Desarrollador Frontend

clase 3 | un mundo asincrónico

javascript asincrónico y XML

¿Qué es AJAX?

- Es un acrónimo de Asynchronous Javascript and XML
- Algo así como Javascript Asincrónico y XML
- Su implementación es parte del lenguaje javascript
- Es una técnica que permite realizar pedidos a un servidor programáticamente, manejando la respuesta en el cliente sin necesidad de cambiar de página
- Utilizando AJAX podemos generar contenidos por demanda del usuario

¿Qué **no** es AJAX?

- No es un lenguaje
- No es una biblioteca de funciones (como jQuery)
- No tiene nada que ver con la manipulación DOM
- No tiene nada que ver con animación de contenidos

Para conocer más sobre la historia de AJAX podemos visitar su entrada oficial en wikipedia

¿Quiénes usan AJAX?

AJAX es una técnica presente en la mayoría de las aplicaciones web de alto contenido dinámico. Algunos casos emblemáticos:

- Gmail: fue uno de los primeros clientes de correo que no requería refrescar la página para ver si había llegado un nuevo mensaje.
- Google Maps: a medida que el usuario recorre el mapa, busca el contenido de esa sección y llena las localizaciones
- Redes sociales: acciones como comentar, compartir, recibir notificaciones, son todas actualizaciones en tiempo real de la página mediante AJAX
- E-commerce y aplicativos: la mayoría de las acciones habituales como llenar un carrito de pedido, pedir más detalles, involucran pedidos asincrónicos

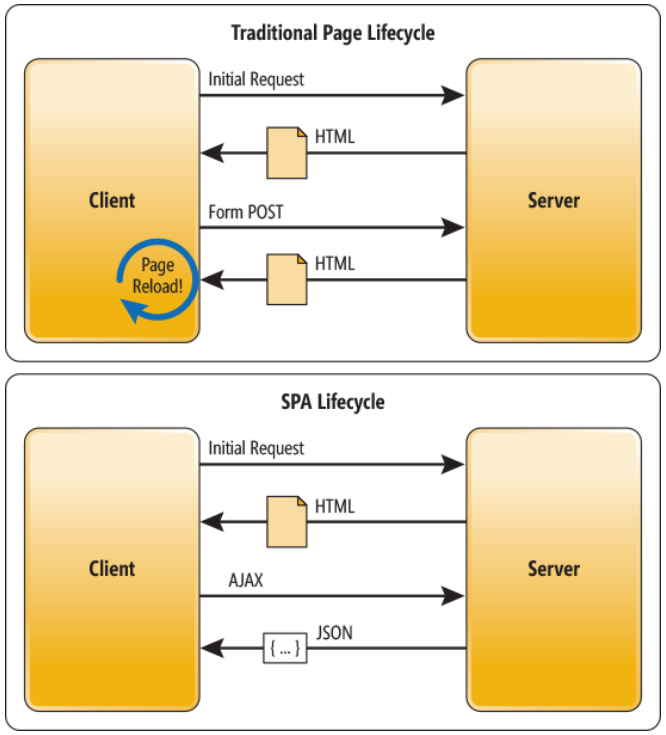
Usos de AJAX en otros proyectos

Proyectos pequeños como páginas web simples también hacen uso de esta técnica, hoy en día AJAX es parte de la caja de herramientas de cualquier desarrollador! Algunas utilizaciones que vemos cotidianamente:

- Validación de disponibilidad de nombre de usuario
- Listados de selección relacionados (provincia llena localidades)
- Buscadores con vista previa
- Cargadores de subsecciones altamente dinámicas
- Actualización de valores contra datos que ofrece un servidor

Es importante entender que para realizar la mayoría de estas tareas estamos combinando la petición de datos al servidor (AJAX) con técnicas de manipulación DOM (javascript plano), técnicas de animación de contenidos (CSS / JS). Bibliotecas como jQuery permiten integrar varios de estos puntos usando una interfaz de programación unificada.

Modelo tradicional vs Modelo AJAX



El ciclo de vida de una página tradicional puede resumirse en:

- El cliente realiza un pedido HTML a un servidor
- El cliente carga todos los activos necesarios (css, img, js)
- En algún momento, mediante la interacción, pide un nuevo recurso o envía datos
- Todos los procesos del documento se suspenden
- Lo que deriva en una nueva página (recarga)

En una Single Page Application o página con alto contenido AJAX:

- El cliente realiza un pedido HTML a un servidor
- El cliente carga todos los activos necesarios (css, img, js)
- En algún momento, mediante la interacción, pide un nuevo recurso o envía datos
- Si existen otros procesos siguen su curso
- Al llegar el resultado, el cliente sabe cómo mostrarlos

Ventajas y desventajas del asincronismo

Ventajas

- Puedo realizar pedidos en paralelo
- No bloquean el resto de la programación y el render de la página (por ejemplo un video que se está visualizando)
- Sólo pedimos al servidor el dato que necesitamos, lo que reduce el tráfico
- Orientado al consumo de servicios web

Desventajas

- La página es “javascript dependiente”
- La página siempre es la misma, por lo que no puedo compartir un sección, esto trae problemas también con el historial de navegación (botón atrás)
- Si gran parte del contenido es dinámico, tengo que realizar estrategias adicionales para que un buscador los indexe

La mayoría de las desventajas pueden trabajarse implementando bibliotecas javascript, aunque esto implica mayor esfuerzo que construir una página HTML tradicional.

Consideraciones previas

Antes de comenzar a trabajar con pedidos asincrónicos debo disponer de un **web server** en mi máquina, para poder realizar los pedidos en un contexto cliente/servidor (http) sin disparar un error de seguridad*.

Algunas posiblidades

- Instalar XAMPP o algún paquete similar**
- Instalar nodejs y utilizar node-static, express o similar
- Probar mi código en algún entorno web como plunker
- En linux / mac puedo utilizar el script python SimpleHTTPServer sin necesidad de instalar nada. Existen versiones similares en windows

* Actualmente los navegadores disparan un error de Cross Origin Request si el pedido que realiza AJAX no incluye el mismo protocolo, dominio y puerto que la página en la que se ejecuta. El protocolo file:// no es una opción válida.

** este tipo de paquetes incluyen PHP y MySQL, no necesarios para utilizar tecnologías client-side

El objeto XMLHttpRequest (XHR)

Este objeto javascript es el que permite realizar un pedido mediante http o https utilizando una interfaz de programación.

Está disponible desde versiones tempranas como IE5+ (a través de un plugin activeX) y fue adoptado por todos los navegadores hasta llegar al estándar actual XHR level 1 y level 2.

Estos pedidos pueden ser bloqueantes (sincrónicos) o asincrónicos.

Sólo en el caso de pedido asincrónico estamos hablando de AJAX, el uso más popular del XHR.

Puede realizar pedidos utilizando métodos como GET, POST, PUT, DELETE

Más información sobre la historia de XHR en la siguiente entrada de wikipedia

Ejemplo completo de pedido

```
var xhr_sincronico = new XMLHttpRequest();
xhr_sincronico.open('GET', 'ejemplo.txt', false);
xhr_sincronico.send();
console.log(xhr_sincronico.responseText);
```

```
var ajax = new XMLHttpRequest();
ajax.addEventListener('load', mostrarRespuesta);
ajax.open('GET', 'ejemplo.txt');
ajax.send();

function mostrarRespuesta(){
  console.log('ajax', ajax.responseText);
}
```

En el caso de un pedido sincrónico, en la linea 4 ya disponemos de la variable **responseText**, ya que la programación se bloquea hasta que XHR reciba la respuesta.

Lo que define que el pedido **no sea asincronico** es el 3er parámetro en el método **open**.

En el caso de un pedido AJAX, podemos observar que el objeto XHR se construye de la misma manera. Al utilizar el método open en modo sincrónico (el 3er parámetro por defecto es **true**) debemos manejar el resultado utilizando un **callback**

Manejando la respuesta

Como vimos en el ejemplo anterior, el objeto XHR almacena dentro de su propiedad **responseText** un String que contiene los caracteres obtenidos del pedido.

En caso de que la respuesta pueda ser interpretada como XML (XML / XHTML) podemos utilizar la propiedad **responseXML** para interactuar con la API XML.

La URL del pedido puede indicar un recurso estático (.txt, .html, .xml, .js) o también un recurso dinámico (.php, .asp, .jsp, /end-point). Lo que obtendremos como resultado siempre son los **caracteres** entregados por el servidor.

Actualmente el formato más utilizado de respuesta es JSON, como veremos en futuros encuentros.

Manejo de errores y testeo

En primer lugar debemos comprender **qué es un error** para AJAX.

Solamente veremos que se dispara el evento **error** en los casos en los que exista un error de red (por ejemplo, el servidor no existe).

Siempre que el servidor responda se disparará el evento **load**. Para poder manejar errores informados por el servidor podemos acceder al estado incluído en el encabezado de la respuesta...

Los estados con los que generalmente nos encontramos son:

- 200 ok
- 304 el recurso no fue modificado (el cliente utiliza el caché)
- 401 error de autenticación
- 403 error de permisos
- 404 error recurso no encontrado

Manejo de errores y testeo

```
var ajax = new XMLHttpRequest();
ajax.addEventListener('load', mostrarRespuesta);
ajax.addEventListener('error', mostrarError);
ajax.open('GET', 'ejemplo.txt');
ajax.send();

function mostrarError(){
  console.warn('ajax error');
}
```

Para que se ejecute el evento **error** la URL debería apuntar a un servidor inexistente.

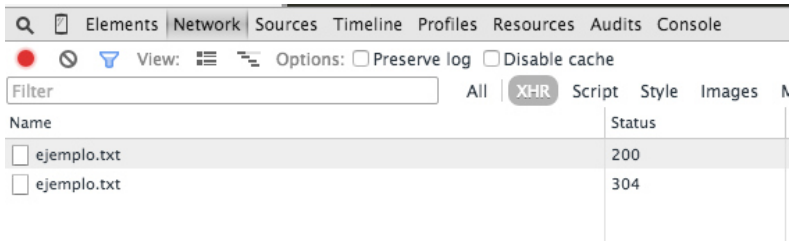
```
function mostrarRespuesta(){
  switch( ajax.status ){
   case 401:
   case 403:
   case 404:
     console.warn('ajax load', ajax.statusText);
     return;
   default:
     console.log('ajax load', ajax.responseText);
  }
}
```

Ejemplo de manejo de la respuesta utilizando la propiedad **status** del objeto XHR.

Manejo de errores y testeo

Las consolas de desarrollo de los navegadores modernos, incluyen una solapa informativa sobre todos los pedidos que realiza el cliente (imágenes, scripts, css, pedidos XHR).

Es muy importante estar atentos a esta información para poder debuggear y testear nuestro código. Un dato puede estar viniendo mal del servidor o podemos estar manejandolo mal desde el cliente.



Ejemplo de consola en google chrome, filtrando sólo los pedidos XHR.

Tocando la fila que nos interesa, podemos ver los caracteres obtenidos por el pedido.

Seguridad y CORS

Teniendo en cuenta la limitación del Cross Origin (el recurso pedido por AJAX debe estar ubicado en el mismo dominio que la página visualizada), ¿cómo podemos resolver un pedido a un servidor externo?

- Una opción es utilizar un servidor intermediario alojado en nuestro dominio (por ejemplo, un PHP que termina consultando los otros servicios)

- Los navegadores modernos (IE10+) implementan un mecanismo llamado CORS (Cross Origin Resource Sharing) que permite, configurando correctamente al servidor, obtener permisos para ejecutar el pedido a otros dominios.

Del lado del cliente, no tenemos que hacer nada para implementar CORS. Básicamente, el cliente efectúa un pedido utilizando el verbo OPTIONS del protocolo HTTP donde le informa al servidor que está queriendo obtener un dato. Si el servidor lo habilita (respondiendo status OK), el navegador efectúa el pedido AJAX.

Podemos leer más sobre CORS y ver implementaciones en diversos lenguajes server side en el siguiente link

Ejercicios en clase

- En el siguiente link se encuentran los ejercicios para trabajar en clase, con sus soluciones. Plunker nos permite ejecutar código online y compartirlo, sin necesidad de tener nada instalado más que el navegador web.