Desarrollo de Plugins

1. Qué son los plugins

Los plugins son la utilidad que pone jQuery a disposición de los desarrolladores para ampliar las funcionalidades del framework. Por lo general servirán para hacen cosas más complejas necesarias para resolver necesidades específicas, pero las hacen de manera que puedan utilizarse en el futuro en cualquier parte y por cualquier web.

En la práctica **un plugin no es más que una función que se añade al objeto jQuery** (objeto básico de este framework que devuelve la función jQuery para un selector dado), para que a partir de ese momento responda a nuevos métodos.

2. Cómo se crea un plugin de jQuery

Los plugins en jQuery se crean asignando una función a la propiedad "fn" del objeto jQuery. A partir de entonces, esas funciones asignadas se podrán utilizar en cualquier objeto jQuery, como uno de los muchos métodos que dispone dicho objeto principal del framework.

```
jQuery.fn.desaparece = function() {
   $(this).each(function(){
      $(this).css("display", "none");
   });
   return $(this);
};
```

```
jQuery.fn.desaparece = function() {
   return $(this).each(function(){
      $(this).css("display", "none");
   });
};
```

A modo de ejemplo, podemos ver a continuación un código fuente de un plugin muy sencillo:

Este plugin permitiría hacer desaparecer a los elementos de la página y podríamos invocarlo por ejemplo de la siguiente manera:

```
$("h1").desaparece();
```

Pero atención, porque tenemos que realizar el trabajo siguiendo una serie de normas, para asegurar que los plugins funcionen como deben y los pueda utilizar cualquier desarrollador en cualquier página web.

Aquí puedes ver un listado normas, que son sólo unas pocas, pero que resultan tremendamente importantes.

1. El archivo que crees con el código de tu plugin lo debes nombrar como **jquery.[nombre de tu plugin].js.** Por ejemplo jquery.desaparece.js.
2. **Añade las funciones como nuevos métodos por medio de la propiedad fn del objeto jQuery**, para que se conviertan en métodos del propio objeto jQuery.
3. Dentro de los métodos que añades como plugins, la palabra **"this"** será una referencia al objeto jQuery que recibe el método. Por tanto, podemos utilizar "this" para acceder a cualquier propiedad del elemento de la página con el estamos trabajando.

4. Debes colocar un punto y coma **";" al final de cada método** que crees como plugin, para que el código fuente se pueda comprimir y siga funcionando correctamente. Ese punto y coma debes colocarlo después de cerrar la llave del código de la función.
5. El método debe retornar el propio objeto jQuery sobre el que se solicitó la ejecución del plugin. Esto lo podemos conseguir con un **return this;** al final del código de la función. Ésto permite concatenar operaciones.
6. Se debe usar **this.each** para iterar sobre todo el conjunto de elementos que puede haber seleccionados. Recordemos que los plugins se invocan sobre objetos que se obtienen con selectores y la función jQuery, por lo que pueden haberse seleccionado varios elementos y no sólo uno. Así pues, con this.each podemos iterar sobre cada uno de esos elementos seleccionados. Esto es interesante para producir código limpio, que además será compatible con selectores que correspondan con varios elementos de la página.
7. **Asigna el plugin siempre al objeto jQuery**, en vez de hacerlo sobre el símbolo $, así los usuarios podrán usar alias personalizados para ese plugin a través del método noConfict(), descartando los problemas que puedan haber si dos plugin tienen el mismo nombre.

Estas reglas serán suficientes para plugins sencillos, aunque quizás en escenarios más complejos en adelante necesitaremos aplicar otras reglas para asegurarnos que todo funcione bien.

Ejemplo de un plugin en jQuery

El plugin que vamos a construir sirve para hacer que los elementos de la página parpadeen, esto es, que desaparezcan y vuelvan a aparecer en un breve instante. Es un ejemplo bien simple, que quizás tenga ya alguna utilidad práctica en tu sitio, para llamar la atención sobre uno o varios elementos de la página.

```
<html>
<head>
<title>Creando plugins en jQuery</title>
<style type="text/css">
   div{
      background-color: #ff9966;
      padding: 10px;
   }
</style>
<meta charset='utf-8'>
<script src="http://code.jquery.com/jquery-1.11.3.min.js"></script>
<script type="text/javascript" src="jQuery.parpadear.js"></script>
<script type="text/javascript">
$(document).ready(function(){
   //parpadean los elementos de class CSS "parpadear"
   $(".parpadear").parpadea();

   //añado un evento clic para un botón, para que al pulsarlo parpadeen los
elementos de clase parpadear
   $("#botonparpadear").click(function(){
      $(".parpadear").parpadea();
   })
})
</script>

</head>
<body>
   <p class="parpadear">Hola que tal, esto parpadeó gracias a jQuery!</p>
```

```
        <p>Párrafo normal que no va a parpadear.</p>
        <p class="parpadear">Sí parpadea</p>
        <p>Párrafo normal que no va a parpadear tampoco...</p>
        <div class="parpadear" >Esta capa también tiene la clase parpadear, con
lo que ya se sabe...</div>
        <p><input type="button" value="Parpadea de nuevo" id="botonparpadear"></p>
    </body>
    </html>
```

3. Gestión de opciones en plugins jQuery

Una de las tareas típicas que realizaremos es la creación de un sistema para cargar opciones con las que configurar el comportamiento de los plugins. Estas opciones las recibirá el plugin como parámetro cuando lo invocamos inicialmente. Nosotros, como desarrolladores del plugin, tendremos que definir cuáles van a ser esas opciones de configuración y qué valores tendrán por defecto.

La ayuda del sitio de jQuery para la creación de plugins sugiere la manera con la que realizar el proceso de configuración del plugin, por medio de un **objeto de "options"**, que nos facilitará bastante la vida.

Definir opciones por defecto en el código del plugin

Con el siguiente código podemos definir las variables de configuración por defecto de un plugin y combinarlas con las variables de options enviadas por parámetro al invocar el plugin.

```
jQuery.fn.miPlugin = function(cualquierCosa, opciones) {
  //Defino unas opciones por defecto
  var configuracion = {
     dato1: "lo que sea",
     dato2: 78
  }
  //extiendo las opciones por defecto con las recibidas
  jQuery.extend(configuracion, opciones);

  //resto del plugin
  //donde tenemos la variable configuración para personalizar el plugin
}
```

La función principal del plugin recibe dos parámetros, uno "cualquierCosa" y otro "opciones". El primero supongamos que es algo que necesita el plugin, pero la configuración, que es lo que nos importa ahora, se ha recibido en el parámetro "opciones".

Ya dentro de la función del plugin, se define el objeto con las opciones de configuración, con sus valores por defecto, en una variable llamada "configuracion".

En la siguente línea se mezclan los datos de las opciones de configuración por defecto y las recibidas por el plugin al inicializarse. Luego podremos acceder por medio de la variable "configuracion" todas las opciones del plugin que se va a iniciar.

Invocar al plugin enviando el objeto de opciones

Ahora podemos ver el código que utilizaríamos para invocar al plugin pasando las opciones que deseamos:

```
$("#elemento).miPlugin({
   dato1: "Hola amigos!",
   dato2: true
});
```

O podríamos enviar sólo alguno de los datos de configuración, para que el resto se tomen por defecto:

```
$("<div></div>").miPlugin({
   dato2: 2.05
});
```

O no enviar ningún dato al crear el plugin para utilizar los valores por defecto en todas las opciones de configuración.

```
$("p").miPlugin();
```

Ejemplo de plugin con parámetros de configuración

Las opciones que vamos a implementar serán las siguientes:

- Velocidad de la animación de mostrar y ocultar el tip
- Animación a utilizar para mostrar el tip
- Animación a utilizar para ocultar el tip
- Clase CSS para la capa del tip

Comenzamos por especificar, con notación de objeto, las opciones de configuración por defecto para el plugin:

```
var configuracion = {
   velocidad: 500,
   animacionMuestra: {width: "show"},
   animacionOculta: {opacity: "hide"},
   claseTip: "tip"
}
```

Ahora veamos el inicio del código del plugin, donde debemos observar que en la función que define el plugin se están recibiendo un par de parámetros. El primero es el texto del tip, el segundo son las opciones específicas para configurar el plugin.

```
jQuery.fn.creaTip = function(textoTip, opciones) {
   //opciones por defecto
   var configuracion = {
      velocidad: 500,
      animacionMuestra: {width: "show"},
      animacionOculta: {opacity: "hide"},
      claseTip: "tip"
   }
```

```
//extiendo las opciones por defecto con las opciones del parámetro.
 jQuery.extend(configuracion, opciones);
      this.each(function(){
      //código del plugin
   });
});
```

Esta sentencia es una llamada al método extend() que pertenece a jQuery. Esta función recibe cualquier número de parámetros, que son objetos, y mete las opciones de todos en el primero. Luego, después de la llamada a extend(), el objeto del primer parámetro tendrá sus propiedades más las propiedades del objeto del segundo parámetro. Si alguna de las opciones tenía el mismo nombre, al final el valor que prevalece es el que había en el segundo parámetro

Código completo:

```
jQuery.fn.creaTip = function(textoTip, opciones) {
   var configuracion = {
      velocidad: 500,
      animacionMuestra: {width: "show"},
      animacionOculta: {opacity: "hide"},
      claseTip: "tip"
   }
   jQuery.extend(configuracion, opciones);

   this.each(function(){
      elem = $(this);
      var miTip = $('<div class="' + configuracion.claseTip + '">' + textoTip +
'</div>');
      $(document.body).append(miTip);

      elem.mouseenter(function(e){
         miTip.css({
            left: e.pageX + 10,
            top: e.pageY + 5
         });
         miTip.animate(configuracion.animacionMuestra, configuracion.velocidad);
      });
      elem.mouseleave(function(e){
         miTip.animate(configuracion.animacionOculta, configuracion.velocidad);
      });
   });

   return this;
};
```

Modos de invocación

Modo 1

```
$("#elemento1").creaTip("todo bien...");
```

Modo 2

```
$("#elemento2").creaTip("Otra prueba...", {
   velocidad: 1000,
   claseTip: "otroestilotip",
   animacionMuestra: {
```

```
      opacity: "show",
      padding: '25px',
      'font-size': '2em'
   },
   animacionOculta: {
      height: "hide",
      padding: '5px',
      'font-size': '1em'
   }
});
```

Para facilitar la personalización del plugin a los usuarios, las opciones por defecto se deben indicar de la siguiente forma:

```
//fichero plugin
$.fn.tipOpPublicas.defaults = {
         velocidad: 500,
         animacionMuestra: {width: "show"},
         animacionOculta: {opacity: "hide"},
         claseTip: "tip"
};

//fichero desde el que se trabaja con el plugin
$.fn.tipOpPublicas.defaults.velocidad = 'fast';
```

4. Alias personalizado y ocultar código en plugins jQuery

Se trata de ocultar de una manera sencilla todo el código de nuestros plugins y utilizar un alias para la variable $ que puede dar conflictos con otras librerías. Algo que nos ayudará de dos maneras:

El símbolo $ se utiliza en muchos otros frameworks y componentes Javascript y si el web donde se coloque el plugin utiliza alguno de ellos, pueden ocurrir conflictos, algo que no ocurrirá en el caso que utilicemos un alias.

En el código de los plugins puedes utilizar tus propias variables o funciones, que tendrán el nombre que hayas querido. Pero alguno de esos nombres puede que ya los utilicen otros programadores en sus páginas, lo que puede generar conflictos también. Por eso no es mala idea ocultar tu código y hacerlo local a un ámbito propio.

Todo esto se consigue **colocando todo tu código dentro de una función que se invoca según se declara.**

```
(function() {
   //Código de tu plugin

   //puedes crear tus variables o funciones
   //sólo serán visibles aquí
   var loquesea;
   function algo(){

   }
})(); //la función se ejecuta instantáneamente
```

Alias

```
(function($) {
   //código del plugin
})(jQuery);
```

Como la variable jQuery siempre es una referencia al framework correcta, puedes estar seguro que no tendrá conflictos con otras librerías. Luego la recibimos con el nombre $, pero en ese caso ya estamos en el ámbito de la función, donde las variables locales pueden tener el nombre que nosotros queramos.

5. Desarrollar plugins con más de una funcionalidad

Para definir funciones internas de nuestro plugin jQuery recomienda que se haga mediante cadenas de texto.

```
 (function( $ ){
       var methods = {
              init : function( options ) {
                //código
              },
              show : function( ) {
                //código
              },
              hide : function( ) {
                //código
              },
              update : function( content ) {
                //código
              }
       };

   $.fn.tooltip = function( method ) {
     // Si existe la función la llamamos
        if ( methods[method] ) {
          return methods[method].apply( this, Array.prototype.slice.call
        ( arguments, 1 ));
        } else if ( typeof method === 'object' || ! method ) {
          //Si no se pasa ningún parámetro o el parámetro es
          //un objeto de configuración llamamos al inicializador
          return methods.init.apply( this, arguments );
        } else {
          //En el resto de los casos mostramos un error
             alert('La función ' +  method + ' no existe en jQuery.tooltip' );
        }
     };
  })( jQuery );
```

De esta forma las llamadas al plugin serían:

```
//Inicializar el plugin
$('div').tooltip();
//Inicializarlo con parámetros
$('div').tooltip({
  velocidad : 200
});
//Llamar a algunas de sus funciones
$('div').tooltip('hide');
$('div').tooltip('update', 'Nuevo contenido');
```

6. Desarrollar plugins que dependen directamente de jQuery

Generalmente hacemos los plugin a través de la propiedad "fn" del objeto jQuery. Esto es importante mencionarlo para que se vean las diferencias con el otro método que vamos a explicar ahora.

```
jQuery.fn.miPlugin = function(){
   //código del plugin
}
```

De este modo, podrás fácilmente usar tus propios plugins a través de cualquier objeto jQuery creado a través de un selector.

```
$("p").miPlugin();
```

El método extend es un método del "core" de jQuery. Sirve para fusionar dos, o más, objetos en uno solo. Pues bien, este método es susceptible de asignarle más información a jQuery en sí, a $.

- Si a extend le envío como parámetro dos o más objetos, extiende las propiedades del objeto del primer parámetro con las propiedades (o métodos) del objeto del segundo parámetro o los siguientes.

  `$.extend(obj1, obj2);` //ahora obj1 tendrá también las propiedades y métodos de obj2

- Si a extend sólo le entregamos un parámetro, entonces extiende el objeto jQuery con las propiedades y métodos del objeto que enviemos en el único parámetro.

  `$.extend(obj1)` //ahora jQuery está conteniendo las propiedades y métodos de obj1

Luego la forma de crear un plugin que dependa directamente de JQuery es:

```
$.extend({
   pluginNuevo: function() {
      //código del plugin
   }
});
```

Esto técnicamente agrega la propiedad "pluginNuevo" al objeto jQuery. (Es cierto que pluginNuevo es un método, pero en realidad en Javascript no deja de ser una propiedad de un objeto en la que hemos asignado una función). Dicho de otra manera, estás asignando nuevos elementos al namespace de jQuery.

En la práctica, la diferencia de este plugin con uno de los que hacíamos antes, es que lo invocas directamente a través de la jQuery.

```
$.pluginNuevo();
```

Si quisieras desarrollar un plugin para trabajar con las cookies ¿qué sería más lógico?

```
$("body").revisaCookie();
```

ó

```
$.revisaCookie();//ok
```