desarrolladores para la creación de aplicaciones web empleadas en diversos ámbitos.

Posteriormente se requirieron páginas web mucho más interactivas, por lo que se comenzó a utilizar un lenguaje interpretado por el mismo navegador, llamado Javascript. Este lenguaje se usó para permitir que se efectúen cambios en un documento HTML directamente en el cliente (navegador), sin necesidad de realizar alguna petición al servidor. Fue así como nacieron las páginas web con arquitectura Single-Page App.

1. Arquitectura Single-Page App

Según Wikipedia, el Single-Page App o SPA "es un sitio web que cabe en una sola página". Una aplicación con arquitectura de este tipo, comúnmente necesita que todos sus recursos (página HTML, hojas de estilo CSS y código Javascript) se carguen una sola vez en una única petición a un servidor web. Cualquier cambio posterior en la página web (como nuevas pantallas) se construye en el mismo navegador utilizando Javascript.

El manejo y la gestión de los datos que van a ser exhibidos y utilizados por la aplicación web, se realizan a través de peticiones directas al *backend*. Para lograrlo, se define e implementa un Application Programming Interface (API) que servirá de protocolo de comunicación entre el cliente y el servidor. En el cliente (navegador) se aplica la técnica de comunicación asíncrona llamada AJAX y se realizan las peticiones al API utilizando el protocolo HHTP.

El servidor donde se implementa el API puede ser escrito en cualquier tecnología (Java, Python, Ruby, etc.). Brehm (2013) dice que uno de los beneficios más reconocidos de este tipo de arquitecturas es que mejora la experiencia del usuario,

a través de una navegación más fluida, sin necesidad de continuas actualizaciones para obtener datos. Incluso, si se programa correctamente, podría funcionar *offline*.

Otra de las ventajas que apunta Brehm (2013) es que facilita la implementación de una aplicación web desde el punto de vista de un desarrollador. Una aplicación que emplee esta arquitectura, separa completamente la interfaz del usuario (*frontend*) de la del servidor (*backend*), donde comúnmente va la lógica del negocio. De este modo, se pueden dividir los equipos y tareas sin ningún problema, ya que no se comparte lógica y no hay problemas de acoplamiento.

A medida que se popularizó esta arquitectura para diversas aplicaciones web, comenzaron a surgir inconvenientes. Brehm (2013) cita algunos de los problemas que suelen suceder en aplicaciones de este tipo:

a) **Dificultades con el Search Engine Optimization (SEO)** El optimizador de motores de búsqueda (SEO, por sus siglas en inglés), es la técnica para mejorar el posicionamiento de un sitio web en un buscador (como Google o Bing). El posicionamiento mide cuán visible es una página para los usuarios en un buscador. Con las aplicaciones SPA, los buscadores no podían tener acceso al contenido, que era construido dinámicamente, por lo que muchas partes del sitio web quedaban sin indexar.

b) ***Performance***. Al hacer solo una petición al servidor se deben descargar todos los recursos. Frecuentemente, debido a la complejidad de las aplicaciones, los recursos son bastante pesados, por lo que su descarga demora cierto tiempo y ocasiona, además, que la página no se cargue al instante. Asimismo, al utilizar peticiones independientes solo para obtener datos estos no se muestran durante la carga.

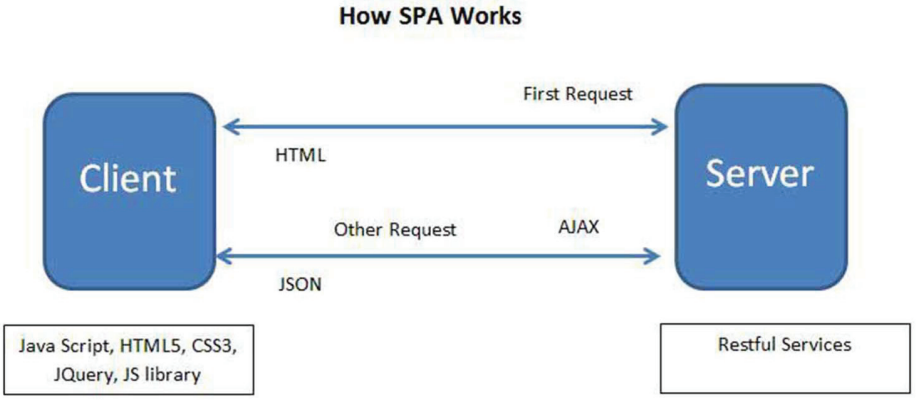

Figura 1. Funcionamiento de arquitectura SPA
Fuente: Bajaj (2014)

c) **Mantenibilidad.** Conforme la aplicación web se torne más compleja y conste de una mayor cantidad de pantallas, también la organización del código se complicará. Tomando en cuenta que se utilizará Javascript para la construcción de la página, es necesaria una organización distinta del código para que este pueda ser entendido y mantenido con facilidad. Esto se soluciona, en cierta medida, utilizando *frameworks* como el AngularJS o librería como el Backbone. Si la aplicación es demasiado compleja no será suficiente el uso de estas herramientas.

2. Enfoques híbridos

De acuerdo a la arquitectura planteada, aún se generarían páginas en el servidor (*backend*), así como también este serviría como API para datos. En el navegador (*frontend*) se seguiría utilizando Javascript para gestionar la interacción del usuario, manipular el Document Object Model (DOM) de la página web y hacer peticiones al API del *backend*, si fuera necesario.

En Mozilla Developer Network (s. f.) se sostiene que el DOM es

> una interface de programación para documentos HTML, XML y SVG. Provee una representación estructurada del documento (un árbol) y define la forma como esta estructura puede ser accedida por programas que puedan cambiar la estructura del documento, su estilo y su contenido.

Al tener acceso al DOM, utilizando Javascript, su manipulación se simplifica en el entorno cliente, y con el avance de los navegadores se vuelve más rápida y eficiente.

En la implementación del *backend* no se presentarían mayores cambios. Con la formulación del patrón arquitectónico Model 2 (Seshadri, 1999) se sientan las bases de cómo debía construirse una aplicación web, generada desde el servidor (figura 2).

Según Robbins (2011), este modelo no implementa totalmente el patrón arquitectónico Model View Controller (MVC). Sucesivas implementaciones o *frameworks*, como el Model Template View (MTV) en el *framework* Django para Python, o la implementación del MVC utilizando Active Records en Ruby on Rails, sí lo hacen. De esta manera, organizan mejor el código en el *backend*, permitiendo un desarrollo más rápido, ordenado y mantenible (figura 3).

A pesar de los avances en el *frontend* y el *backend*, aún se mantenía la complejidad de estas aplicaciones híbridas. Se debía tener un equipo de desarrolladores que dominaran tecnologías de *frontend* (usualmente HTML + Javascript + CSS más *frameworks* referentes a estos), y otro equipo de desarrolladores que dominaran tecnologías de *backend* (Django, Ruby on Rails, JSP, ASP.NET, etc.).

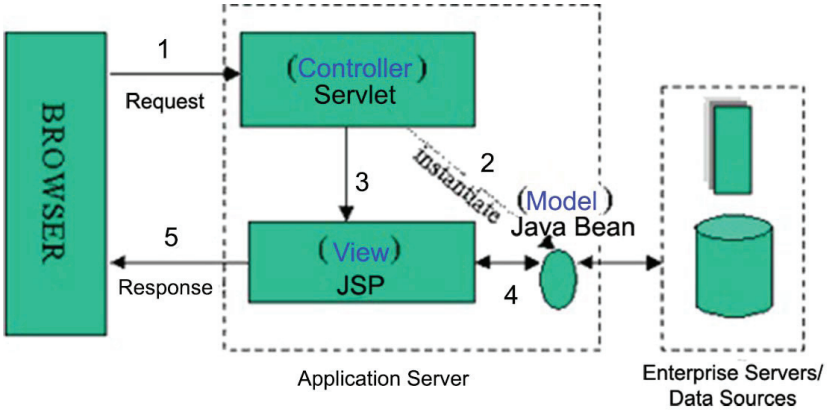

Figura 2. Arquitectura del Modelo 2
Fuente: Seshadri (1999)

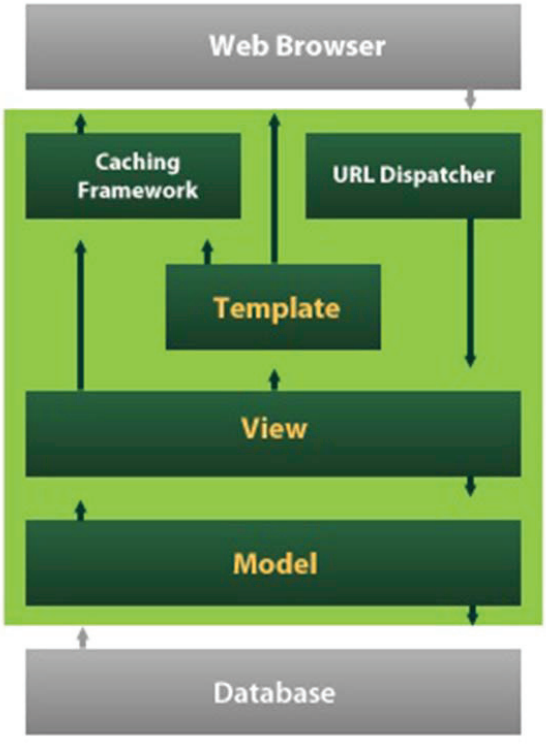

Figura 3. Arquitectura MTV en Django
Fuente: http://blog.chattyhive.com

Con la aparición de Node.js, el panorama empezó a cambiar. En Wikipedia se indica que Node.js es

> un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

Al basarse en ECMAScript, se pueden desarrollar aplicaciones Node.js utilizando el lenguaje de programación Javascript.

Por estas consideraciones, se comenzó a pensar en una arquitectura web que permita ejecutar un mismo código, tanto en el cliente como en el servidor. De esta manera, es perfectamente factible reutilizar un código que genere, modifique y renderice una página web en el servidor del cliente, dependiendo del contexto. Es así como nace el concepto de una aplicación isomórfica.

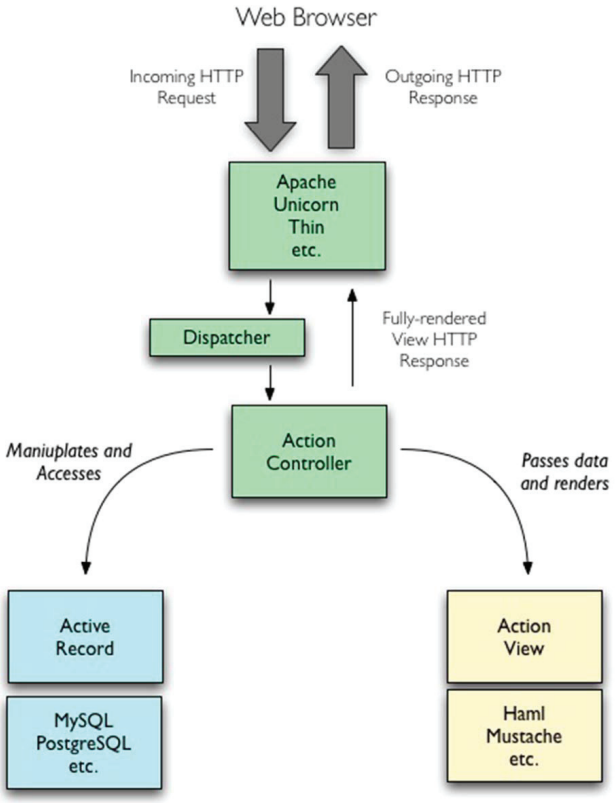

Figura 4. Arquitectura Active de Ruby on Rails
Fuente: Robbins (2011)

3. Aplicaciones isomórficas

Para Robbins (2011), isomórfico significa que cualquier línea de código (con algunas excepciones) de una aplicación web puede ser ejecutada en el cliente así como en el servidor. De esta forma, la reutilización del código sería una realidad. Aprovechando el concepto de DOM y su implementación en el estándar HTML5, se podrían crear o manipular estructuras dentro del DOM de una página, ya sea desde el cliente (como usualmente se hacía) o desde el servidor. Para lograr esto se necesitarían *frameworks* o librerías que nos permitan organizar nuestra aplicación. Según Brehm (2013), los *frameworks* o librerías que usemos deberían permitirnos realizar lo siguiente:

a) ***Routing* o enrutamiento.** En el caso de aplicaciones con arquitectura Single-Page App (SPA), las rutas que se ingresan necesitan ser interpretadas para poder efectuar cambios en las pantallas. Las rutas también deberán ser interpretadas por parte del servidor para renderizar una página.

b) **Obtención y persistencia de datos.** Debe ser posible describir recursos utilizando arquitectura REST. Fielding y Taylor (2002) indican que una arquitectura REST nos permite organizar todas nuestras entidades en recursos. Para lograr la obtención de datos de estos recursos, se obtiene una representación de estos mediante llamadas http y en un formato determinado (en su mayoría en notación JavaScript Object Notation (JSON).

c) **Renderizado de vistas.** Es necesario generar elementos del DOM isomórficamente. Cualquier modificación en el DOM puede ser realizada en el cliente, así como en el servidor.

d) **Armado y empaquetado.** Puesto que desde ahora se va a usar una aplicación en el cliente (en el navegador) que utilice varias librerías (especialmente de Javascript) será necesario emplear herramientas que faciliten el manejo de dependencias.

4. Desarrollo de una aplicación isomórfica

Se realizó una aplicación que permite crear y listar evaluaciones para proyectos de alumnos. Los requerimientos funcionales planteados son los que se detallan a continuación: listar las evaluaciones creadas, seleccionar una evaluación y registrar una nota para cada uno de los criterios de calificación (esto se hará por alumno o equipo). El código de esta aplicación se encuentra en el repositorio GIT: https://github.com/jsatch/evaly/ tree/isomorphic

La aplicación contará con tres nodos: el *frontend*, el *backend* y un servidor de base de datos MongoDB. En el navegador *frontend* se utilizarán las tecnologías HTML, Javascript y CSS, mientras que en el *backend* se usará Node.js (https://nodejs.

org/en/) como entorno de ejecución de aplicaciones. La base de datos que se utilizó fue MongoDB (https://www.mongodb.org/) por su fácil integración con Javascript mediante la utilización de objetos de tipo JSON.

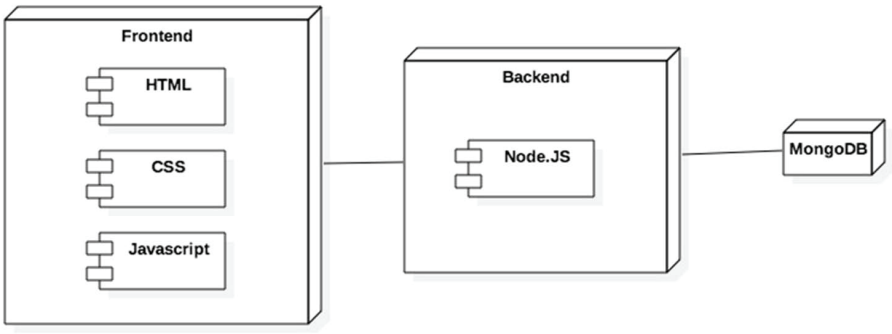

Figura 5. Arquitectura en grande
Elaboración propia

4.1 Librerías de *frontend*

En la figura 6 se muestran algunas librerías utilizadas en el nodo *frontend*.

a) **React.** Se utilizó la librería React para la gestión del DOM. Esta librería permite modificar el DOM utilizando Javascript en el cliente. Además, cuenta con el concepto de DOM virtual, que es una representación paralela del DOM de una página web, realizada por la librería React. Por esta razón, es factible su manipulación no solamente desde el cliente, sino también desde el servidor. El objetivo principal de React es ser la librería encargada del renderizado de las vistas (interfaces de usuario). React cubre perfectamente este punto, que es un requerimiento para las aplicaciones isomórficas, como lo indica Brehm (2013). React permite definir componentes que luego serán renderizados como elementos del DOM. Para esto, se crearán archivos con el lenguaje JSX, que facilitan la definición de los componentes que formarán la pantalla, que luego se traducirán al código Javascript, tal como se muestra en el código 1.

   Los archivos con lenguaje JSX son una extensión del Javascript con partes de XML. Este archivo se transformará después en un archivo de Javascript que será interpretado por el navegador (o por el servidor).

b) **React-Bootstrap.** Al definir un componente en JSX mediante etiquetas, estas serán mapeadas a elementos del DOM de una página (div, input, etc.). Por lo tanto, podríamos definirles atributos como *class* o *style* si queremos darle algún estilo gráfico.

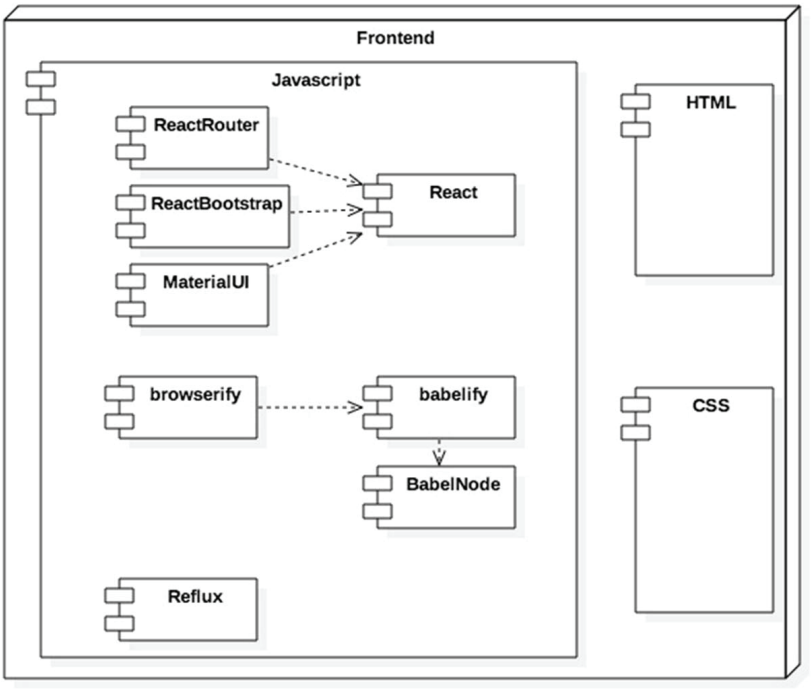

Figura 6. Algunas librerías utilizadas en el *frontend*
Elaboración propia

Código 1: client/components/EvalyApp.react.js

```
import React from 'react';
import EvaluationListPage from './EvaluationListPage. react';

export default class EvalyApp extends React. Component {
   constructor (props) {
     super (props);
    }
   render ( ) {
     return (
       <EvaluationListPage  />
      );
    }
  }
```

Una de las librerías de estilos CSS es Bootstrap, que nos permite asignarles clases (*class*) predefinidas a los componentes de DOM para así cambiarles el aspecto y el comportamiento.

Para posibilitar la utilización de Bootstrap dentro de React es necesario utilizar la librería React-Bootstrap.

c) **React-Router.** Como sugiere Brehm (2013), es necesario tener un mecanismo que nos ayude con el manejo de rutas (asignación y enrutamiento). Por defecto, cuando alguien escribe una ruta en el navegador, se debe realizar una petición al servidor web y armar el DOM de la página web (utilizando React). Luego, cualquier cambio en la página (estructura del DOM) deberá ejecutarse desde el cliente, y si se hace utilizando un URL, deberá hacerse localmente (no realizar una petición al servidor). Para esto se utiliza la librería React-Router.

d) **Flux.** Adicionalmente, se aplicará el patrón arquitectónico de *frontend* llamado Flux. Este patrón "complementa el uso de React permitiendo un flujo de datos unidireccional", como lo indica la documentación oficial proporcionada por Facebook. Por este motivo, se utilizará una librería de Javascript llamada Reflux (https://github.com/reflux/refluxjs) que facilita la implementación de este patrón arquitectónico en una aplicación Javascript.

e) **Librerías adicionales en el *frontend*.** Se utilizará la especificación ECMAScript 6 (ES6) de Javascript, la cual trae algunos cambios en la sintaxis (una orientación mayor a objetos) y por ser una versión nueva, aún no es soportada por algunos navegadores. Por esto, se utilizan las librerías Babel y Babelify, que se encargarán de traducir el código Javascript con estándar ES6 a estándar ES5. Igualmente, el Babel permite realizar la transformación de código JSX a Javascript.

Tabla 1

Librerías utilizadas en el frontend

Nombre de la librería	Dirección web
React	https://facebook.github.io/react/
React-Router	https://github.com/rackt/react-router
React-Bootstrap	https://react-bootstrap.github.io/
Reflux	https://github.com/reflux/refluxjs
MaterialUI	http://material-ui.com/#/

Elaboración propia

Se usará también la librería browserify. Debido a que se emplearán varios archivos Javascript (por lo menos un archivo por componente React), será necesario manejar de alguna manera las dependencias. Se utilizará el npm para gestionar las dependencias de una aplicación Node.js. Además, se manejará el concepto de paquetes y de *imports* (ECMAScript 6) que nos proporciona Javascript. Por ende, necesitamos una herramienta que agrupe todas las librerías para poder descargarlas desde el navegador y que sean interpretadas por él.

4.2. Implementación de componentes de interfaz gráfica

Como se comentó, se implementaron los componentes de la interfaz gráfica utilizando la librería React. La jerarquía fue la siguiente:

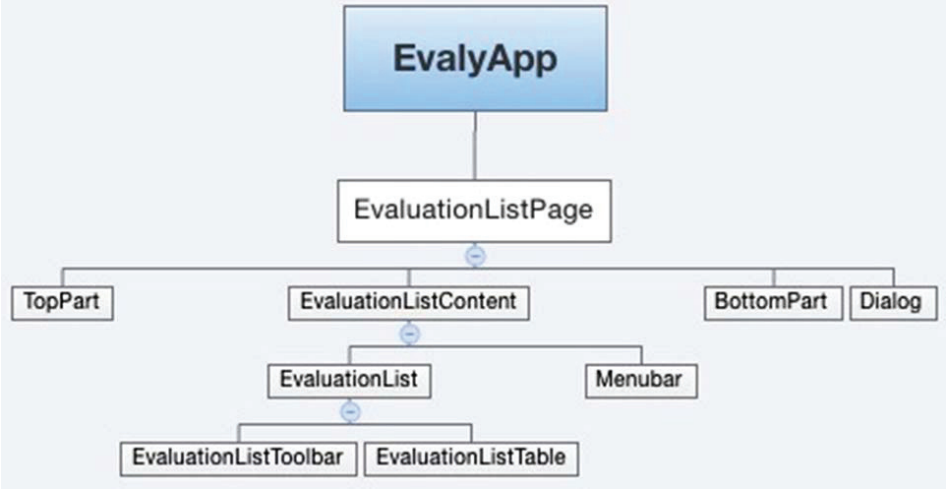

Figura 7. Estructura de componentes React
Elaboración propia

Cada componente puede tener componentes HTML que no están diagramados en la figura 7. Tampoco se encuentran diagramados los componentes utilizados en la librería MaterialUI para manejo de componentes de interfaz gráfica. Dentro de cada componente React se pueden programar los estados de cada componente; así, cada uno tiene su propia lógica de funcionamiento específico, dentro del método Render del componente React. En este método se define la lógica de renderizado del componente, como se muestra en el código 2.

Código 2: client/components/evaluations/EvaluationListPage.react.js

```
render () {
  var dialog;
  if (this.state.modal) {
     let standardActions  = [
       { text: 'Cancel', onTouchTap: thi.s._onDialogCancel.bind(this) }
     ];
    dialog = <Dialog
      ref="evaluationDialog"
      actions={standardActions}
      title="Evaluación"
      openInmediately={true}>
      <Evaluation initialEventConfig={this. state.eventConfig}/>
    </Dialog>;
  }
  return (
  <div className="container">
    <TopPart></TopPart>
    <EvaluationListContent
      evaluationList={this.state.evaluationList}
      onEvaluationStart={this.onEvaluationStart.bind(this)}>
    </EvaluationListContent>
    <BottomPart></BottomPart>
    {dialog}
  </div>
 );
}
```

En el cual se define que dado el cumplimiento de ciertos parámetros, se muestre el componente Dialog (componente de la librería MaterialUI). Cada componente React tiene variables de dos categorías: de estado (*state*) y propiedades (*props*). Según la documentación oficial de React, los componentes React funcionan como máquinas de estados. Cada estado de un componente React viene dado por un grupo de variables de estado, que lo definen. Por lo tanto, al cambiar una o más variables de estado del componente, este cambia su estado y se vuelve a renderizar.

Es importante resaltar que, según la documentación oficial de React, es conveniente mantener la mayor cantidad de componentes con un único estado (*stateless*) y que solamente los componentes padre sean los que manejen los cambios de estado. Por otro lado, las propiedades son variables que sirven para recibir (o pasar) datos entre componentes hijos y padres. El sentido del flujo de los datos deberá ser único (de hijos a padres) y este es uno de los pilares sobre los que se fundamenta el manejo de datos de la interfaz gráfica.

En el código 2 al componente EvaluationListContent se le pasa dos propiedades: evaluationList y onEvaluationStart, siendo EvaluationList el listado de las evaluaciones y onEvaluationStart un objeto función de tipo *callback* que reaccionará cuando se inicie una evaluación. La aplicación constará de dos páginas: una de *login* y otra de listado de evaluaciones. La pantalla de login únicamente servirá para permitirnos ingresar a la aplicación, mientras que en la pantalla de listado de evaluaciones es donde se encuentra la lógica del registro de datos de las evaluaciones. Cabe resaltar que –para fines del artículo– no se implementó la conexión a base de datos para el caso del *login*, sino solamente para graficar la presencia de dos páginas distintas que se carguen desde el servidor, como se aprecia en el código 3.

Código 3: client/components/app.js

```
var Route = Router.Route;

var routes =(
  <Route path="/" handler={App}>
    <Route path="login" handler={LoginPage}/>
    <Route path="main" handler={EvaluationListPage}/>
  </Route>
);
var RouteHandler = Router.RouteHandler;

export default class App extends React.Component{
 render(){
     <RouteHandler />
 }
};
Router.run(routes, Router.HistoryLocation, (Root) => {
 var initialState = JSON.parse(
  document.getElenentById('initial-state').innerHTML);
 React.render(<Root initialState={initialState} />, document.getElementById('container'));
});
```

Se definen dos rutas: */login*, que abrirá el componente React LoginPage, y */main*, que abrirá el componente React EvaluationListPage. Con base en lo explicado, este enrutamiento se realizará solamente en el cliente. Los datos iniciales que se enviarán a cada uno de los componentes se hará como propiedades dentro del atributo initialState, que será obtenido de la misma página web, la cual se generará en el servidor (componente *div* con *id initial-state)*.

4.3 Renderizado en el servidor

Al haberse definido los componentes, y de acuerdo con los requerimientos propuestos por Brehm (2013), ahora debemos reutilizar los componentes React creados para el *frontend* desde el *backend*, para que nuestra aplicación pueda ser isomórfica.

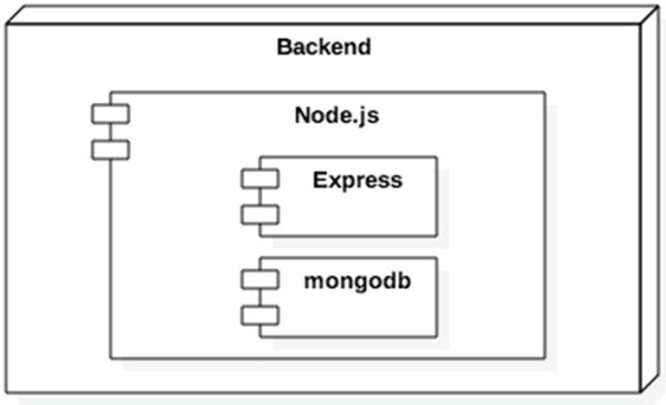

Figura 8. Librerías utilizadas en el *backend*
Elaboración propia

En la figura 8 se aprecia que se está utilizando como plataforma tecnológica en el *backend* Node.js, ya que nos permite utilizar Javascript como lenguaje de programación. Dentro de nuestra aplicación Node.js, se empleará como servidor web el *framework* Express. Tal como lo indica la documentación oficial de Express, este nos "proporciona un conjunto de métodos HTTP" que nos permiten realizar una aplicación web de forma rápida. Dichos métodos los aplicaremos de dos formas: para servir peticiones que generen páginas web (utilizando React) y servir peticiones de datos (peticiones REST) como lo ve necesario Brehm (2013) dentro de sus requerimientos. Por último, para la conexión a la base de datos Mongodb se utilizará la librería Mongodb para Node.js.

Tabla 2
Librerías utilizadas en el backend

Nombre de librería	Dirección web
Express	http://expressjs.com/es/
Mongodb	https://docs.mongodb.org/ecosystem/drivers/node-js/

Elaboración propia

Para la generación de páginas web, se define un archivo de rutas que están relacionadas con un método de un objeto Javascript, el cual se encarga de generar los componentes React para la generación de la pantalla, como se muestra en el código 4.

Código 4: server/routes.js

```
export default {
  '/login' : LoginController.show,
  '/main' : ListaEvaluacionesController.show
};
```

Dentro de este archivo se configura el enrutamiento de la URL */main*, que llamará al método *show* de la clase Javascript (véase el código 5), encargado de crear e inicializar los componentes React.

Código 5: server/ListaEvaluacionesController

```
export default class ListaEvaluacionesController {
  static show( req, res){
    var unsuscribe = EvaluationStore.listen((data) => {
      if (data .event === EvaluationStore. events. EVALUATION_LOADED){
        var reactHtml = React.renderToString(<EvaluationListPage evaluationList={data.data}/>);
        res.render('index.ejs', {
          reactOutput: reactHtml,
          state: JSON.stringify({
            evaluationList: data.data
          })
        });
        unsubscribe();
      }
    });
    (evaluationActions.listEvaluationsAction)()
  }
}
```

Como se muestra en el código 5, se llama al método RenderToString, pasándole como parámetro el componente React EvaluationListPage (previamente definido), con la propiedad EvaluationList que ha sido obtenido de una consulta al API Rest del servidor (*backend*).

Este método convierte el componente React generado a una cadena de texto en formato HTML. Esta cadena de texto (mediante el método Render) se pasa como parámetro a un *template*, tal como se aprecia en el código 6.

Código 6: views/index.ejs

```
<html>
  <head>
    <meta charset="UTF-8"/>
    <title>Evaly</title>
    <link rel="stylesheet" type="text/css"
      href="https//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">
   <link rel="stylesheet" type="text/css" href="app.css">
   <link href='http//fonts.googleapis.com/css?family=Roboto:400,300,500' rel='stylesheet' type='text/css'>
  </head>
  <body>
    <section id="container"><%- reactOutput %></section>
    <script id="initial-state" type="application/json"><%- state %></script>
    <script scr="app.js"></script>
  </body>
</html>
```

El *template* index.ejs es un archivo con código HTML, que es cargado por el Express y permite completarlo con los parámetros que le vaya pasando. En este caso, al método Render le hemos pasado el parámetro *reactOutput* y el parámetro *state*, reemplazando en las partes del mismo nombre en el *template* index.ejs. De esta manera tenemos una página HTML con los componentes React ya generados (*reactOutput)*, así como los datos que la página utilizará para cargarse (*state*). Cabe señalar que el *state* será utilizado luego como una propiedad del componente React cuando se haga la carga desde el cliente.

5. Funcionamiento isomórfico

Cuando el usuario pone la dirección en el navegador, se realiza una petición *http* de tipo GET al servidor. El servidor Express (*backend*) recibe la petición y la direcciona hacia método del objeto Javascript, que ha sido configurado en el archivo *routes.js*. Es en este método donde se generan los componentes React, que posteriormente serán componentes DOM. De igual manera, luego de generar los componentes se genera la página web HTML, que será devuelta al cliente (navegador). Esta página se genera tomando como base un *template* (index.ejs) y llenándolo con dos valores: los componentes React generados y los datos que serán utilizados por estos.

En el cliente (navegador) la página obtenida del servidor (*backend*) es renderizada. Luego de cargar la página se inicia el enrutamiento (usando el React-Router). Debido a que el URL sigue siendo el mismo de la petición GET que inició todo, se vuelve a cargar el DOM (ahora en el cliente). React compara que el DOM virtual

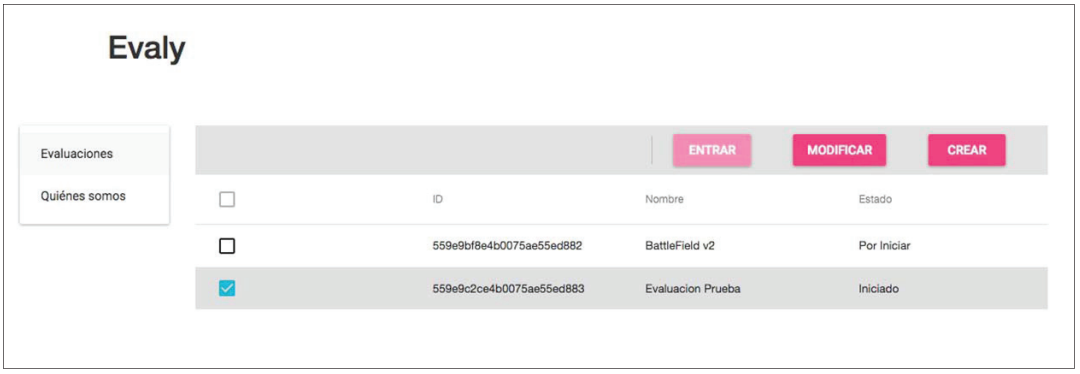

Figura 9. Pantalla con el listado de evaluaciones
Elaboración propia

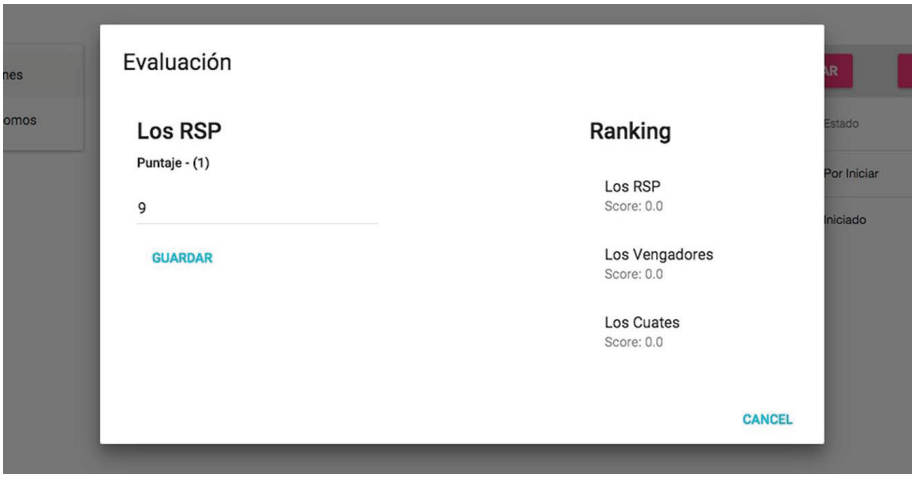

Figura 10. Pantalla de ingreso de puntaje a evaluación
Elaboración propia

(DOM de React) no ha variado; entonces no hace ningún cambio. Si más adelante cambia el DOM virtual, como, por ejemplo, que se agregue una nueva evaluación a la grilla, se vuelven a renderizar los componentes que han cambiado.

6. Conclusiones

El presente artículo muestra una manera de estructurar una aplicación web isomórfica utilizando Node.js y la librería React para interfaces gráficas. Esta configuración no es única, pero se ha utilizado debido a su sencillez y porque nos permite mostrar y cumplir con los requerimientos de aplicaciones isomórficas que Brehm (2013) define.

El tratamiento de las interfaces gráficas web, desde el punto de vista de los componentes, facilita su creación y la reutilización, incrementando los tiempos de desarrollo y disminuyendo la tasa de errores. Es pertinente referir que la tendencia a tratar las páginas web como componentes es algo que aún no se encuentra estandarizado. Existen muchas tecnologías que buscan alcanzar el grado de estándar (Polymer de Google, Web components, React de Facebook), pero hasta el momento no hay ninguna especificación.

Conforme las aplicaciones web se tornan más complejas e interactivas, se requieren nuevas técnicas que mejoren la experiencia de los usuarios. Sumado a las constantes mejoras en los navegadores, hacen falta herramientas y técnicas que permitan sacar el mayor provecho al ambiente de ejecución del navegador. Javascript, con todas sus librerías (incluida React), facilita la creación de este tipo de aplicaciones web.

A pesar del auge de las tecnologías móviles, la web aún se sigue manteniendo como una plataforma útil y versátil para el desarrollo de aplicaciones. Teniendo en cuenta que normalmente el desarrollo de aplicaciones móviles nativas son costosas, y se desarrollan en forma independiente según la plataforma, una aplicación web es versátil y puede funcionar en cualquier sistema operativo móvil y en cualquier tamaño de pantalla (diseño adaptativo), ya que se ejecuta en el navegador. El reto, entonces, es que esta aplicación web iguale la misma experiencia visual que una aplicación móvil nativa proporciona.

Anexo

PROCEDIMIENTO PARA CONFIGURACIÓN DE AMBIENTE DE EJECUCIÓN

- Instalar el ambiente Node.js (https://nodejs.org/en/)
- Actualizar el *npm*, que es el gestor de paquetes de Node.js. Para esto ejecútese el comando *sudo npm install npm -g*
- Descargar la aplicación del sitio web https://github.com/jsatch/evaly/tree/isomorphic
- Instalar todas las dependencias (librerías requeridas). Para esto ejecutar el comando *npm install*
- Para compilar y generar los archivos necesarios (ejecución del *browserify*), ejecutar el comando *npm run build*
- Para comenzar la ejecución del servidor web Express, ejecutar el comando *npm start*

Referencias

Bajaj, P. (11 de junio de 2014). Overview of Single-Page Application (SPA). Recuperado de http://www.c-sharpcorner.com/Blogs/15740/overview-of-single-page-application-spa.aspx

Brehm, S. (8 de noviembre de 2013). *The future of web app is ready? isomorphic JavaScript*. Recuperado de http://venturebeat.com/2013/11/08/the-future-of-web-apps-is-ready-isomorphic-javascript/

Fielding, R., y Taylor, R. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology, 2*(2), 115-150. DOI: 10.1145/514183.514185

Flux. (s. f.). Documentación oficial. Recuperado de https://facebook.github.io/flux/docs/overview.html

Mozilla Developer Network (s. f.). Document Object Model (DOM). Recuperado de https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model# DOM_interfaces

React. (s. f.). *Interactivity and dynamic UIs*. Documentación oficial. Recuperado de https://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html

Robbins, C. (18 de octubre de 2011). Scaling Isomorphic Javascript Code. Recuperado de http://blog.nodejitsu.com/scaling-isomorphic-javascript-code/

Seshadri, G. (29 de diciembre de 1999). *Understanding JavaServer Pages Model 2 architecture. Exploring the MVC design pattern*. Recuperado de http://www.javaworld.com/article/2076557/java-web-development/understanding-javaserver-pages-model-2-architecture.html