JAVASCRIPT FRONTEND AVANZADO

CLASE 5 : SPA & REST

SINGLE PAGE APPLICATION

Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.

En un SPA todos los códigos de HTML, JavaScript, y CSS se carga de una vez o los recursos necesarios se cargan dinámicamente como lo requiera la página y se van agregando, normalmente como respuesta de las acciones del usuario. La página no tiene que cargar otra vez en ningún punto del proceso tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API de HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación. La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está detrás. [1]

En el momento en el que empezamos a traer contenido dinámico reemplazando parte del contenido actual del cliente estamos creando un nuevo estado de navegación al cual los usuarios no van a poder acceder a menos que nos encarguemos de guardar estos estados y reflejarlos en el front-end para que los usuarios los puedan usar en un futuro. Para ello vamos a aprender a controlar el historial del cliente y su posición dentro de la navegación a través de la URL para que puedan no solo compartir ese estado a otros usuarios sino que también puedan acceder a ese estado previo de su propia navegación.

HISTORY API

La API WEB de History nos permite manipular la información del historial de sesión del cliente para la pestaña o frame en el que actualmente se encuentra navegando. Su interfaz se encuentra como propiedad de window en window.history.

Este objeto nos da algunas propiedades y métodos de los cuales nos van a estar interesando el state y pushState()

STATE

Representa el estado de navegación correspondiente al estado actual de la URL y posición de sesión de historial del cliente de la pestaña actual. El mismo puede ser de cualquier tipo de dato.

PUSHSTATE

El método pushState nos permite agregar información a la sesión del historial del cliente. Podemos especificar un título [2] , caché y una URL.

```
History.pushState(Any state, String title, String URL)
```

Agregando una URL haría que la misma se muestre en el navegador pero no redirige al usuario a esa ubicación, lo cual nos permite conservar nuestro estado del DOM. Agregar información extra en el parámetro de state nos permitirá en un futuro poder acceder a esa información si el usuario retrocede o avanza en su historial de navegación, a través de la propiedad state de la interfaz de la API.

POPSTATE

El evento popstate se suele usar conjuntamente con la WEB API History ya que el mismo se dispara cada vez que el usuario intenta retroceder o avanzar en su historial usando cualquiera de los métodos disponibles :

- Botones atrás y adelante en la barra de navegación
- Las combinaciones de teclas Ctrl+> / Ctrl+<
- Usando los métodos back() y forward() de la misma API

El objeto evento de un popstate, además de la información habitual con la que ya cuenta por defecto además contará con una referencia directa a la propiedad state de history que corresponda al estado de navegación actual del usuario.

OPTIMISTIC UI

Optimistic User Interfaces Es solamente una técnica para manejar las interacciones humano-computadora. La misma se basa en poder mostrar un resultado óptimo por default dándole al usuario la sensación de satisfacción ante a operación realizada mientras por atrás y de manera asincrónica no bloqueante realizamos todas nuestras operaciones. En el

caso de que alguna falle, solo en ese caso interactuamos con el usuario para informar de lo sucedido, de lo contrario él ya pensará que todo funcionó como corresponde.

Vamos a pasar por un ejemplo básico para integrar AJAX , History API y Optimistic UI. Primero tengamos nuestro front-end armado :

```
<!DOCTYPE html>
<html lang="en">
   <head>
       <meta charset="UTF-8">
       <title>UI</title>
   </head>
   <body>
       <nav>
           <a href="link_uno.html">link uno</a>
           <a href="link_dos.html">link dos</a>
           <a href="link_tres.html">link tres</a>
       </nav>
       <script src="script.js"></script>
   </body>
</html>
```

Vamos a tener una barra de navegación que nos permita cargar el contenido de esos tres archivos pero de manera asincrónica, es decir usando AJAX. Para eso vamos a registrarles un click y cancelarles su comportamiento por defecto, de lo contrario nos van a redirigir a esas ubicaciones:

```
const links = document.querySelectorAll("a")
links.forEach(link => {
    link.addEventListener("click",e=>{
        e.preventDefault()
        e.stopPropagation()
        //....
    })
});
```

Vamos a querer también pedir por AJAX los archivos correspondientes dentro del evento click. Cuando tengamos la respuesta del servidor entonces vamos a querer guardar una copia en el historial de sesión :

```
//...
const xhr = new XMLHttpRequest()
xhr.open("get",url)
xhr.addEventListener("load",e=>{
    if (xhr.status == 200) {
        history.pushState(xhr.response,"",url)
        //...
    }
})
xhr.send()
//...
```

De esta forma, cada vez que el usuario vuelva para atrás en su historial, podemos interceptar su pedido con un evento de popstate y usar el estado actual de navegación dentro del caché de sesión para construir su DOM y si el contenido fuera dinámico, pedir la información faltante al servidor de manera asincrónica y al obtenerla inyectarla sin que el usuario pierda interacción con el front-end :

```
window.addEventListener("popstate",e=>{
    console.log(e.state)
    document.body.innerHTML = e.state
})
```

REST API

Representational State Transfer (REST) es un estilo arquitectónico que define un conjunto de restricciones y propiedades basadas en HTTP. Los servicios web que se ajustan al estilo arquitectónico REST o a los servicios web RESTful brindan interoperabilidad entre los sistemas informáticos de Internet. Los servicios web que cumplen con REST permiten que los sistemas solicitantes accedan y manipulen las representaciones textuales de los recursos web mediante el uso de un conjunto uniforme y predefinido de operaciones sin estado.

Otros tipos de servicios web, como los servicios web SOAP, exponen sus propios conjuntos arbitrarios de operaciones.

Los "recursos web" se definieron por primera vez en la World Wide Web como documentos o archivos identificados por sus URL. Sin embargo, hoy tienen una definición mucho más genérica y abstracta que abarca cada cosa o entidad que se puede identificar, nombrar, abordar o manejar, de cualquier manera, en la web.

En un servicio web RESTful, las solicitudes realizadas al URI de un recurso provocarán una respuesta que puede estar en HTML, XML, JSON o en algún otro formato. La respuesta puede confirmar que se ha realizado alguna modificación en el recurso almacenado, y la respuesta puede proporcionar enlaces de hipertexto a otros recursos relacionados o colecciones de recursos.

Cuando se usa HTTP, como es más común, las operaciones disponibles son GET, POST, PUT, DELETE y otros métodos HTTP CRUD predefinidos. Mediante el uso de un protocolo sin estado y operaciones estándar, los sistemas REST apuntan a un rendimiento rápido, confiabilidad y la capacidad de crecer, al reutilizar componentes que se pueden administrar y actualizar sin afectar al sistema en su conjunto, incluso mientras se está ejecutando [2].

CONCEPTOS REST

Para entender bien la dinámica de una API REST tenemos que conocer primero cada parte que la componen, de esta forma el día de mañana si quisiéramos usar una API que nunca hayamos usado, tan solo con conocer qué partes son fundamentales dentro de una API REST podríamos indagar en su misma documentación buscando aquellos pedazos de información que sabemos que nos van a ser útiles para empezar a usarla [3].

1. **Uso correcto de las URIs**
    - URI de base
    - URI de cada recurso
    - Parametros de filtrado
2. **Uso correcto de HTTP**
    - Métodos
    - Códigos de estado
    - Formato

URI BASE

Va a ser la URI a partir de la cual vamos a hacer todos nuestros pedidos a la API. La misma suele ser el mismo dominio del sitio o bien agregando un path extra identificatorio de la API REST.

EJ.: Implicando que nuestro sitio se encuentra en www.misitio.com , la API REST podría encontrarse quizá en www.misitio.com/api

RECURSOS

Cada uno de los recursos debe tener su propia URI que los identifique inequívocamente de los demás. Estos puntos de acceso, también a veces llamados funciones, recursos(resources), end-points , etc, van a tener una representación total de cada uno de los recursos que contempla la API. Para lograr este cometido, se deben tener en cuenta algunos conceptos como por ejemplo que la URI de cada recurso deben ser únicas por recurso, no deben presentar acciones (es decir que no se deberían usar verbos en ellas) ni identificar formatos (Es decir que una URL terminada en .../recurso.json no sería correcto) .

EJ.: Implicando que nuestra API cuenta con los recursos de Usuarios , Mensajes y Logs , podríamos tener una URI para cada uno de la siguiente forma :

- *www.misitio.com/usuarios*
- *www.misitio.com/mensajes*
- *www.misitio.com/logs*

PARAMETROS

Para filtrar, ordenar, paginar o buscar información en un recurso, debemos hacer una consulta sobre la URI, utilizando parámetros HTTP en lugar de incluirlos en la misma. Esto implica que deberíamos seguir usando parámetros de la forma en que estamos acostumbrados por formularios de HTML y no que ellos formen parte de la URI.

EJ.: Implicando que un Usuario se puede filtrar por su fecha de registro y edad, la siguiente URI sería incorrecta :

www.misitio.com/usuarios/fecha-registro/2018/01/01/order/edad/ASC

En vez de eso, podríamos tener parámetros tradicionales del tipo :

www.misitio.com/usuarios?fecha_registro=2018-01-01&orderAsc=edad

METODOS

Como hemos visto en el anterior nivel, a la hora de crear URIs no debemos poner verbos que impliquen acción, aunque queramos manipular el recurso.
Para manipular los recursos, HTTP nos dota de los siguientes métodos con los cuales debemos operar:

- **GET** : Para consultar y leer recursos
- **POST** : Para crear recursos
- **PUT** : Para editar recursos
- **DELETE** : Para eliminar recursos.
- **PATCH** : Para editar partes concretas de un recurso.

Entonces una misma URI puede tener implementación dual dependiendo el método por el que se intentó acceder a ella.

CÓDIGOS DE ESTADO

Uno de los errores más frecuentes a la hora de construir una API suele ser el reinventar la rueda creando nuestras propias herramientas en lugar de utilizar las que ya han sido creadas, pensadas y testadas. La rueda más reinventada en el desarrollo de APIs son los códigos de error y códigos de estado.
Cuando realizamos una operación, es vital saber si dicha operación se ha realizado con éxito o en caso contrario, por qué ha fallado.

Un error muy común es querer enviar códigos de error customizados cuando los mismos ya existen dentro del universo de HTTP. Este es un error común que tiene varios inconvenientes:

- No es REST ni es estándar.
- El cliente que acceda a este API debe conocer el funcionamiento especial y cómo tratar los errores de la misma, por lo que requiere un esfuerzo adicional importante para trabajar con nosotros.
- Tenemos que preocuparnos por mantener nuestros propios códigos o mensajes de error, con todo lo que eso supone.

FORMATO

Cuando hablamos sobre URLs, vimos también que no era correcto indicar el tipo de formato de un recurso al cual queremos acceder o manipular.

HTTP nos permite especificar en qué formato queremos recibir el recurso, pudiendo indicar varios en orden de preferencia, para ello utilizamos el header **Accept**.

Nuestra API devolverá el recurso en el primer formato disponible y, de no poder mostrar el recurso en ninguno de los formatos indicados por el cliente mediante el header Accept, devolverá el código de estado **HTTP 406**.En la respuesta, se devolverá el header **Content-Type**, para que el cliente sepa qué formato se devuelve.

Tradicionalmente los formatos disponibles de una API REST son XML, JSON y JSONP

CORS

El Intercambio de Recursos de Origen Cruzado (CORS) es un mecanismo que utiliza encabezados adicionales HTTP para permitir que un user agent obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio), al que pertenece. Un agente crea una petición HTTP de origen cruzado cuando solicita un recurso desde un dominio distinto, un protocolo o un puerto diferente al del documento que lo generó.

Por ejemplo, una página HTML localizada en http://domain-a.com hace una solicitud <img src a http://domain-b.com/image.jpg. Actualmente muchas páginas en la web cargan recursos como hojas de estilos CSS, imágenes y scripts desde dominios separados, como son redes de distribución de contenido (CDNs).

Por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script. Por ejemplo, XMLHttpRequest y la API Fetch siguen la política de mismo-origen. Ésto significa que una aplicación que utilice esas APIs XMLHttpRequest sólo puede hacer solicitudes HTTP a su propio dominio a menos que se utilicen encabezados CORS [4].

El encabezado de respuesta Access-Control-Allow-Origin por ejemplo, indica si la respuesta puede ser compartida con recursos del origin dado. Si el servidor responde con un `Access-Control-Allow-Origin: *` significa que el recurso puede ser accedido por **cualquier** dominio en una forma de sitio cruzado.

JSONP

JSONP o **JSON con *padding*** es una técnica de comunicación utilizada en los programas JavaScript para realizar llamadas asíncronas a dominios diferentes. JSONP es un método concebido para suplir la limitación de AJAX entre dominios, que únicamente permite realizar peticiones a páginas que se encuentran bajo el mismo dominio y puerto por razones de seguridad [6].

Bajo la política del mismo origen (SOP), un código JavaScript cargado en un dominio no tiene permiso para obtener datos de otro dominio. Sin embargo, esta restricción no se aplica a la etiqueta `<script>` de HTML, para la cual se puede especificar en su atributo `src` la URL de un script alojado en un servidor remoto.

Para realizar llamadas asíncronas entre distintos dominios, los datos se obtienen cargando un script que contiene el objeto JSON. De por sí, esto provocaría un error de sintaxis ya que el navegador estaría tratando de interpretar JSON crudo como JavaScript. Y aún suponiendo que el navegador interpretase el objeto JSON como un objeto literal de JavaScript, sin estar asignado a una variable no sería accesible.

En la técnica de JSONP, el objeto JSON se devuelve envuelto en la llamada de una función. De esta forma, una función ya definida en el entorno de JavaScript podría manipular los datos JSON. Por convención, el nombre de la función de retorno se especifica mediante un parámetro de la consulta, normalmente, utilizando `jsonp` o `callback` como nombre del campo en la solicitud al servidor.

```
<script src="http://misitio.com/api/usuarios?callback=manejarRespuesta"></script>
```

Aunque el padding es generalmente el nombre de la función que envuelve al objeto JSON, también es posible asignar el objeto a una variable o realizar cualquier otra sentencia o declaración de JavaScript. La petición JSONP no es JSON crudo y no es analizada como tal, por lo que se puede devolver cualquier expresión válida de JavaScript, y no tiene por qué incluir JSON.

Hay que tener en cuenta que agregar un script de manera dinámica dentro del DOM implica que su contenido se evalúa de manera asincrónica, es decir que no bloquea el resto del programa, por lo que el usuario puede seguir navegando sin inconvenientes mientras que el programa sigue probablemente descargando o bien parseando la información obtenida a cambio por la API.

1. https://es.wikipedia.org/wiki/Single-page_application
2. https://en.wikipedia.org/wiki/Representational_state_transfer
3. http://asiermarques.com/2013/conceptos-sobre-apis-rest/
4. https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS
5. https://developer.mozilla.org/es/docs/Web/HTTP/Headers/Access-Control-Allow-Origin
6. https://es.wikipedia.org/wiki/JSONP