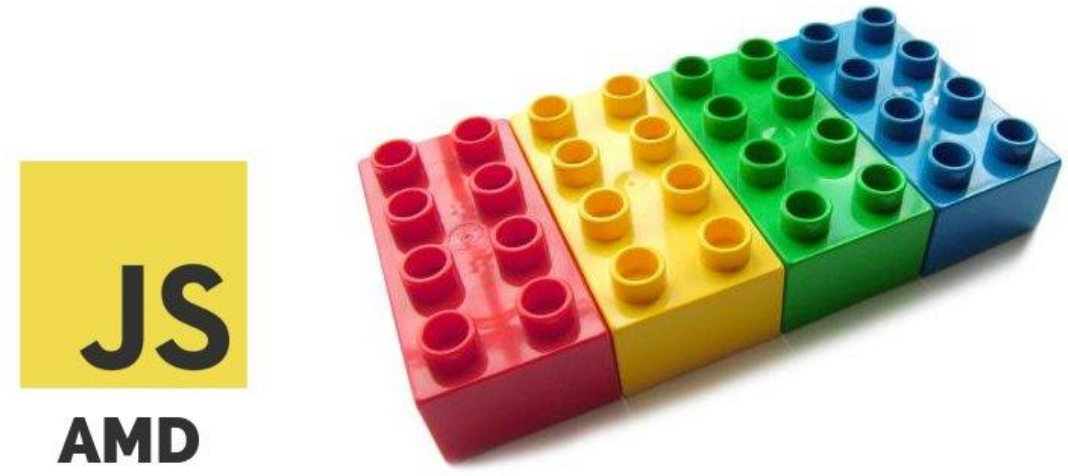
Creación de componentes JS utilizando módulos - AMD, CommonJS y UMD - Parte 1/3



Después de un tiempo la escritura de código Javascript, usted comienza a darse cuenta de que algunas cosas empiezan a repetirse, otros que hay que volver a utilizar, entonces piensas:

¿Cómo puedo modularizar, por lo que este componente se reutiliza en muchos proyectos diferentes?

Para responder a esta pregunta, entran en juego dos especificaciones del módulo: **AMD** y **CommonJS** (o **CJS**).

Ellos le permiten escribir su código de forma modular, siguiendo algunas definiciones. Vamos a hablar de cada uno de ellos.

AMD

Asíncrono Módulo Definición (AMD) se hizo conocido debido RequireJS . El uso de formato de AMD es el siguiente:

```
1  define( 'moduleName', [ 'jquery' ], function( $ ) {
2    return 'Hello World!';
3  });
```

Esa es la estructura básica de un módulo escrito en formato de AMD. La función de `los conjuntos` serán utilizados por cada nuevo módulo que cree.

Como el primer parámetro de esta función, puede mover el nombre del módulo - opcional - (a utilizar para que usted pueda "incluir" en otro lugar).

El segundo parámetro es una matriz de dependencias que módulo también opcional. En nuestro ejemplo, hemos puesto como dependencia de la biblioteca jQuery.
Y el tercer parámetro es la función de devolución de llamada que define el módulo y devuelve su contenido. Esta función es obligatoria. Ella será llamada tan pronto como todos arsenal dependencias en el segundo parámetro se descargan, y están listos para su uso.

Las dependencias normalmente devuelven valores. Para utilizar los valores devueltos, debe pasar los parámetros de la función de las variables que van a recibir los valores de las dependencias, siempre en el mismo orden que la matriz.

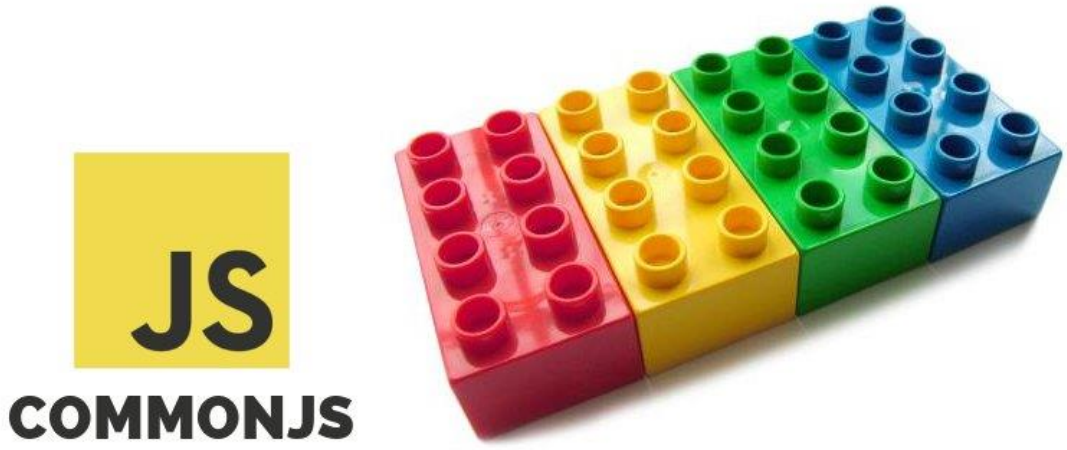
Por ejemplo, si en su módulo tendrá que usar jQuery y LoDash, se puede realizar la llamada de la siguiente manera:

```
define([ 'jquery', 'lodash' ], function( $, _ ) {
  // código de su módulo
});
```

Si no hay dependencias, la función ya se realiza pronto como se le llama:

```
define ( function ()  {
  // código de su módulo
 });
```

Creación de componentes JS utilizando módulos - AMD, CommonJS y UMD - Parte 2/3



CommonJS

También conocido cariñosamente como **CJS**, si ya escribiste algo de código en NodeJS estará familiarizado con el formato.

Utilizando el ejemplo dado cuando hablamos de AMD, podemos escribir bien, utilizando CJS:

```
var $ = require( 'jquery' );
function myModule() {
  // código de su módulo
}
module.exports = myModule;
```

En este formato, las cosas se ponen muy explícito. Las variables `$`, que están importando a través de `require` el módulo de `jQuery` . La función `myModule()` , que escribirá su módulo, que se exporta a través de `module.exports` , para que pueda ser importado en otro archivo usando `require` .

Y el módulo de `jQuery` , para ser importado con `require` , tendrá una estructura como esta (archivo `jquery.js`):

```
function jQuery() {
  // código de jQuery
}
module.exports = jQuery;
```

Fácil de entender, ¿no?

El **CommonJS** también cuenta con un objeto `exports`, que no es más que un alias para `module.exports`. Con el uso de `exports`, se puede exportar varios métodos por separado:

```
exports.method1 = function method1() {
  // Método 1 de su módulo
 };

exports.method2 = function method2() {
  // Método 2 de su módulo
 };
```

Lo que también se puede escribir así:

```
module.exports = function () {
  return {
    method1: function method1() {
      // Método 1 de su módulo
     },

    method2: function method2 () {
      // Método 2 de su módulo
     }
  }
};
```

Los dos retornan exactamente lo mismo. De hecho, en el segundo ejemplo es necesario ejecutar la función antes de poder utilizar los métodos, pero los dos son formatos estándar de objetos JavaScript.

Podemos utilizar **CommonJS** tanto en el servidor con NodeJS, por ejemplo, como en el navegador con browserify.

Bueno, hemos visto que, utilizando tanto **AMD** como **CommonJS**, es fácil de modularizar cualquier código JS. Pero pensando en un panorama más amplio: ¿cómo sé si el desarrollador que va a usar mi módulo está usando **AMD** o **CommonJS**? Voy a tener que proporcionar dos versiones?

Crear dos versiones del mismo código es casi una herejía! **No te repitas**!

Pero cual es la forma de resolver esto?

Creación de componentes JS utilizando módulos - AMD, CommonJS y UMD - Parte 3/3

Hemos visto cómo modularización de nuestro código usando AMD y CommonJS . Hoy vamos a ver a **el capitán Planeta** de los módulos: el **UMD** !

UMD

Universal Module Definition (UMD), es el que va a unir a los poderes de AMD y CommonJS en un solo componente! De hecho, él es el responsable de la comprobación de cuál de los dos formatos se está usando, así no duplicamos el código :)

El código utilizando UMD se verá algo como esto:

```
;(function(root, factory){
  if( typeof define === 'function' && define.amd ) {
    define(['jquery'], factory );
  }
  else if( typeof exports === 'object' ) {
    module.exports = factory( require( 'jquery' ) );
  }
  else {
    root.myModule = factory( root.jQuery );
  }
})(this, function( $ ) {
  return 'Hello World'!
});
```

Lo sé, el código es bastante feo, y a primera vista es difícil de entender. Pero voy a dividirla en partes para explicar mejor lo que hace, ¿de acuerdo? Vamos!

Básicamente, usted comenzará con un IIEF , o una función inmediata, que es una función auto-ejecutable:

```
; ( función ()  {

}) ();
```

Esta función recibe dos parámetros: el primero es `this` , que se define como `root` . Todo el mundo sabe cómo `this` en Javascript es controvertido, ya que varía su valor de acuerdo con el ámbito en el que se encuentra. En el caso que estamos llamando `this` en el ámbito global, por lo que si estoy en el cliente (navegador), será el objeto `window`. Qué si estoy en el servidor (usando Node, por ejemplo), será el objeto `global`.

El segundo parámetro, definido como `factory` , es la función que va a ejecutar el módulo. Sabemos que en Javascript, las funciones son objetos de primera clase. Ellas son tratadas como cualquier otro tipo de valor, entonces, también se pueden pasar como parámetro a otra función. El **UMD** se aprovecha de esto para hacer el negocio un poco más ilegible y difícil de entender. :P

Pero mirando el lado bueno, una vez que entienda esto, será fácil identificar cada parte: D

Luego, pasando los parámetros, se verá así:

```
;(function( root, factory ) {

})( this, function() {} );
```

Dentro de la función pasada como parámetro, va el código de su módulo:

```
;(function( root, factory ) {

})( this, function() {
  // código de su módulo
});
```

Ahora vamos a ver lo que sucede dentro de la función principal. Como **UMD** identifica que se está utilizando **AMD** o **CommonJS** .

El formato **AMD** tiene por defecto la función `define` y una propiedad `amd` definida en está. Esto es lo que verifica el primer `if` :

```
if( typeof define === 'function' && define.amd ) {
  // ...
}
```

Es decir, si hay un `define` y es una función, y la propiedad `amd` está definida para esta función, entonces el desarrollador está utilizando alguna lib en formato **AMD**. Sabiendo esto, se usa la función `define` para "definir" mi módulo, se pasan las dependencias en un array y se llama a la función que ejecuta el módulo:

```
if( typeof define === 'function' && define.amd ) {
  define([ 'jquery' ], factory );
}
```

¿Recuerdas cuando hablamos de **AMD**? Cada parámetro de la función del módulo representa una dependencia en el array, manteniendo un orden. Entonces la función que se pasa como parámetro (factory), debe recibir el parámetro para llamar a jQuery dentro de nuestro módulo, ya que es una dependencia:

```
;(function( root, factory ) {
  if( typeof define === 'function' && define.amd ) {
    define([ 'jquery' ], factory );
  }
})( this, function( $ ) {
  // Código del módulo
});
```

Resuelto el problema de **AMD** : D

Ahora, si el usuario no está utilizando AMD, vamos a ver si está usando **CommonJS** en la siguiente verificación:

```
else if( typeof exports === 'object' ) {
  // ...
}
```

Hemos visto que una de las cosas que define el formato **CommonJS** es que tiene un objeto `exports`. Así que eso es lo que vamos a verificar. Si `exports` existe, y es un objeto, exportamos nuestro módulo:

```
// ...
else if( typeof exports === 'object' ) {
  module.exports = factory( require( 'jquery' ) );
}
// ...
```

Cómo tenemos que pasar jQuery como parámetro a nuestro módulo, usamos `require`, porque hemos visto que es así como se incluye una dependencia utilizando este formato.

También resuelto los **CommonJS**!

Pero ¿y si el desarrollador quiere usar nuestro módulo, pero no está utilizando AMD, ni CommonJS?

En este caso, podemos dar un nombre a nuestro módulo, exportarlo en el ámbito global, usando `root`, que será el objeto `window`, si es en el navegador, o `global`, si se está utilizando Node. Pasamos también el objeto `jQuery`, que debe estar también en el ámbito global:

```
else {
  root.myModule = factory( root.jQuery );
}
```

Listo!!! : D

Hay algunas otras variaciones de **UMD** , que se pueden ver aquí .

*Así que la solución es utilizar **UMD** para todo lo que hago?*

No es así.

Para todo lo que es referente a su aplicación en específica, debe utilizar un único patrón: AMD o CommonJS, o algún *patrón propio*.

Ahora, cuando se quiere tiene un módulo genérico que desea volver a utilizar en cualquier ambiente y en cualquier proyecto, es el momento de utilizar **UMD**.
Sacado de:
http://blog.da2k.com.br/2015/01/03/como-criar-componentes-js-usando-modulos-amd-commonjs-e-umd-parte-1-3/

http://blog.da2k.com.br/2015/01/04/como-criar-componentes-js-usando-modulos-amd-commonjs-e-umd-parte-2-3/

http://blog.da2k.com.br/2015/01/05/como-criar-componentes-js-usando-modulos-amd-commonjs-e-umd-parte-3-3/