Aplicaciones Modulares en JavaScript

Modular Applications in JavaScript

Daniel Esteban Puerta Ortíz - daespuor1991@gmail.com

Luis Felipe Becerra Arias - luisbecerra.424@gmail.com

Universidad Distrital Francisco José de Caldas

Tipo: Artículo de Investigación

Para citar este artículo: Puerta Ortiz D.E. & Becerra Arias L.F. (2014) Aplicaciones Modulares en JavaScript. Revista TIA, pp 17-29.

Fecha de recepción: 05 de noviembre de 2014 **Fecha de aceptación:** 05 de junio de 2014

Revista Digital TIA
Tecnología Investigación y Academia

Resumen

La creciente popularidad de JavaScript en los últimos años y los mitos que aún se tienen sobre él, fueron las motivaciones para escribir el presente artículo, cuyo objetivo es demostrar que en este lenguaje es posible crear aplicaciones complejas, mantenibles y con capacidad de evolucionar mediante módulos reutilizables. Para esto se realizó una revisión de algunas de las formas como se pueden definir módulos en JavaScript, tomando como base conceptual las principales características de los sistemas modulares y del mismo lenguaje a la luz de los lenguajes de scripting. Finalmente se identificó que tal definición de módulos está en proceso de estandarización.

Palabras clave: componente, JavaScript, lenguajes basados en objetos, lenguajes de scripting, modelos de componentes, módulo.

Abstract

The growing popularity of JavaScript in recent years and the myths that still exist related with him, were the motivations for writing this article , which aims to shown that it is possible to create complex and maintainable applications with capacity to evolve through reusable modules. For this reason was realized a review of the forms for define modules in JavaScript, taking as conceptual bases the main characteristics of modular systems and of the same language in the light of scripting languages. Finally was identified that such definition of modules is in the standardization process.

Key words: component models, component, JavaScript, module, object-based languages, scripting languages.

I. Introducción

Durante los últimos años los denominados lenguajes de scripting han estado ganando popularidad entre los desarrolladores, como se puede evidenciar en el sitio web IEEE SPECTRUM que a mediados de julio del 2014 publicó una lista con los lenguajes de programación más populares, basándose en métricas tomadas de sitios como Google, Xplore, Github y por supuesto la IEEE. [1].

Como se puede apreciar en la figura 1, a pesar de que los lenguajes tradicionales la encabezan, el quinto puesto es ocupado por Python, seguido de JavaScript, PHP y Ruby en los siguientes puestos.
Otra prueba de la popularidad de estos lenguajes se encuentra en la medición realizada por RedMonk (figura 2) que correlaciona las discusiones acerca del tema en Stack Overflow y el uso de los mismos manifestado en GitHub; JavaScript es el lenguaje más popular seguido muy de cerca por Java. PHP ocupa el tercer puesto, Python el cuarto y Ruby se ubica en el quinto lugar. [2]

Ranking IEEE SPECTRUM

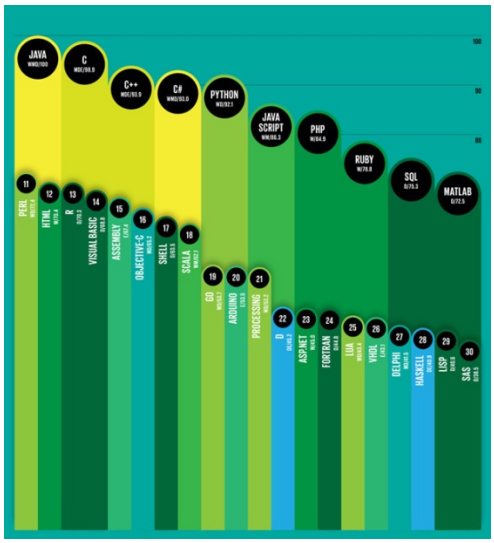

Figura 1. Ranking IEEE SPECTRUM, [1]

Ranking RedMonk

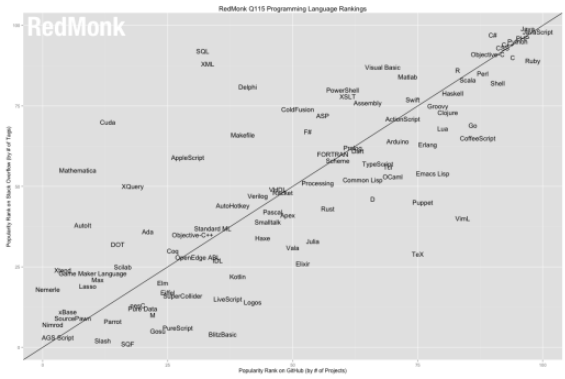

Figura 2. Ranking RedMonk.

Finalmente, el sitio TIOBE que realiza mediciones de popularidad cada mes basado en motores de búsqueda como Google, Amazon, Wikipedia, Yahoo, Bing y Baidu, para el mes de abril del 2014 ubica a Java de nuevo en el primer lugar y hasta el sexto, séptimo y octavo puesto encontramos a JavaScript, PHP y Python respectivamente. [3] Sin embargo, sigue siendo evidente que los lenguajes de scripting están cobrando fuerza y no es difícil encontrarlos entre los primeros diez puestos de la mayoría de mediciones de este tipo, y es aún más evidente que JavaScript se está convirtiendo en uno de los lenguajes de programación más usados, a pesar de que durante algún tiempo se consideraba como un intermediario entre la interfaz de usuario y lenguajes de servidor (PHP, Java, etc.).

Con el tiempo y la aparición de herramientas como Node.js las capacidades de JavaScript se han ampliado, hasta posibilitar la construcción de aplicaciones completamente desarrolladas en él. A pesar de esto y gracias a las malas prácticas de desarrollo, aún existe la creencia de que no es útil para realizar productos de software de gran tamaño, lo que es una motivación para analizar de qué manera se pueden definir módulos reutilizables en este lenguaje que simplifiquen la complejidad de aplicaciones grandes y que faciliten su mantenimiento y evolución. Con este objetivo se ha organizado el artículo de la siguiente manera. La sección Sistemas modulares, tratará los conceptos de módulos, componentes y aplicaciones o sistemas modulares; la sección JavaScript, un lenguaje de scripting definirá a JavaScript desde el punto de vista de los lenguajes de scripting y la sección; la sección Definición de módulos en JavaScript mostrará cómo

se pueden crear aplicaciones modulares utilizando herramientas como CommonJS, AMD y Servicios Web en JavaScript.

II. Sistemas modulares

En esta sección se definirán los sistemas modulares como sistemas de software mantenibles con capacidad de evolucionar.

Los sistemas modulares y la ingeniería de software basada en componentes (CBSE) surgen de la necesidad de constituir una ingeniería de software madura, donde existan unidades funcionales que puedan ser reutilizadas en la construcción de sistemas de software con el fin de disminuir esfuerzos, costos y otros recursos, emulando lo que ocurre en otras ingenierías como la civil o la electrónica. Dichas unidades funcionales se conocen como módulos o componentes, tienen una funcionalidad determinada y pueden integrarse para conformar sistemas más complejos, como si de ladrillos de un edificio se tratase.

Estos sistemas pueden ser creados con un enfoque de abajo hacia arriba (bottom-up), donde se desarrollan primero los componentes y más adelante se combinan para generar el sistema [4]. Este método se llama "composición" y es apropiado para trabajar con modelos de proceso iterativos e incrementales, tanto así que incluso existen métodos para mapear requerimientos a componentes, como se evidencia en [5], donde cada nueva iteración significa una aproximación más cercana al sistema final (principio del incremento) [5]. Sin embargo, también pueden construirse por medio de un enfoque de arriba hacia abajo (top-down) llamado "descomposición", donde primero se planifica el sistema y cómo interactúan sus componentes antes de entrar a desarrollar cada uno de manera individual [5]. En este enfoque se ve cada módulo como una caja negra donde se ignoran sus detalles internos y solo se tiene en cuenta lo que requieren para funcionar y los servicios que proveen, de tal manera que puedan intercomunicarse.

Además los módulos o componentes deben estar en capacidad de integrarse con nuevos componentes provenientes de librerías y aplicaciones de terceras partes, ser integrados en sistemas diferentes para el que fueron creados o integrarse con otros componentes para conformar un componente compuesto. Para esto es necesario que se apliquen los patrones de separación de responsabilidades (GRASP) alta cohesión y bajo acoplamiento [6]. La cohesión tiene relación con la estructura interna de los módulos, cuyos elementos deben estar estrechamente relacionados y tener un objetivo común. El acoplamiento tiene que ver con la relación entre módulos de tal manera que la dependencia entre estos sea mínima [4]. De esta forma se garantiza que los módulos puedan ser agregados, retirados y modificados sin que haya un impacto negativo en el sistema (figuras 3 y 4).

Representación gráfica de la baja cohesión y el alto acoplamiento

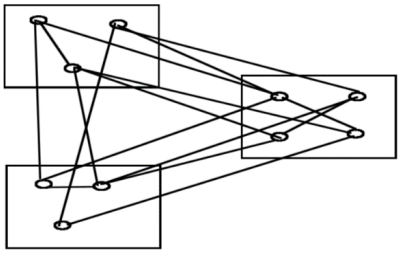

Figura 3. Baja cohesión y alto acoplamiento [4]

Representación gráfica de la alta cohesión y el bajo acoplamiento

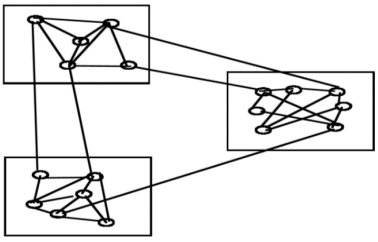

Figura 4. Alta cohesión y bajo acoplamiento [4]

En términos generales los componentes o módulos dotan a los sistemas de software o aplicaciones la capacidad de mantenerse, adaptarse al cambio y evolucionar sin importar su complejidad. Por lo tanto en la sección IV se mostrará cómo crear este tipo de elementos en JavaScript.

II. JavaScript, un lenguaje de scripting

En esta sección se definirá JavaScript con base en los lenguajes de scripting.

Existen muchas dudas sobre las características que debe tener un lenguaje de programación para ser

lenguaje de scripting. Esto se debe a que los recientes avances tecnológicos han desvanecido aún más la frontera entre los lenguajes tradicionales y dichos lenguajes. Hace una década esta tarea de taxonomía era menos compleja, los lenguajes tradicionales como C o Java eran propicios para el desarrollo de aplicaciones o componentes y los de scripting servían como intermediario, como una herramienta auxiliar que por su flexibilidad eran idóneos para unir componentes o en el caso de JavaScript contener pequeñas lógicas de negocio como validación de formularios, controles de interfaz, etc. Ahora es posible crear aplicaciones completas y complejas con solo estos lenguajes, llegando a competir con los tradicionales y alcanzando popularidad entre los desarrolladores, por lo que categorizarlos es una tarea más difícil.
Según [7], los lenguajes de scripting son interpretados en lugar de compilados, las variables no necesitan ser declaradas y tienen una asignación de tipos dinámica en lugar de estática. Sin embargo, con el surgimiento de la compilación JIT (Just in Time) los lenguajes de programación compilados pueden tener una flexibilidad similar a los que no lo son y en algunos lenguajes de programación tradicionales, como Java, es posible la asignación de tipos de datos como si de objetos se tratasen (en algunos casos Java es considerado como un lenguaje de scripting).

Por otro lado, en [8] se presentan como características de los lenguajes de scripting, la utilización de tipos de datos de alto nivel (arreglos, listas, etc.) y el manejo de memoria por el recolector de basura; pero estas características son convencionales, tanto en estos lenguajes como en los tradicionales. Finalmente, se señala que los lenguajes de scripting tienen afinidad con el paradigma de programación procedimental, no obstante, en la actualidad esto ya no es tan cierto porque muchos han migrado hacia la programación orientada a objetos, como el caso de PHP, Python y Perl, además de otros que fueron creados bajo este paradigma, como Ruby que es tan orientado a objetos como el mismo SmallTalk [7].
Como se puede ver, la ambigüedad abunda a la hora de determinar si un lenguaje de programación es de scripting o no basándose solo en sus características, tal vez porque lo que importa no es eso, sino cómo se utilice, lo que puede evidenciarse en la primera parte de la definición dada en [9], donde "Un lenguaje de scripting es un lenguaje de programación simple usado para escribir una lista de comandos ejecutables llamado script". Esto significa que sin importar si el lenguaje es compilado, interpretado, orientado a objetos, procedimental, manejado por una máquina virtual, con asignación de tipos de datos dinámicos o estáticos, es posible realizar scripts.

JavaScript es uno de los denominados lenguajes de scripting (PHP, Perl, Ruby, Python, VBScript, etc.) al que más se le ajusta esta definición por la forma como fue sido utilizado durante muchos años y que poco a poco ha ido quedándose atrás a medida que su popularidad crece. Esto se debe a la también creciente popularidad de internet (al ser un lenguaje principalmente desarrollado para aplicaciones Web) y según [10], al esfuerzo de los proveedores de navegadores de crear motores de procesamiento más potentes, junto con la posibilidad de crear aplicaciones del lado del servidor con Node.js (Que funciona con el motor V8), además de robustecer la funcionalidad del lado del cliente con la llegada de *frameworks* como JQuery que suavicen las diferencias entre navegadores y AngularJS que den una estructura al código siguiendo el patrón MVC.

Esto ha propiciado que JavaScript llegue a terrenos insospechados como al de las aplicaciones móviles mediante Phonegap, que en combinación con JQuery Mobile y Sencha pueden tener interfaces responsivas que incluso parecen nativas al manejo de sistemas embebidos (Arduino) y la creación de aplicaciones Stand-alone por medio de NodeWebkit.

A pesar de estos avances, sigue siendo un lenguaje estigmatizado sin razón. Un ejemplo de esto son las palabras de Miles Lines en [10], "Estéticamente no me gusta JavaScript....Es una cuestión personal", lo que indica que no tiene una razón técnica para juzgar el lenguaje, solo puede basarse en su estructura y es por eso que creo que CoffieScript es como JavaScript, pero con una sintaxis más limpia limitando un poco el control de los programadores sobre la puntuación del lenguaje.
Esto es importante, porque da cuenta de que aún existen inconformidades provocadas por su flexibilidad, que si bien permite abordar un problema desde muchas perspectivas, también ha conducido a la implementación de malas prácticas (como las variables globales), por parte de desarrolladores inconscientes, lo que a fin de cuentas genera la falsa concepción de que no es apto para desarrollar aplicaciones complejas y robustas. Para contradecir esto en la siguiente sección se muestran varias

opciones para crear aplicaciones modulares en este lenguaje.

III. Definición de módulos en JavaScript

En esta sección se presentaran algunas de las formas como se pueden definir módulos en JavaScript.
"Los módulos son la pieza integral de cualquier arquitectura de software robusta y típicamente ayudan a mantener las unidades de código de un proyecto limpiamente separadas y organizadas", a esta definición dada en [11] debe agregársele que desde el punto de vista de JavaScript los módulos son objetos cuyas propiedades (funciones) son tratadas como unidades funcionales individuales, que vendrían siendo el equivalente a los componentes utilizados en aplicaciones del lado del servidor [12]. Estos, como ya se mencionó, son el elemento constituyente de las aplicaciones o sistemas modulares definidos en la sección II.

Existen muchas formas de definir módulos en JavaScript, tanto para la creación de la interfaz de usuario (ComponentJS) como para la lógica de negocio, mediante frameworks (BackBone.js, Ember.js) como a través de código nativo (Patrones de diseño, ES6), tanto para definición asíncrona del lado del cliente (AMD) como síncrona del lado del servidor (CommonJS), estandarizados tanto para la creación de SOA (Servicios Web) como con otro estilo arquitectónico, entre otros. A continuación se trataran específicamente patrones para definición de módulos, AMD, CommonJS, Servicios Web y ES6.

A). Patrones para la definición de módulos

Los patrones de diseño orientados a la definición de módulos más importantes son el patrón módulo y el módulo revelado. El primero tiene la intención de emular el encapsulamiento y visibilidad de una clase en los lenguajes de programación orientada a objetos, es decir la capacidad de definir métodos y atributos de una clase como si fueran públicos (que puedan ser usados por otros módulos) o privados (que puedan ser usados solo por el mismo módulo). En el gráfico 1 es posible ver la implementación de un módulo mediante dicho patrón [11].

```
var testModule = (function () {
 var counter = 0;
 return {
 incrementCounter: function () {
 return counter++;
 },
 resetCounter: function () {
 console.log( "counter value prior to reset:
 " + counter );
 counter = 0;
0
 }
1
 };
2
 })();
3


4
 console.log(testModule.incrementCounter());
5
 testModule.resetCounter();
6
```

Gráfico 1. Declaración del módulo *testModule* mediante el patrón módulo [11]

Como se puede apreciar hay un atributo *counter* que por el simple hecho de estar fuera del *return* ya es privado y no puede ser invocado fuera del módulo al que pertenece. Por el contrario, se definieron dos operaciones públicas, *insertCounter* que incrementa un contador y *resetCounter*que lo reinicia. El resultado de ejecutar ambas operaciones se pone de manifiesto en la figura 5.

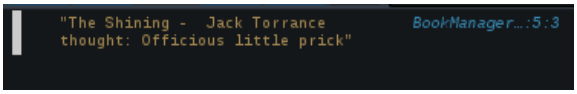

Figura 5. Resultado en consola de ejecutar las funciones del módulo 'TestMdoule'.

El patrón módulo revelado es una variación del patrón módulo donde todos sus miembros son privados y donde hay unas interfaces públicas para ciertas propiedades. El ejemplo anterior definido con el patrón módulo revelado tendría la estructura presente en el gráfico 2 [12].

```
var testModule = (function () {
 var counter = 0;
 function privateIncrementCounter() {
 return counter++;
 }
 function privateResetCounter() {
 console.log( "counter value prior to reset:
 " + counter );
 counter = 0;
 }
0
```

```
1  return {
2  incrementCounter:privateIncrementCounter,
3  resetCounter:privateResetCounter
4  };
5  })();
6
7  console.log(testModule.incrementCounter());
8  testModule.resetCounter();
```

Gráfico 2. Declaración del módulo *testModule* mediante el patrón módulo revelado.

La principal desventaja de definir módulos mediante patrones es que existen muchas más variantes del patrón módulo (que pueden ser consultadas en [11]), las cuales terminan provocando "el síndrome del salvaje oeste", donde cada desarrollador hace lo que considere y como lo considere. Además no hay manera de controlar la carga asíncrona y paralela de módulos, las dependencias entre estos, la disminución de las etiquetas HTML, la optimización y algunas características más [12]. Como una alternativa para solucionar estos problemas aparecen CommonJS y posteriormente AMD.

B). AMD (Asynchronous Module Definition)

A excepción de los lenguajes tradicionales, JavaScript no proveía en su versión ECMA-262, la forma de definir módulos de manera estándar, limpia y ordenada, la única forma de hacer esto era mediante los patrones de diseño mencionados en el numeral *A* con las desventajas que esto tenía, por eso se crearon APIs como AMD y CommonJS, que pueden trabajar en conjunto con unas herramientas llamadas script loaders, cuya función es evitar que los scripts sean declarados manualmente en páginas HTML.

Tradicionalmente esta declaración se da mediante etiquetas <Script>, donde es necesario añadir la ruta del recurso (similar a lo que sucede con la etiqueta <Link> para especificar CSS); sin embargo esto presenta un problema y es que el navegador carga los elementos de manera secuencial, lo que puede tener efectos negativos en la aplicación si las etiquetas no están correctamente ordenadas o si no se etiquetaron scripts, de los cuales dependen otros scripts para funcionar. Para solucionar esto los scripts loader cargan scripts junto con sus dependencias de manera automática y en algunos casos lo hacen de manera asíncrona.

Cargar scripts de esta forma puede ser beneficioso para el rendimiento de las aplicaciones (sobre todo cuando los scripts tienen bajo acoplamiento), ya que son cargados en paralelo permitiendo además que el proceso de carga del resto de la página continúe de manera independiente. Dentro de los script loader asíncronos encontramos RequireJS, curl.js, BDLoad, $script.js, Steal.js, LABjs y yepnope.js. [13]
AMD justamente tiene como función definir módulos donde tanto el módulo como las dependencias del mismo puedan ser cargados de manera asíncrona[11], por lo que necesita un script loader asíncrono (para los ejemplos que se verán más adelante se utilizó RequireJS).

Esta API tiene dos conceptos fundamentales, la idea de un método *define* que se encarga de la definición del módulo y la idea de un método *require* que se encarga de manejar las dependencias del módulo.
El primer método está constituido por un identificador de módulo que puede ser opcional, en cuyo caso el módulo seria anónimo (lo que puede evitar colisiones con otros módulos de terceras partes), un arreglo de dependencias que son requeridas por el módulo para su funcionamiento y una función que se utiliza para definir el módulo (gráfico 3) [11].

```
define(
 module_id /*optional*/,
 [dependencies] /*optional*/,
 definition function /*function for
instantiating the module or object*/
);
```

Gráfico 3. Declaración del método define en AMD.

El siguiente ejemplo muestra cómo mediante el método *define* se pueden definir los módulos *Book* y *BookManager*. El primero tiene los atributos *title* y *content* con sus respectivos métodos de obtención y modificación (*setters y getters*) emulando el paradigma orientado a objetos, mientras que el segundo expone funciones para modificar el estado de los ejemplares del módulo *Book* y por ende este último aparece en su arreglo de dependencias, gráficos 4 y 5.

```
define("Book",function(){
var title;
var content;
return {
 setTitle:function(bookTitle){
 title=bookTitle;
 },
 getTitle:function(){
 return title;
 },
 setContent:function(bookCont){
 content=bookCont;
 },
 getContent:function(){
 return content;
 }
}
});
```

Gráfico 4. Definición del módulo *Book* en AMD.

```
define("BookManager",['Book'],
function(bookManaged){
 return{
 read:function(){

console.log(bookManaged.getTitle()+"
 "+bookManaged.getContent());
 },
 write:function(title,text){
  bookManaged.setTitle(title);
 bookManaged.setContent(text);
 }
 };
});
```

Gráfico 5. Definición del módulo *BookManager* en AMD.

El método *require* es utilizado para cargar código en un archivo JavaScript de más alto nivel (que se comunique con la interfaz de usuario) o para cargar de manera dinámica las dependencias de un módulo. En el ejemplo es posible observar que el método *require* está haciendo uso de una ejemplificación de *BookManager* para escribir y luego leer el contenido del ejemplar del módulo *Book* (gráfico 6). Por medio de la función *write* se modifica el estado de dicho ejemplar pasando como parámetros el título y el contenido del mismo, que en este caso en un fragmento de la reconocida novela *The Shining* escrita por Stepen King [14]. Luego se utiliza la función *read* para mostrar en consola el título y contenido del ejemplar (figura 6).

```
require(['BookManager'],function(bookManager
){
 bookManager.write("The Shining"," Jack
 Torrance thought: Officious little prick");
 bookManager.read();
});
```

Gráfico 6. Definición del método require para *BookManager* en AMD. Fuente propia

El método *require* está contenido dentro de un archivo JavaScript de nombre *App.js* que es cargado junto con todas sus dependencias mediante RequireJS, a través de una etiqueta <Script> que tiene un atributo especial llamado *data-main,* donde es necesario especificar la ruta de dicho archivo (gráfico 7). El diagrama de secuencia en la figura 7 muestra esta inyección de dependencias y el paso de mensajes entre el script App.js y los módulos definidos en AMD.

```
<script data-main="js/app"
src="js/require.js"></script>
```

Gráfico 7. Declaración del archivo *app.js* mediante la etiqueta <script>. Fuente propia

Resultado en consola de ejecutar la función *read* del módulo *BookManager* definido en AMD

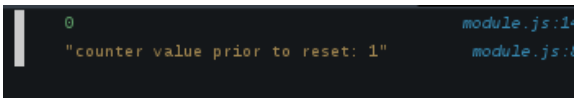

Figura 6. Resultado en consola de ejecutar la función 'read' del módulo 'BookManager' definido en AMD.

Diagrama de secuencia del sistema con los módulos definidos en AMD

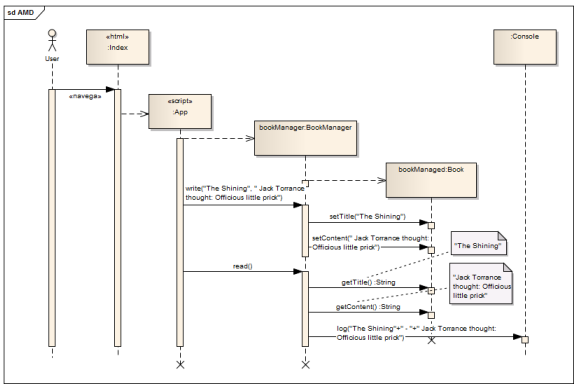

Figura 7. Diagrama de secuencia del sistema con los módulos definidos en AMD.

AMD presenta una forma clara, flexible y encapsulada de definir módulos y dependencias en JavaScript sin necesidad de etiquetas <script> (salvo la etiqueta del script principal), no presenta problemas de cross-domain, se pueden incluir varios módulos en un solo archivo, es posible realizar carga *perezosa* de scripts si se necesita y la mayoría de script loaders soportan la carga de módulos en navegadores sin un proceso de compilación. Además puede integrarse con frameworks como Backbone.js y ember.js para mantener las aplicaciones ordenadas y se ha utilizado en grandes compañías como IBM Y BBC iPlayer en proyectos no triviales [11]

C). CommonJS, módulos optimizados para servidor

CommonJS es un API de definición de módulos del lado del servidor, al contrario de AMD que está más enfocado a la definición de módulos del lado del cliente (esto no significa que no se puedan utilizar en sentido contrario, solo que CommonJS tiene algunas funciones especiales que no funcionarían en el navegador, como el manejo de archivos, *promises,* entre otros). Tiene varias implementaciones como Node.js y surgió gracias a un grupo de trabajo encargado del diseño, prototipado y estandarización de las APIs de JavaScript [11].

Un módulo, desde el punto de vista de CommonJS, es una pieza reutilizable de código que exporta objetos para que sean utilizados por otros módulos mediante la variable *exports*. Además cuenta con el método *require* que importa los objetos exportados por otros módulos [11]. En el ejemplo se definió un módulo llamado *dey* que tiene dos funciones *sayHello* y *sayBye* pero solo exporta la primera (dado que en JavaScript las funciones son objetos). Esta función es importada por el script 'exeDep.js por medio del método *require* que recibe como parámetro la ruta del módulo (gráfico 8 y 9).

```
function sayHello (name){
 console.log('Hello, my name is '+name+'
!!!');}
function sayBye(){
 console.log('Bye, bye !!!');
}
exports.sayHello=sayHello;
```

Gráfico 8. Definición del módulo '*Dep* en CommonJS.

```
var dep=require("./dep");
dep.sayHello('Daniel');
```

Gráfico 9. Script exeDep.js.

La función *sayHello* recibe en tiempo de ejecución el String Daniel y lo asigna a la variable *name* en el String *'Hello, my name is '+name+' !!!'* para después mostrarlo en consola (figura 8).

Resultado en consola de ejecutar la función *sayHello* del módulo *dep* definido en CommonJS

```
linux-l2ly:/home/daespuor # node Scripts/js/CommonJS/exeDep
Hello, my name is Daniel !!!
```

Figura 8. Resultado en consola de ejecutar la función 'sayHello' del módulo 'dep' definido en CommonJS.

La simplicidad de CommonJS es evidente, porque solo es necesario utilizar el método *require* para solicitar contenido de manera síncrona. También es posible apreciar la ausencia de una función *envoltura* o *wrapper* como en AMD, lo que resta complejidad a la definición de módulos, pero permite la declaración de variables globales. Para evitar esto RequireJS provee una forma para crear *wrappers* para los módulos hechos en CommonJS que además puedan ser declarados mediante etiquetas <Script> como si de un módulo AMD se tratase.

AMD y CommonJS podrían ser herramientas complementarias y son muy similares en su estructura (AMD inicio como una extensión de CommonJS). Sin embargo AMD presenta ciertas ventajas como el soporte de pluguins, poder cargar más que archivos JavaScript, tener facilidades para trabajar tanto en el navegador como en el servidor y haber sido adoptado por librerías JavaScript como Dojo, Mootools y Jquery [15].

D). Servicios web

En este mundo globalizado la integración entre empresas es algo común, sin embargo esto significa también una integración entre sistemas de software que siempre ha sido una labor dispendiosa y problemática, máxime cuando dichos sistemas están en diferentes plataformas, hardware y lenguajes de programación [16]. Con el objetivo de hacer menos traumático este proceso y de generar arquitecturas de software flexibles con capacidad de adaptación, se generó la arquitectura orientada a servicios (SOA) [17].

Este estilo arquitectónico concibe los sistemas de software como conjuntos de componentes llamados *servicios*, que se interrelacionan por medio de interfaces para cumplir con el objetivo del sistema, con una característica extra y es que los *servicios* pueden integrarse sin importar en que tecnología estén desarrollados (plataforma, hardware, lenguaje de programación, etc), lo que significa desacoplar la funcionalidad de un componente de su implementación. Además SOA aprovecha la amplia aceptación de XML y JSON para la especificación de las interfaces mediante Servicios Web (no es la única pero sí la más común) [17].

Existen varios modelos para servicios web, pero los más utilizados son SOAP y REST. SOAP (Simple Object Access Protocol) es el modelo estándar y está fuertemente basado en XML y XML *Schema*, donde se especifican las operaciones que el *servicio* provee, los parámetros de entrada tipados (entero, cadena, flotante e incluso tipos complejos) de dichas operaciones y su respectiva respuesta. Todo esto se codifica en un WSDL (Web Service Definition Lenguage) que por analogía sería el contrato entre el proveedor del servicio y el consumidor. Un servicio REST (Representational State Transfer) por su parte hace uso del protocolo HTTP y utiliza los verbos que este ofrece para representar las operaciones proveídas; los verbos más comunes son GET, PUT, POST y DELETE, además los datos pueden estructurarse tanto XML como en YAML o JSON entre otros (Aunque JSON es el preferido). De igual manera REST también presenta su propio esquema o contrato llamado WADL (Web Application Description Lenguage) definido en XML donde se describen las acciones disponibles y algunos metadatos (es mucho más flexible que el WSDL) (SmartBear).

En JavaScript es posible definir servicios web tipo REST mediante Node.js y su *framework* Express; sin embargo existe un módulo llamado Restify que es más apropiado para la realización de esta tarea, porque como está señalado en su página oficial (Restify Documentation), “Restify existe para dejarte construir APIs 'estrictas' de servicios que sean mantenibles y observables”, en cambio Express tiene como objetivo la creación de aplicaciones Web robustas. A continuación se puede observar una parte de la creación de un Servicio Web llamado *BookStore* con el módulo Restify , la creación del servidor HTTP que contendrá el servicio y la declaración de operaciones asociadas a verbos HTTP que en este caso son funciones de búsqueda e inserción de libros (gráfico 10 y 11).

```
var restify= require('restify');
var PATH='/BookStore';
var server=restify.createServer({
 name: "myBookStore"
});
server.listen('8080' ,'127.0.0.1',
function(){
console.log('%s listening at %s ',
server.name ,
server.url); });
```

Gráfico 10. Creación del Servicio Web y del servidor HTTP.

```
server.get({path : PATH , version :
'0.0.1'} ,
findAllBooks);
server.get({path : PATH +'/:isbn' ,
version :
'0.0.1'} , findBook);
server.post({path : PATH , version:
'0.0.1'}
,postNewBook);
```

Gráfico 11. Definición de operaciones asociadas a verbos HTTP.

En Restify también se pueden añadir pluguins para personalizar el servicio por ejemplo para parsear las peticiones HTTP, acceder a recursos en otros dominios, transformar los datos de la petición en un objeto JavaScript en el servidor automáticamente, etc.

Una vez declarado el servicio es posible acceder a este mediante una URL y el conjunto de verbos HTTP (Para este caso GET y POST). En el ejemplo para obtener todos los libros del *BookStore* se debe realizar una petición GET al servidor, para obtener

solo uno específicamente debe realizarse una petición GET con el parámetro ISBN (Número de identificación del libro) en la URL y finalmente para añadir un nuevo libro es necesario hacer una petición POST. Los resultados de dichas peticiones se pueden observar en la figura 9, la figura 10 y la figura 11. Para generar el código JavaScript que consuma el servicio existen muchas opciones, es posible utilizar por ejemplo Restify, SoapUI o un módulo de Node.js llamado wadl-client que genera código JavaScript a partir de una especificación WADL.

Petición GET al servicio *BookStore*

```
linux-l2ly:/home/daespuor # curl -is http://127.0.0.1:8080/BookStore
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/json
Content-Length: 168
Date: Tue, 19 May 2015 05:54:43 GMT
Connection: keep-alive

[{"isbn":"001","title":"The Shinning","author":"Stepen King"},{"isbn":"
le":"Do has","author":"Anonimo uno"},{"isbn":"003","title":"Do","author
```

Figura 9. Petición GET al servicio ´BookStore´.

Petición POST al servicio *BookStore*

```
linux-l2ly:/home/daespuor # curl -i -X POST -H "Content-Type: applicati
-d '{"isbn":"004","title":"Do has miss","author":"Anonimo tres"}' http:
.1:8080/BookStore
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/json
Content-Length: 60
Date: Tue, 19 May 2015 06:01:48 GMT
Connection: keep-alive
```

Figura 10. Petición POST al servicio ´BookStore´.

Petición GET con parámetro al servicio *BookStore*

```
linux-l2ly:/home/daespuor # curl -is http://127.0.0.1:8080/BookS
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/json
Content-Length: 60
Date: Tue, 19 May 2015 06:03:05 GMT
Connection: keep-alive

{"isbn":"004","title":"Do has miss","author":"Anonimo tres"}
```

Figura 11. Petición GET con parámetro al servicio ´BookStore´.

El principal problema de ambos modelos está en su especificación. El WSDL de SOAP es demasiado rígido, si hay una modificación (aunque sea solo un parámetro) en alguna operación del servicio, la especificación debe modificarse y esto implica una modificación en la implementación de los consumidores de ese servicio, mientras que el WADL de REST es opcional, lo que le da flexibilidad pero a cambio de no especificar las operaciones que ofrece el servicio. Para superar estas deficiencias existen aplicaciones para documentación como RAML y JSON-Home (SmartBear).

E). EcmaScript 6 (ES6), el futuro

EcmaScript es un "lenguaje de scripting" estandarizado por Ecma International e implementado por JavaScript, por ende la siguiente versión de dicho lenguaje (EcmaScript 6) es en esencia la siguiente versión de JavaScript. Esta ya tiene una sintaxis definida al menos en lo que a especificación de módulos se refiere, sin embargo aún no es soportada por los navegadores y se espera que siga siendo así al menos hasta junio del 2014 que es su fecha de lanzamiento inicial. Por esta razón para implementar algunas de las características de ES6 en ES5 (la versión actual), es necesario el uso de algunas herramientas extra por el momento[18].

Entre las características más llamativas de la nueva versión de JavaScript están el soporte para clases, utilización de funciones *arrow* y *promises*, *block scoping* y soporte para definición de módulos[19]. Esta última se incluyó con el fin de crear un formato que fuera fácil de utilizar tanto por usuarios de AMD como de CommonJS y por lo tanto tiene propiedades de ambas APIs. Es similar a AMD por que permite la carga y definición de módulos de manera asíncrona y es similar a CommonJS en su simplicidad y la inclinación por exportar un único objeto por módulo [20].

Los módulos definidos en ES6 pueden exportar recursos mediante el prefijo *export* e importar módulos por medio del prefijo *import* donde además debe especificarse que recursos se importaran y de que módulo [20]. En el ejemplo se define un módulo llamado *Adder* que exporta una función *double* que duplica el valor de un número. Dicha función es importada y utilizada por el script *'main.js* generando una respuesta en consola (gráfico 12 y 13).

```
var double = function(x) {
 return x + x;
}
export { double };
```

Gráfico 12. Definición del módulo '*Adder* en ES6. Fuente (Franklin, 24Ways to impress your friend, 2014).

```
import { double } from './adder';
console.log("El resultado de la operación es:
"+double(2)); //4
```

Gráfico 13. Script '*main.js*, (Franklin, 24Ways to impress your friend, 2014).

Esta sintaxis para la definición de módulos es la que

proveerá ES6 de manera definitiva. Sin embargo como se mencionó antes, no es una versión de JavaScript actualmente soportada por los navegadores. Para solucionar este inconveniente es necesario utilizar otros recursos para poder ejecutar módulos hechos en ES6 al menos por ahora. La primera opción es utilizar ES6 module-transpiler, que toma los módulos definidos en ES6 y los compila en AMD o CommonJS para que puedan ser interpretados por el navegador (una guía para la instalación del ES6 module-transpiler se encuentra en (Franklin, JavaScript Playground, 2014)) (gráfico 14 y 15). Para el ejemplo anterior el resultado de la compilación fue en CommonJS y se ejecutó por medio de Node.js, dando como resultado el que se puede ver en la figura 12.

```
"use strict";
var double = function(x) {
 return x + x;
}
exports["double"] = double;
//# sourceMappingURL=adder.js.map
```

Gráfico 14. Resultado en CommonJS de compilar el módulo '*Adder* con ES6 module-transpiler.

```
// 4
"use strict";
var adder$$ = require("./adder");
console.log("El resultado de la operación es: "
+adder$$["double"](2));
//# sourceMappingURL=app.js.map
```

Gráfico 15. Resultado en CommonJS de compilar el script '*main.js* con ES6 module-transpiler.
Resultado en consola de ejecutar la función *'double* por medio de ES6 module-transpiler

```
linux-l2ly:/home/daespuor # node compile
El resultado de la operación es: 4
```

Figura 12. Resultado en consola de ejecutar la función 'double' del módulo 'adder' por medio de ES6 module-transpiler.

La segunda opción es utilizar SystemJS que es un compilador para módulos universal, sin importar que estén definidos en ES6, AMD, CommonJS o cualquier otra forma. Este depende de dos librerías para cumplir su función ES6 module loader polyfill y Traceur. La primera se encarga de cargar archivos JavaScript y sus dependencias de manera automática por medio de la función *System.import* similar a como lo hace RequireJS. Traceur por su parte es un *transpiler* que compila de ES6 a ES5 lo que tiene como ventaja que puedan ser implementadas otras características (Además de los módulos) de la nueva versión de JavaScript que aún no son soportadas por el navegador [21].

Para su funcionamiento SystemJS debe ser cargado mediante las etiquetas <script>, luego es necesario declarar la función *System.import* (mediante las mismas etiquetas), que recibe como parámetro la ruta del archivo JavaScript principal que carga los módulos. En el ejemplo esta función carga el archivo *main.js* y las dependencias del mismo, en este caso el módulo *adder* (gráfico 16). El resultado en la consola del navegador puede verse en la figura 13 y la inyección de dependencias puede apreciarse en el diagrama de secuencia en la figura 14.

```
<script
src="js/System/system.js"></script>
<script>
 System.import('./js/ES6/main');
</script>
```

Gráfico 16. Declaración del archivo *'system.js* mediante la etiqueta <script> y carga del archivo *'main.js*.

Resultado en consola de ejecutar la función *double* por medio de SystemJS

```
"El resultado de la operación es: 4"   es6-module-…
```

Figura 13. Resultado en consola de ejecutar la función 'double' del módulo 'adder' por medio de SystemJS.

Diagrama de secuencia del sistema con el módulo definido en ES6

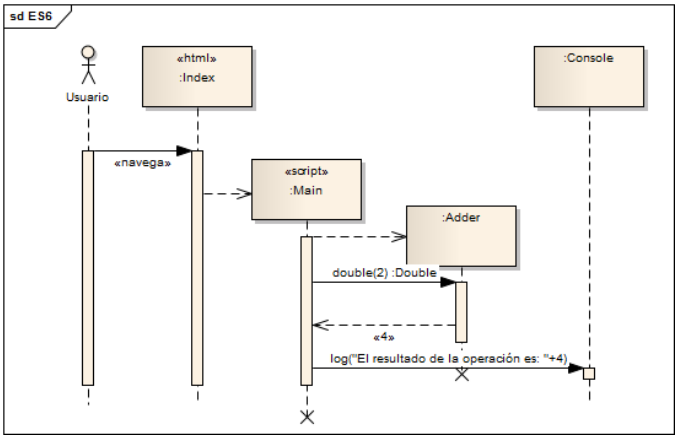

Figura 14. Diagrama de secuencia del sistema con el módulo definido en ES6.

Además en ES6 se puede definir exportar solo un recurso (objeto) por módulo similar a CommonJS, añadiendo el prefijo *default* al prefijo *export*. También es posible importar módulos de manera condicional mediante la *Programmatic Loader API* y se pueden definir dependencias cíclicas entre módulos [20].

A pesar de que hace falta un poco de trabajo extra para definir módulos en ES6 gracias a que ningún navegador actualmente lo soporta (Y por unos meses seguirá siendo así), no dista mucho del trabajo que significa utilizar RequireJS por ejemplo; además cuando esta versión de JavaScript sea finalmente soportada, los módulos hechos ahora en ES6 serán *migrados* automáticamente [18]. Ya no serán necesarias las herramientas intermediarias para su compilación e interpretación (ES6 module-transpiler o SystemJS) y se tendrá una solución nativa (mejor que el patrón módulo), lo que es sin duda es una ventaja frente a APIs como AMD y CommonJS.

Como se mencionó antes existen muchas formas de definir módulos que constituyan aplicaciones mantenibles y con capacidad de evolución en JavaScript, sin embargo las anteriormente especificadas son las más utilizadas y sobre las que existe mayor documentación. Sumado a esto, muestran de una manera clara que todo va encaminado hacia el establecimiento de un estándar para definir módulos reutilizables en JavaScript (exceptuando los servicios web que se incluyeron por la relevancia que está cobrando el estilo arquitectónico SOA en la actualidad), objetivo que según la promesa se podrá conseguir en ES6.

IV. Conclusiones

El concepto de componente o módulo ha venido cobrando trascendencia en los últimos años, gracias a que son elementos que pueden constituir aplicaciones o sistemas mantenibles, adaptables y con capacidad de evolucionar sin importar su complejidad. En JavaScript existen multitud de formas de crear aplicaciones con este tipo de características mediante APIs (AMD, CommonJS y todas las que surgieron a partir de CommonJS), servicios web y de manera nativa (patrón módulo, sus derivados y ES6), lo que posibilita que ahora en lugar de ser un lenguaje de programación utilizado para integrar componentes desarrollados en lenguajes de servidor, pueda ser utilizado para integrar componentes hechos enteramente en el mismo lenguaje, demostrando que la creación de aplicaciones robustas con él es una realidad.

Por otro lado, en los lenguajes de programación *tradicionales* para la creación de aplicaciones del lado del servidor (Java, .Net) existen modelos de componentes cuya función es estandarizar la implementación, nombramiento, personalización, composición, interoperabilidad, evolución y despliegue de componentes[23]. En JavaScript, por su parte, no existe aún una definición de módulos o componentes estándar, pero se espera que esto cambie cuando ES6 sea soportado por los navegadores. Lo que tiene como principal objetivo mediar entre las distintas formas de crear módulos (AMD, CommonJS, patrones, etc.) definiendo reglas comunes y acabando por ende con *El síndrome del salvaje oeste,* al menos en este aspecto.

Finalmente, aún en su versión ES6, JavaScript es permisivo en cuanto a que se pueden exportar más que operaciones entre módulos, es decir se pueden compartir variables u objetos con estado. En ese orden de ideas el estatus de componente estaría reflejado más por las buenas prácticas del desarrollador que por el mismo lenguaje.

IV.Referencias

[1] S. Cass, “IEEE spectrum”, 19 de julio de 2014, [en línea]. Consultado el 14 de abril de 2014, disponible en: http://spectrum.ieee.org/computing/software/top-10-programming-languages

[2] S. O'Grady, “RedMonk”, 14 de enero de 2014, [en línea]. Consultado el 14 de abril de 2014, disponible en: http://redmonk.com/sogrady/2014/01/14/language-rankings-1-15/

[3] Tiobe, “Tiobe Software, the software quality company”, abril de 2014, [en línea]. Consultado el 14 de abril de 2014, disponible en: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[4] C. Ghezzi, D. Mandrioli y M. Jazayeri, “Software Qualities and Principles”, *Computer Science Handbook*, Second Edition, Chapman & Hall/CRC,

pp. 2376-2401, 2004.

[5] K. Lau, A. Nordin, F. Taweel and T. Rana, "Constructing Component-based Systems Directly from Requirements using Incremental Composition", *Proc. 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2010.

[6] C. Larman, *Applying UML and patterns,* Second Edition, Prentice Hall, 2002.

[7] R. E. Noonan y . W. L. Bynum, "Scripting Lenguages", *Computer Science Handbook*, Second Edition, Chapman & Hall/CRC, pp. 2213-2230, 2004.

[8] A. Kanavin, "An overview of scripting languages", *Lappeenranta*, 2002.

[9] WebMonkey, "WebMonkey", febrero de 2010, [en línea]. Consultado el 27 de abril de 2014, disponible en: http://www.webmonkey.com/2010/02/scripting_language/

[10] P. Wayner, "InfoWorld", 17 de octubre de 2011, [en línea]. Consultado el 28 de abril de 2014, disponible en: http://www.infoworld.com/article/2620515/application-development/from-php-to-perl--what-s-hot--what-s-not-in-scripting-languages.html?page=2

[11] A. Osmani, "Learning JavaScript Design Patterns", *O'Reilly*, 2014.

[12] A. Castrounis, "innoarchitech.com", 24 de agosto de 2014, [en línea]. Consultado el 21 de mayo de 2014, disponible en: http://www.innoarchitech.com/scalable-maintainable-javascript-modules/

[13] J. Hann, "unscriptable.com", 30 de marzo de 2011, [en línea]. Consultado el 8 de agosto de 2014, disponible en: http://unscriptable.com/2011/03/30/curl-js-yet-another-amd-loader/

[14] S. King, *The Shining*, 1977.

[15] M. Medeiros, 30 de septiembre de 2011, [en línea]. Consultado el 10 de mayo de 2014, disponible en: http://blog.millermedeiros.com/amd-is-better-for-the-web-than-commonjs-modules/

[16] C. Kankanamge, *Web Service Testing with SOAPUI*, Birmingham, Pack Publishing, 2012.

[17] M. Bolo, "Aquitectura de integración orientada a servicios", *INTERFACES*, nº 1, pp. 19-46, 2006.

[18] J. Franklin, "24 Ways to impress your friend", 3 de diciembre de 2014, S.f., [en línea]. Consultado el 15 de mayo de 2014, disponible en: http://24ways.org/2014/javascript-modules-the-es6-way/

[19] N. Stieglitz, "Wintellect Blogs", 24 de marzo de 2014. S.f., [en línea]. Consultado el 15 de mayo de 2014, disponible en: http://www.wintellect.com/devcenter/nstieglitz/5-great-features-in-es6-harmony

[20] A. Rauschmayer, "2ality JavaScript and More", 7 de septiembre de 2014. [en línea]. Consultado el 15 de mayo de 2014, disponible en: http://www.2ality.com/2014/09/es6-modules-final.html

[21] J. Franklin, "JavaScript Playground", 8 de junio de 2014. [en línea]. Consultado el 15 de mayo de 2014, disponible en: http://javascriptplayground.com/blog/2014/06/es6-modules-today/

[22] SmartBear, "SoapUI", S.f., [en línea]. Consultado el 18 de mayo de 2014, disponible en: http://www.soapui.org/testing-dojo/world-of-api-testing/soap-vs--rest-challenges.html

[23] G. T. Heineman y W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley Longman Publish CO,* 2001.

[24] "Restify Documentation", S.f., [en línea]. Consultado el 18 de mayo de 2014, disponible en: http://mcavage.me/node-restify/