jQuery y plugins (Parte II)

AJAX

Competencias

- Crear un script que realice peticiones asíncronas utilizando la librería JQuery y AJAX para resolver un problema planteado.
- Entender los conceptos de API REST para generar consultas a servidores que respondan de una forma estándar y legible.
- Entender el formato JSON como uno de los protocolos más usados de construcción de objetos devueltos por API REST.

Introducción

¡Hablemos con los servidores!. Ahora en este punto, comenzaremos a trabajar con la integración de una aplicación desde el cliente hacia el servidor y viceversa. Una de las cosas que han llevado a la web tal y como la conocemos es la capacidad de comunicar sitios con bases de datos y servidores en todo el mundo. Una página estática es el primer nivel para un desarrollador Javascript. Ahora, hablaremos de dinamismo en la web y como podemos transformar una página web estática en un **cliente** dinámico en comunicación con algún **servidor**, en donde, en base a la información que capturemos de dicho servidor, podemos renderizar y generar contenido en nuestro sitio. Generando una aplicación funcional, tal y como se trabaja en el mercado.

Para llevar a cabo este nuevo nivel de dinamismo es donde hablaremos de un nuevo concepto que está muy bien integrado con JQuery. AJAX.

¿Qué es AJAX?

AJAX (Asynchronous JavaScript and XML) se refiere a un grupo de tecnologías que se utilizan para desarrollar aplicaciones web. Al combinar estas tecnologías, las páginas web parece que son más receptivas puesto que los paquetes pequeños de datos se intercambian con el servidor y las páginas web no se vuelven a cargar cada vez que un usuario realiza un cambio de entrada. Ajax permite que un usuario de la aplicación web interactúe con una página web sin la interrupción que implica volver a cargar la página web. La interacción del sitio web ocurre rápidamente sólo con partes de la página de recarga y renovación.

AJAX se compone de las siguientes tecnologías:

- XHTML y CSS para presentar información.
- DOM (Document Object Model - modelo de objetos de documento) para visualizar e interactuar de forma dinámica la información presentada.
- El objeto XMLHttpRequest para manipular los datos de forma asíncrona con el servidor web.
- XML, HTML y XSLT para el intercambio y la manipulación de datos.
- Se visualiza JavaScript para enlazar solicitudes e información de datos.

AJAX incorpora estas tecnologías para crear un nuevo enfoque al desarrollo de aplicaciones web.

AJAX define un método de iniciar un cliente con la comunicación del servidor sin recargas de páginas. Proporciona una forma de permitir actualizaciones de página parciales. Desde una perspectiva de usuario de página web, significa que la mejora de la interacción con una aplicación web, que proporciona al usuario más control de su entorno, es similar a la de una aplicación de escritorio.

En una aplicación web tradicional, las solicitudes HTTP, que se inician mediante la interacción del usuario con la interfaz web, se realizan a un servidor web. El servidor web procesa la solicitud y devuelve una página HTML al cliente. Durante el transporte HTTP, el usuario no puede interactuar con la aplicación web.

Como todo lo explicado anteriormente no es oro, existen algunas limitaciones. Para que una petición AJAX funcione, el navegador debe entender e interpretar Javascript. En la actualidad, el trabajar con navegadores tan antiguos que no conozcan Javascript es un caso de borde bastante dificil. Sin embargo este "pero" se debe tener en consideración.

Introducción al Concepto de API

Una interfaz de programación de aplicaciones (API) es un conjunto de herramientas, definiciones y protocolos que se usa para diseñar e integrar software de aplicaciones. Permite que un producto o servicio se comunique con otros productos y servicios, sin la necesidad de saber cómo se implementan. Las API simplifican el desarrollo de las aplicaciones, lo cual permite ahorrar tiempo a los desarrolladores y generar un estándar o protocolo legible para futuros programadores que aborden dicho proyecto. Cuando se diseñan herramientas y productos nuevos, las API otorgan flexibilidad; simplifican el diseño, la administración y el uso; y proporcionan oportunidades para la innovación.

Debido a que simplifican la forma en que los desarrolladores integran los elementos de las aplicaciones nuevas en una arquitectura actual. Las necesidades comerciales suelen cambiar rápidamente en respuesta a los mercados digitales en constante cambio, donde la competencia puede modificar un sector entero con una aplicación nueva. Para seguir siendo competitivos, es importante admitir la implementación y el desarrollo rápidos de servicios innovadores. El desarrollo de aplicaciones nativas de la nube es una forma identificable de aumentar la velocidad de desarrollo y se basa en la conexión de una arquitectura de aplicaciones de microservicios a través de las API.

Las API son un medio simplificado para conectar tu proyecto a través del desarrollo de aplicaciones nativas de la nube, pero también le permiten compartir sus datos con clientes y otros usuarios externos. Las API públicas representan un valor comercial único porque simplifican y amplían la forma en que se conecta con sus partners y, además, pueden rentabilizar sus datos. Un ejemplo conocido es la API de Google Maps o de servicios de Sharer de facebook, incluso integraciones con servicios más pequeños que nos permiten obtener valores como el clima, provincias, ciudades, estados de un país o información de carácter global. Las API públicas son escenciales para el desarrollo de código abierto.

Por ejemplo, imagina una empresa distribuidora de libros. La distribuidora de libros podría dar a sus clientes una aplicación que les permita a los empleados de la librería verificar la disponibilidad de los libros con el distribuidor. El desarrollo de esta aplicación podría ser costoso, estar limitado por la plataforma y requerir mucho tiempo de desarrollo y mantenimiento continuo.

De forma opcional, la distribuidora de libros podría proporcionar una API para verificar la disponibilidad en inventario. Existen varios beneficios para este enfoque:

- Permitir a los clientes el acceso a los datos con una API que les ayude a añadir información sobre su inventario en un solo lugar.
- La distribuidora de libros podría realizar cambios en sus sistemas internos sin impactar en los clientes, siempre que el comportamiento de la API no cambie.

Con una API disponible de forma pública, los desarrolladores que trabajan para la distribuidora de libros, vendedores o terceros podrían desarrollar una aplicación para ayudar a los clientes a encontrar los libros que buscan. Esto podría dar como resultado mayores ventas u otras oportunidades.

En resumen, las API le permiten habilitar el acceso a sus recursos y, al mismo tiempo, mantener la seguridad y el control. Cómo habilitar el acceso y a quiénes depende de ti. La seguridad de las API tiene que ver con una buena gestión de ellas. Para conectarse a las API y crear aplicaciones que utilicen los datos o las funciones que estas ofrecen, se puede utilizar una plataforma de integración distribuida que conecte todos los elementos, incluidos los sistemas heredados y el Internet de las cosas (IoT).

Transferencia de Estado Representacional (REST)

Luego de una introducción teórica al concepto de API, cabe preguntarnos realmente como podemos comunicarnos con alguna de éstas de una forma estándar en donde, da igual la aplicación que consulta por cierta información a un servidor, el resultado mediante un protocolo establecido siempre será el mismo. Así es como nace REST.

REST, es un estilo arquitectónico diseñado para proporcionar estándares entre los sistemas informáticos en la web, lo que facilita la comunicación entre éstos. Los sistemas compatibles con REST, a menudo llamados sistemas RESTful, se caracterizan por ser *stateless*, es decir, separa la lógica del cliente y el servidor.

Un servidor que opera bajo el paradigma REST, no importa que tecnología utilice, solo importa definir de forma correcta los puntos de consulta **(endpoints)** que quedarán expuestos o abiertos para que otro sistema, a través de una **url** pueda hacer una petición HTTP para obtener información.

Peticiones (Request)

REST permite que un cliente pueda hacer peticiones HTTP a un servidor, este puede ser remoto, público, privado. Normalmente una petición es una función que gatilla una acción la cual puede recolectar datos o modificar datos de una base de datos remota alojada en dicho servidor.

Una petición normalmente consta con:

- Una acción HTTP. (Más adelante se detallarán acciones existentes).
- Una cabecera (header) que indica por lo general información que pueda necesitar el servidor para decodificar o interpretar la petición. Los headers por lo general también almacenan información que pueden otorgar seguridad al momento de hacer peticiones a servidores privados.
- Una ruta (PATH) en donde se hace la consulta
- De forma opcional y depende del tipo de petición, puede contener opciones adicionales como un cuerpo (BODY) que puede contener cierto tipo de información.

Acciones HTTP

Existen 4 acciones básicas HTTP que son utilizadas para interactuar con servidores.

- GET -> Busca contenido específico o una colección de datos.
- POST -> crea o genera un nuevo recurso en un servidor.
- PUT -> modifica un recuerso en específico.
- DELETE -> Elimina un recuerso en específico.

Estos conceptos se basan en la operación básica de orientación a objetos llamado CRUD. Recomendamos al estudiante investigar sobre estos términos a modo de complemento con esta lectura.

Cabeceras (headers)

Como se explicó anteriormente, las cabeceras permiten entregarle al servidor información adicional sobre como se espera recibir o enviar cierta información.

Por ejemplo:

```
GET /articles/23
Accept: text/html, application/json
```

Genera una acción de tipo get y en su cabecera se determina que dicha petición acepta como respuesta texto en formato de HTML o bien un objeto JSON.

La cantidad de caberas o condiciones a la petición son extensas, recomendamos leer la documentación oficial de Mozilla que indica la gama de posibilidades y condiciones que se pueden exponer en una cabecera.

Rutas (PATH)

La ruta es uno de los elementos más importantes a la hora de hacer una petición HTTP. Es la dirección física en donde se indica qué consultar y a dónde hacerlo.

entendamos el concepto con un ejemplo:

`http://fashionboutique.com/customers/223/orders/12`

Si trozamos esta ruta, podemos observar varios elementos.

- Un protocolo HTTP por el cual se hace la consulta.
- Una ruta base (BASE_URL) en este caso sería `fashionboutique.com`. Esta es la ruta base, la raíz a donde las peticiones irán en todo momento.
- Una ruta relativa (RELATIVE PATH) `customers/223/orders/12` que indica la acción, busca en `customers` con un `id=223` sus `orders` específicamente la que posea `id=12`.

La ruta del ejemplo se puede construir de forma genérica de la siguiente forma:

`http://fashionboutique.com/customers/:customer_id/orders/:order_id`

En donde las variables `:customer_id` y `:order_id` pueden ser variables según lo que requiera la consulta.

¿Qué pasaría si la consulta es la siguiente?

`http://fashionboutique.com/customers/:customer_id/orders`

En este caso, obtendríamos todas las `orders` de un usario en particular.

Por otra parte si hacemos la siguiente consulta:

`http://fashionboutique.com/customers`

Podemos obtener una lista de todos los usuarios.

Ahora, claramente estas son peticiones utópicas, puesto que el servidor puede poner restricciones y especificar que ciertos endpoints sean públicos y otros sean consultados de forma privada con una autenticación. Si lo comparamos con la realidad, nosotros no podemos consultar toda la información de todos los perfiles existentes en facebook, solamente podemos saber al detalle la información de nuestro perfil. Es por este motivo que se integran sistemas de registro y login en sitios, para validar que las peticiones estén autorizadas para consultar a ciertos servicios y servidores en particular.

Códigos de respuestas (Response Codes)

Cuando uno realiza una petición HTTP, además de traer información, en su metadata podemos observar diversos parámetros, uno de los más importantes son los códigos de respuestas a la petición realizada.

Dichos códigos nos permiten entender si la consulta fue relizada con éxito, si hay un problema con la consulta realizada por parte del cliente o bien, el servidor presenta un problema al momento de realizar la consulta.

Algunos códigos de respuestas conocidos:

- 200 (OK)
  201 (CREATED)
  204 (NO CONTENT)
  400 (BAD REQUEST)
  403 (FORBIDDEN)
  404 (NOT FOUND)
  500 (INTERNAL SERVER ERROR)

Para internalizar mejor estos conocimientos, invitamos a los estudiantes de este módulo a averiguar los contextos de dichas respuestas y como utilizar esta información a beneficio de las personas que desarrollan aplicaciones.

Algunos ejemplos

A modo de mostrar como funciona básicamente una petición HTTP mostraremos algunos ejemplos a continuación:

```
POST http://fashionboutique.com/customers
Body:
{
  “customer”: {
    “name” = “Scylla Buss”
    “email” = “scylla.buss@some.org”
  }
}
```

En este ejemplo realizamos una petición de tipo `POST` en donde pasamos por `Body` la creación de un nuevo customer con los parámetros explicitados.

Si realizamos la petición de forma correcta, deberíamos recibir una respuesta como ésta:

```
201 (CREATED)
Content-type: application/json
```

Un ejemplo de tipo `GET` podría ser:

```
GET http://fashionboutique.com/customers/123
Accept: application/json
```

Nuestra respuesta a esta petición debería ser la información de un `customer` de `id=123`. Además como información de metadata debería contener:

```
Status Code: 200 (OK)
Content-type: application/json
```

Como pueden observar, la estructura de peticiones bajo la arquitectura REST es siempre bajo las reglas de pedir, modificar, crear y borrar información. La estructura de las peticiones tiende a ser siempre similar, lo que permite que se genere un estándar al momento de realizar una petición.

Notación de Objetos de Javascript (JSON)

Ya sabemos como realizar peticiones, en base a un formato estándar y conocido. Lo que nos queda por resolver es cómo recibimos y decodificamos la información. Es acá donde nace el concepto de JSON.

```
{
  “customer”: {
    “name” = “Scylla Buss”
    “email” = “scyllabuss1@some.com”
  }
}
```

JSON es el acrónimo para JavaScript Object Notation, y aunque su nombre lo diga, no es necesariamente parte de JavaScript, de hecho es un estándar basado en texto plano para el intercambio de información, por lo que se usa en muchos sistemas que requieren mostrar o enviar información para ser interpretada por otros sistemas, la ventaja de JSON al ser un formato que es independiente de cualquier lenguaje de programación, es que los servicios que comparten información por éste método, no necesitan hablar el mismo idioma, es decir, el emisor puede ser Java y el receptor PHP, cada lenguaje tiene su propia librería para codificar y decodificar cadenas de JSON.

JSON puede representar cuatro tipos primitivos(cadenas, números, booleanos, valores nulos) y dos tipos estructurados(objetos y arreglos).

En JSON:

- Una Cadena es una secuencia de ceros o más caracteres Unicode.
- Un Objeto es una colección desordenada de cero o más pares nombre:valor, donde un nombre es una cadena y un valor es una cadena, número, booleano, nulo, objeto o arreglo.
- Un Arreglo es una secuencia desordenada de ceros o más valores.

Veamos un ejemplo:

```
var jsnow = {
   "nombre":"Jon Snow",
   "edad":27,
   "ciudad":"Winterfell",
   "altura":"175 cm",
   "peso":72
}
```

El ejemplo muestra un formato básico de un JSON. Veamos ahora otro ejemplo un poco más complejo.

```
var starks = {
  [
    {
      "nombre":"Jon Snow",
      "edad":27,
      "ciudad":"Winterfell",
      "altura":"175 cm",
      "peso":72
    },
    {
      "nombre":"Arya Stark",
      "edad":20,
      "ciudad":"Winterfell",
      "altura":"155 cm",
      "peso":52
    },
    {
      "nombre":"Sansa Stark",
      "edad":23,
      "ciudad":"Winterfell",
      "altura":"185 cm",
      "peso":75
    },
  ]
}
```

En este ejemplo podemos ver un JSON formado de un arreglo, en que cada elemento de dicho arreglo es otro objeto con información. Este tipo de respuestas es común verlas servicios web con una complejidad un poco más elevada, javascript es un lenguaje que puede integrarse muy bien al parsear e interpretar la información que recibe.

Utilizando AJAX

Vamos a ir dejando la teoría un poco de lado para aplicar los conceptos aprendidos utilizando AJAX. Para que la explicación sea más sencilla, consideraremos un ejemplo.

```
$('.btn').click(function() {

  $('.text').text('loading . . .');

  $.ajax({
    type:"GET",
    url:"https://api.meetup.com/2/cities",
    success: function(data) {
      $('.text').text(JSON.stringify(data));
    },
    dataType: 'jsonp',
  });
});
```

Si vemos el HTML que gatilla dicha acción:

```
<button type="button" class="btn">Click me!</button>
<p class="text">Replace me!!</p>
<script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
```

Vamos analizando paso por paso la solicitud.

Tenemos una función que se gatilla al hacer click en el botón de clase `btn`. Al realizar la acción de click, reemplazamos el texto presente en la etiqueta `p` con otro contenido y de forma asíncrona realizamos una consulta HTTP utilizando una función AJAX.

Dicha función recibe varios parámetros:

- type: intica el tipo de la petición
- url: el PATH al cual se hará la consulta
- success: una función que gatilla una acción cuando la petición fue relizada con éxito.

AL gatillar la función de éxito, se reemplaza el contenido en `p` con la repsuesta recibida por la API. En este caso se aplica una función que transforma la respuesta en formato JSON en una cadena de caracteres (string) de esa forma entregamos una respuesta comprimida del resultado.

Como pueden observar, una vez interiorizados los conceptos de API y JSON, es sencillo armar peticiones AJAX considerando los elementos necesarios para realizar dicha petición.

Considera además, que todas las peticiones son asíncronas, es decir, si esperas la respuesta de una petición para ejecutar otras acciones, siempre declara estas funciones dentro de la opción de tipo success. De esta forma nos aseguramos que una vez tengamos la respuesta, ejecutaremos la acción esperada.

Analicemos otro ejemplo:

```
$(document).ready(function(){
    $('.httpbin').click(function(){
        $.ajax({
        type:"POST",
        url:"https://httpbin.org/post",
        data:"test=POST+PROJECT&by=Eddie+Kidiw",
        dataType:"json",
        success: function(data) {
            $('.result').append('<ul class="list-group"><li class="list-group-item">'+data.form['by']+'</li><li class="list-group-item">'+data['form']['test']+'</li><li class="list-group-item">'+data['headers']['User-Agent']+'</li><li class="list-group-item">'+data['headers']['Content-Type']+'</li><li class="list-group-item">'+data['headers']['Accept-Encoding']+'</li><li class="list-group-item">'+data['headers']['Accept-Language']+'</li></ul>');
        }
        });
    });
});
```

El HTML que gatilla dicha acción es:

```
<div class="container">
    <button type="button" class="btn btn-primary btn-sm httpbin">Jquery Post</button>
    <div class="result"></div>
</div>
```

No entraremos a detallar la acción directa de este ejemplo, como pueden observar, realizamos una acción de tipo POST para luego renderizar de forma dinámica los datos de la máquina desde donde se realiza dicha petición. Para los nuevos programadores, siempre tengan presente que los navegadores guardan los datos de donde se realizan transacciones de datos, es por ello que es posible saber el tipo de máquina, navegador, sistema operativo, etc. Solamente con intercambiar información.

Les recomiendo programar este código de forma local y ver el resultado que les entrega a cada uno de ustedes. ¡La mejor forma de aprender es Probando!

Plugins

Competencias

- Entender el comportamiento y aspectos básicos de un plugin.
- Crear plugins propios para expandir conocimientos en el JQuery, profesionalizando el trabajo.
- Utilizar e integrar plugins existentes en algún proyecto.

Introducción

Si has llegado a este punto y has puesto en práctica los ejemplos realizados hasta ahora, no me cabe duda que tendrás ya una pequeña idea de las cosas que se pueden hacer con el framework. Habrás comprobado que, con pocas líneas de código, se pueden hacer diversos efectos y dotar a la página de interacción con el usuario, pero quizás todavía te sientas un poco perdido a la hora de encarar el desarrollo de problemas más complejos con los que podrás enfrentarte.

Es acá donde le damos un punto final (por ahora) a la infinita posibilidad de cosas que se pueden hacer con JQuery, hablaremos de los Plugins.

Entendiendo el Concepto de Plugin

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. Como ya sabemos, en este framework todo está basado en el objeto jQuery, así que con los plugins podemos añadirle nuevas utilidades.

¿Te imaginas creando tus propias capas de funcionalidades añadidas a JQuery? En este módulo te explicaremos como diseñar tus propias soluciones y utilizar soluciones populares trabajadas por terceros.

Creación de un Plugin

Para poder crear un plugin es necesario aplicar y conocer ciertas reglas que son necesarias para respetar la estructura de un plugin como tal. Por ejemplo:

- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.

Una vez conocidas estas reglas fundamentales, revisemos un ejemplo:

```
jQuery.fn.parpadea = function() {
  this.each(function(){
    elem = $(this);
    elem.fadeOut(250, function(){
      $(this).fadeIn(250);
    });
  });
  return this;
};
```

Volià Ya tenemos nuestro primer plugin. Como puedes observar en el ejemplo, para generar una función del plugin debemos invocar la función `jQuery.fn.[nombre_de_la_función]` de esta forma JQuery entenderá que es una función diseñada por tí y propia.

En particular esta función genera una especie de parpadeo suave en un elemento.

Bueno, ahora la queremos usar, entonces es sencillo:

```
$(document).ready(function(){
  //parpadean los elementos de class CSS "parpadear"
  $(".parpadear").parpadea();

  //añado evento click para un botón. Al pulsar parpadearán los elementos de clase parpadear
  $("#botonparpadear").click(function(){
    $(".parpadear").parpadea();
  })
})
```

¿Fácil no?

No hay nada más entretenido que definir funciones propias para aplicaciones más complejas que requieran acciones específicas.

Existen muchas librerías basadas en JQuery que por debajo hacen exactamente lo mismo que revisamos. A continuación te contaremos sobre Plugins populares y conocidos creados por la comunidad para uso común.

Integración de Plugins Externos basados en JQuery a un Proyecto

Muchos desarrolladores en el mundo, han aprovechado la potencia de JQuery para crear sus propios plugins basados en JQuery para generar proyectos más complejos y específicos. Añadir capas de funcionalidad a los proyectos y especificidad de algunas tareas. Estos proyectos complejos a su vez, los más populares siempre tienen soporte, son open source y abiertos para su uso. Permiten a equipos de desarrollo de pequeñas y medianas empresas, tener un punto de inicio para sus proyectos.

para integrar dichos plugins a un proyecto tomaré un ejemplo:

```
<!DOCTYPE html>
<html>
   <head>
      <title>CanvasJS Chart jQuery Plugin</title>
      <script type="text/javascript" src="https://canvasjs.com/assets/script/jquery-1.11.1.min.js"></script>

      <script type="text/javascript" src="https://canvasjs.com/assets/script/jquery.canvasjs.min.js">
</script>
      <script type="text/javascript">
      $(function () {
         $("#chartContainer").CanvasJSChart({
            data: [
            {
            type: "splineArea",
            dataPoints: [
               { x: 10, y: 10 },
               { x: 20, y: 14 },
               { x: 30, y: 18 },
               { x: 40, y: 22 },
               { x: 50, y: 18 },
               { x: 60, y: 28 }
            ]
         }
         ]
      });

      });
      </script>
   </head>
   <body>
   <div id="chartContainer" style="width:100%; height:300px;"></div>
   </body>
</html>
```

La integración de un plugin en un proyecto puede ser muy sencilla, la forma que recomendamos es que lo integres vía CDN (Content Delivery Network). En resumen, CDN es una url en donde el código de la librería es accesible a través de dicho link, de esa forma te aseguras que tu integración estará siempre en línea.

Es muy importante que antes de integrar el plugin tengas integrada la librería de JQuery, los `script` `html` los lee de forma secuencial, si no encuentra JQuery en tu proyecto antes que el plugin, tendrás errores al momento de ejecutar las funciones particulares que te ofrece el plugin.

Algunos Plugins Comunes

Canvas JS

Este plugin es uno de los más populares porque te permite crear gráficas diversas simplemente con añadir dicho plugin y la librería de JQuery. Por lo generar las librerías complejas que existen para generar gráficas son muy pesadas o requieren una instalación previa, tienden a sobrecargar el servidor en donde se encuentra alojado una aplicación web. En este sentido Canvas JS es muy interesante puesto que es una librería ligera y sencilla de implementar

```
<!DOCTYPE html>
<html>
  <head>
    <script>
    window.onload = function () {

    var options = {
      exportEnabled: true,
      animationEnabled: true,
      title:{
        text: "Units Sold VS Profit"
      },
      subtitles: [{
        text: "Click Legend to Hide or Unhide Data Series"
      }],
      axisX: {
        title: "States"
      },
      axisY: {
        title: "Units Sold",
        titleFontColor: "#4F81BC",
        lineColor: "#4F81BC",
        labelFontColor: "#4F81BC",
        tickColor: "#4F81BC",
        includeZero: false
      },
      axisY2: {
        title: "Profit in USD",
        titleFontColor: "#C0504E",
        lineColor: "#C0504E",
        labelFontColor: "#C0504E",
        tickColor: "#C0504E",
        includeZero: false
      },
      toolTip: {
        shared: true
      },
      legend: {
        cursor: "pointer",
        itemclick: toggleDataSeries
      },
      data: [{
        type: "spline",
        name: "Units Sold",
        showInLegend: true,
```

```
            xValueFormatString: "MMM YYYY",
            yValueFormatString: "#,##0 Units",
            dataPoints: [
                { x: new Date(2016, 0, 1),  y: 120 },
                { x: new Date(2016, 1, 1), y: 135 },
                { x: new Date(2016, 2, 1), y: 144 },
                { x: new Date(2016, 3, 1),  y: 103 },
                { x: new Date(2016, 4, 1),  y: 93 },
                { x: new Date(2016, 5, 1),  y: 129 },
                { x: new Date(2016, 6, 1), y: 143 },
                { x: new Date(2016, 7, 1), y: 156 },
                { x: new Date(2016, 8, 1),  y: 122 },
                { x: new Date(2016, 9, 1),  y: 106 },
                { x: new Date(2016, 10, 1),  y: 137 },
                { x: new Date(2016, 11, 1), y: 142 }
            ]
        },
        {
            type: "spline",
            name: "Profit",
            axisYType: "secondary",
            showInLegend: true,
            xValueFormatString: "MMM YYYY",
            yValueFormatString: "$#,##0.#",
            dataPoints: [
                { x: new Date(2016, 0, 1),  y: 19034.5 },
                { x: new Date(2016, 1, 1), y: 20015 },
                { x: new Date(2016, 2, 1), y: 27342 },
                { x: new Date(2016, 3, 1),  y: 20088 },
                { x: new Date(2016, 4, 1),  y: 20234 },
                { x: new Date(2016, 5, 1),  y: 29034 },
                { x: new Date(2016, 6, 1), y: 30487 },
                { x: new Date(2016, 7, 1), y: 32523 },
                { x: new Date(2016, 8, 1),  y: 20234 },
                { x: new Date(2016, 9, 1),  y: 27234 },
                { x: new Date(2016, 10, 1),  y: 33548 },
                { x: new Date(2016, 11, 1), y: 32534 }
            ]
        }]
    };
    $("#chartContainer").CanvasJSChart(options);

    function toggleDataSeries(e) {
        if (typeof (e.dataSeries.visible) === "undefined" || e.dataSeries.visible) {
            e.dataSeries.visible = false;
        } else {
            e.dataSeries.visible = true;
        }
        e.chart.render();
    }

    }
    </script>
</head>
<body>
<div id="chartContainer" style="height: 370px; width: 100%;"></div>
<script src="https://canvasjs.com/assets/script/jquery-1.11.1.min.js"></script>
<script src="https://canvasjs.com/assets/script/jquery.canvasjs.min.js"></script>
```

```
    </body>
</html>
```

En el ejemplo anterior podemos observar una forma de aplicar la librería a un proyecto, si revisan la documentación oficial, notarán que sus gráficas se pueden customizar y añadir una enorme cantidad de parámetros en formato JSON. Para conocer a fondo la potencia de este plugin recomendamos leer siempre la documentación oficial link.

Muuri

Te gustaría realizar componentes dinámicos que funcionen en base al concepto **drag & drop** (arrastrar y soltar). Muuri es la solución. Es un plugin muy completo creado a base de JQuery que permite generar contenedores y arrastrarlos hacia otro contenedor, manipulando desde la interfaz usuaria la disposición de los elementos.

Revisemos un ejemplo demo (el ejemplo completo lo puedes encontrar aquí).

```
<div class="grid grid-1">
  <div class="item">
    <div class="item-content">1</div>
  </div>
  <div class="item">
    <div class="item-content">2</div>
  </div>
  <div class="item">
    <div class="item-content">3</div>
  </div>
  <div class="item">
    <div class="item-content">4</div>
  </div>
  <div class="item">
    <div class="item-content">5</div>
  </div>
  <div class="item">
    <div class="item-content">6</div>
  </div>
  <div class="item">
    <div class="item-content">7</div>
  </div>
  <div class="item">
    <div class="item-content">8</div>
  </div>
  <div class="item">
    <div class="item-content">9</div>
  </div>
  <div class="item">
    <div class="item-content">10</div>
  </div>
  <script src="https://canvasjs.com/assets/script/jquery-1.11.1.min.js"></script>
  <script src="https://unpkg.com/muuri@0.7.1/dist/muuri.min.js"></script>
  <script>
    $(document).ready(function(){
      var grid1 = new Muuri('.grid-1', {
        dragEnabled: true,
        dragContainer: document.body,
```

```
            dragSort: function () {
                return [grid1]
            }
        });
    })
  </script>
</div>
```

Analicemos detenidamente el ejemplo anterior. Al igual como en el primer plugin que revisamos, es necesario incluirlo después que JQuery, de esta forma evitamos que la aplicación no funcione.

Este plugin en particular nos permite crear un nuevo objeto denominado `Muuri` (no es un nombre al azar, la librería indica que debe ser creado así) y a este objeto se le pasan ciertos parámetros. El primero es a los elementos del DOM que se verán afectados, en este caso los que tengan clase `grid-1`, posterior a ello se le entregan opciones, estas al detalle se encuentran en la documentació de dicho plugin, sin embargo, dentro de las propiedades entregadas también existen acciones, el ejemplo muestra la acción dragSort la cual retorna el arreglo ordenado generado por la variable llamada `grid1`.

Ahora es tu turno de investigar algunos plugins interesantes, la gama de posibilidad es infinita, con JQuery puedes hacer proyectos pequeños, complejos y de cualquier tema. La base siempre está en cómo aplicar las herramientas. Te invitamos a ser proactivo y aprender con estas herramientas, por tus medios.