SEMINARIO DE JS LENGUAJES

CLASE 6

EXPLORAR EL FUNCIONAMIENTO DEL MULTITASKING Y ASNICRONISMO EN JAVASCRIPT

CLIENTES Y SERVIDORES

Cuando visitamos un sitio web con nuestro navegador predilecto (Firefox, Chrome, Edge, etc.) se produce una conexión vía Internet entre un cliente —quien inicia la conexión y solicita el contenido del sitio web— y un servidor —quien recibe la conexión y envía el contenido solicitado.

CLIENTES Y SERVIDORES

Los clientes hacen peticiones, son dispositivos de distinto tipo. Los servidores son como un negocio, atienden las peticiones de los clientes.

CLIENTES Y SERVIDORES

- El navegador se conecta con el DNS para localizar la dirección del servidor (la dirección de la tienda)
- El navegador envía una petición HTTP al servidor pidiéndole una copia de la página. Este mensaje se envía vía una conexión TCP/IP
- Si el servidor aprueba la solicitud, envía “200 OK” y comienza a enviar los archivos de la solicitud, en trozos pequeños llamados “paquetes de datos”.
- El navegador reúne todos esos trozos, forma el documento HTML y lo muestra (tus nuevas compras de la tienda están en casa).

PROTOCOLO HTTP

¿QUÉ ES HTTP Y HTTPS? | Aprende que es http, https y para que utiliza en menos 5 minutos! Y repasamos... Kahoot

SINCRONISMO - 1

Pensemos que le pedimos a una persona que haga la Tarea A, luego la Tarea B y después la Tarea C. En un modelo síncronico, la persona tiene que completar la Tarea A por completo antes de poder empezar la Tarea B. Y la Tarea B debe terminarse completamente antes de empezar la Tarea C.

Si la Tarea A tarda mucho, la persona se queda "bloqueada" y no puede empezar la Tarea B ni la C hasta que A termine.

SINCRONISMO - 2

En el contexto de un programa, las instrucciones se ejecutan una después de otra, en orden, y espera que una termine para continuar.
Si una operación tarda mucho (por ejemplo, cargar una página web lenta o leer un archivo grande), el programa se detiene y no responde hasta que esa operación finaliza.

FUNCIONES SINCRÓNICAS EN JAVASCRIPT

JavaScript tiene un hilo de proceso único, lo que significa que dos porciones del script no pueden ejecutarse al mismo tiempo, deben hacerlo una después de la otra

FUNCIONES SINCRÓNICAS

```
function saludar(name) {
        return `Hola, mi nombre es ${name}!`;
    }

const name ='Trinidad';
const saludo = saludar(name);
console.log(saludo);
```

saludar() espera a que se termine de ejecutar para continuar con el resto del código

FUNCIONES SINCRÓNICAS

¿Qué sucede si una función tarda demasiado en ejecutar?

Es bloqueado y no es posible seguir ejecutando.

Ejemplo de generar primos. Abre una nueva página

ASINCRONISMO - 1

Ahora le pedimos a esa persona que empiece la Tarea A (que va a tardar un poco). En lugar de quedarse esperando a que A termine, la persona empieza la Tarea A e inmediatamente, se mueve para empezar la Tarea B. Luego, empieza la Tarea C.

La persona no se queda "bloqueada" esperando.

Empezó las tres tareas y, en algún momento en el futuro, recibirá una notificación:

"¡Ya terminé la Tarea A!", "¡Ya terminé la Tarea B!", etc.

FUNCIONES ASINCRÓNICAS

En algunos casos, sería ideal:

- Ejecutar una función que lleva un tiempo
- Mientras se ejecuta, el programa continúa ejecutando otros eventos
- Ser notificado del resultado de la ejecución

ASINCRONISMO EN JS

El asincronismo es una técnica que permite al programador habilitar una tarea que potencialmente va a demorar y en lugar de esperar continua respondiendo a otros eventos mientras la tarea se sigue ejecutando.
Es similar a la atención de un mozo en un restaurant.

Un mozo atiende muchas mesas

ASINCRONISMO - 2

Mientras las tareas asíncronas se están ejecutando en "segundo plano", el programa principal continua la ejecución otras instrucciones y no se detiene. Cuando una operación asíncrona finaliza, el programa es notificado (a través de mecanismos como callbacks, promesas o async/await) y puede entonces procesar el resultado o manejar el error.

¿POR QUÉ ES IMPORTANTE EL ASINCRONISMO?

Responsividad de la Interfaz de Usuario
Operaciones de Entrada/Salida (I/O)
Optimización del Rendimiento

ASINCRONISMO - 3

```
console.log("Before setTimeout()");
setTimeout(function() {
        console.log("In the callback function");
    }, 5000); // 5000 milliseconds, or 5 seconds
console.log("After setTimeout()");
```

FUNCIONES ASINCRÓNICAS EN JS

- fetch( ) para hacer requerimientos HTTP
- getUserMedia( ) para acceder a la cámara o microfono del usuario
- showOpenFilePicker() para permitir al usuario subir un archivo

FUNCIONES ASINCRÓNICAS

Se utiliza la API XMLHttpRequest para asociar eventos asincrónicos, con un manejador de eventos. Es un mecanismo muy confuso y con algunos problemas en la gestión de las funciones callbacks.

OBJETO PROMISE O PROMESAS

Imaginemos que somos un cantante de rock y nuestros fans preguntan día y noche cuando va a estar nuestra próxima canción. Generás una lista para que dejen ahí su email y les avisarás cuando esté la canción, para que todos la reciban cuando la canción esté disponible. Si algo sale mal, un incendio en el estudio por ejemplo, también le vas a avisar. Nadie sale abrumado y los fanáticos no se pierden la canción.

PROMESAS EN JS

En la vida real tenemos:

- Un código productor (el cantante) que se toma su tiempo para cargar los datos a través de una red
- Un código consumidor, que requiere el resultado del código consumidor. Muchas funciones pueden requerir este código, son los fans
- Una promesa es el código de JS que une al productor con el consumidor

La analogía no es tan precisa porque las promesas son más complejas, pero sirven para empezar.

"I Promise a Result!"

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

JavaScript Promise. w3schools

OBJETO PROMESA

```
let promise = new Promise(function(resolve, reject) {
/* "Contructor del objeto promesa: let promise=new Promise(function(
```

La función se denomina ejecutor.
resolve y reject son funciones callbacks proporcionadas por JS.
El ejecutor corre automáticamente e intenta realizar una tarea.
Cuando termina con el intento, llama a resolve si fue exitoso o reject si hubo un error.

OBJETO PROMISE - PROPIEDADES

- state: inicialmente "pendiente", luego cambia a "cumplido" cuando se llama a resolve o "rechazado" cuando se llama a reject.
- result: inicialmente undefined, luego cambia a valor cuando se llama a resolve(valor) o error cuando se llama a reject(error).

Estos estados se acceden con los métodos .then/.catch/.finally

OBJETO PROMISE - MÉTODOS

Las funciones consumidoras (los “fanáticos”), recibirán el resultado o error. Las funciones de consumo pueden registrarse (suscribirse) utilizando los métodos .then, .catch y .finally.

OBJETO PROMISE - EJEMPLO

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve("hecho!"), 1000);
});

// resolve ejecuta la primera función en .then
promise.then(
    result => alert(result), // muestra "hecho!" después de 1 segund
    error => alert(error)    // no se ejecuta
);
```

OBJETO PROMISE - EJEMPLO

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// reject ejecuta la segunda función en .then
promise.then(
  result => alert(result), // no se ejecuta
  error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo
);
```

OBJETO PROMISE - FINALLY

Se ejecuta siempre más allá del resultado.
Útil para realizar la limpieza

```
new Promise((resolve, reject) => {
setTimeout(() => resolve("resultado"),
    })
.then(result => alert(result))
.finally(() => alert("Promesa lista"
```

EJEMPLOS COMPLETO

Ejemplo obtener datos de un usuario

VENTAJAS DE LAS PROMESAS

- Permiten hacer las cosas en el orden natural. Primero, ejecutamos loadScript (script), y .then escribimos qué hacer con el resultado.
- Podemos llamar a “.then” en una promesa tantas veces como queramos. Cada vez, estamos agregando un nuevo “fan”, una nueva función de suscripción, a la “lista de suscripción”.
- Las promesas se pueden encadenar en una Cadena de promesas

CADENA DE PROMESAS

```
..
obtenerUsuario()
    .then(usuario => {
        alert(`Usuario: ${usuario.nombre
        return obtenerPedidos(usuario.id
        })
    .then(pedidos => {
        alert(`Pedidos: ${pedidos.join("
        })
.catch(error => alert(error));
```

WEB API

- Un Web API es una API para la Web.
- La Browser API puede extender la funcionalidad del navegador para soportar operaciones complejas. Por ejemplo la geolocalization API para acceder a la ubicación del servidor
- La server API puede extender la funcionalidad del servidor
- APIs de terceros, como OpenAI, X(ex Twitter), Facebook, ...
- API Fetch permite hacer requerimientos HTTP a un servidor. No se necesita XMLHttpRequest

API FETCH

- Provee un método global fetch() para proporcionar recursos de la red en forma asincrónica.
- Sirve para hacer peticiones HTTP (por ejemplo traer datos de una API o enviar información a un servidor)
- Permite asociar otros conceptos relacionados con HTTP como CORS y extensiones HTTP.
- Retorna una promesa que se procesa con un objeto response.

API FETCH

```
const url = "https://jsonplaceholder.typ
fetch(url)
    .then(respuesta => respuesta.json())
    .then(datos => {
        alert("Título del post: " + datc
    })
    .catch(error => {
        alert("Error al obtener el post:
    });
```

FETCH - PARAMETROS

fetch(url, options)

FETCH - PARAMETROS - OPCIONES

```
//  Opciones de la petición
const opciones = {
    method: "GET"
};

//Petición HTTP
fetch("http://misitio.edu.ar/gatitos.json", opciones)
    .then(response => response.text())
    .then(data => {
        /** Procesar los datos **/
});
```

```
//  Opciones de la petición
const opciones = {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify(jsonData)
};


//Petición HTTP
fetch("http://misitio.edu.ar/gatitos.json", opciones)
    .then(response => response.text())
```

EL OBJETO RESPONSE

```
//En forma más eficiente
fetch('http://misitio.com.ar/ajax-info.json')
  .then(function(response)){
    /** Código que procesa la respuesta **/
    });
```

- Objeto Response. Posee propiedades y métodos.

Propiedad	Descripción	Tipo
.status	Código de error HTTP de la respuesta (100-599).	Number
.statusText	Texto representativo del código de error HTTP anterior.	String
.ok	Devuelve true si el código HTTP es 200 (o empieza por 2).	Boolean
.headers	Cabeceras de la respuesta.	Objeto
.url	URL de la petición HTTP.	String

EL OBJETO RESPONSE. MÉTODOS

```
fetch('http://misitio.com.ar/ajax-info.json')
 .then(response => response.text())
 .then(data => console.log(data));
```

Método	Descripción	Tipo
.text()	Devuelve una promesa con el texto plano de la respuesta.	String
.json()	Idem anterior con un objeto json. Equivalente a usar JSON.parse().	Objeto
.blob()	Idem anterior con un objeto Blob (binary large object).	Objeto
.arrayBuffer()	Idem anterior con un objeto ArrayBuffer.	Objeto
.formData()	Idem anterior con un objeto FormData.	Objeto
.clone()	Crea y devuelve un clon de la instancia.	Objeto
Response.error()	Devuelve un nuevo objeto Response con un error de red asociado.	Objeto
Response.redirect(url, code)	Redirige a una url, opcionalmente con un code de error.	Objeto

OBJETO RESPONSE

```
// Petición HTTP
fetch('http://misitio.com.ar/ajax-info.json')
  .then(response => {
    if (response.ok)
        return response.text()
    else
        throw new Error(response.status);
    })
.then(data => {
    console.log("Datos: " + data);
 })
.catch(err => {
    console.error("ERROR: ", err.message)
});
```

Comprobamos que la petición es correcta.

Procesamos la respuesta con .text()

Si hubo error, levantamos una excepción con el código de error.

Procesamos los datos y se muestran en consola.

Si la promise es rechazada, se captura el error con catch().

```
// Alternativa a la anterior
const isResponseOk = (response) => {
if (!response.ok)
    throw new Error(response.status);
    return response.text()
}

fetch('http://misitio.com.ar/ajax-info.json')
  .then(response => isResponseOk(response))
  .then(data => console.log("Datos: ", data))
  .catch(err => console.error("ERROR: ", err.message));
```

- Se agrega una función flecha para procesar la respuesta.
- Los .then y .data se utilizan también con funciones flecha para simplificarlos.

ASYNC-AWAIT

Código más moderno y legible. Introducido en ES2017

```
async function mostrarUsuario() {
try {
    const usuario = await obtenerUsuario
    alert(`Usuario: ${usuario.nombre}, E
        } catch (error) {
    alert(error);
    }
}
```

OBJETO REQUEST

- JS permite crear nuestros propios objetos request, constructor Request.
- Se puede utilizar para pasar como parámetro a fetch

OBJETO REQUEST

```
var myHeaders = new Headers();

var myInit = {
    method: 'GET',
    headers: myHeaders,
    mode: 'cors',
    cache: 'default'
};

var myRequest = new Request('flowers.jpg

fetch(myRequest)
    .then(function(response) {
        return response.blob();
```

JSON

Es una notación de objetos para el intercambio de datos. Popularizado por Douglas Crockford. Formado por un subconjunto de JS. Standard ECMA-262 3rd Edition - Diciembre 1999 Leerlo y escribirlo es simple para humanos y fácil y para las maquinas interpretarlo y generarlo. Los JSON son cadenas, que deben ser convertidas a Javascript para acceder a sus datos.

JSON está constituido por dos estructuras: Una colección de pares de nombre/valor Una lista ordenada de valores.

JSON

```
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
```

JSON FORMAS

- Objetos
- Strings
- Arreglos
- Number
- Espacios en blanco

JSON

Convertir una cadena a un objeto nativo se denomina parsing, mientras que convertir un objeto nativo a una cadena para que pueda ser transferido a través de la red se denomina stringification

JSON EJEMPLO

```
superHeroes.homeTown
superHeroes['active']
superHeroes['members'][1]['powers'][2]
```

JSON

Algunas consideraciones importantes..

- JSON es sólo un formato de datos — contiene sólo propiedades, no métodos.
- JSON requiere usar comillas dobles para las cadenas y los nombres de propiedades. Las comillas simples no son válidas.
- validar JSON utilizando una aplicación como JSonlint

REFERENCIAS

Asincronía en JS

Promesas en JS

Promise Basics

Using Fetch

Herencia de cadenas de prototipos

Interpolación de cadenas en JS

JSon Specification

FIN DE LA CLASE