1. Clases

Las clases de javascript, introducidas en ECMAScript 2015, son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript.

1.1 Declaración de clases.

Una manera de definir una clase es mediante una **declaración de clase**. Para declarar una clase, se utiliza la palabra reservada `class` y un nombre para la clase "Rectangulo".

```
class Rectangulo {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
}
```

1.2 Constructor

El método `constructor` es un método especial para crear e inicializar un objeto creado con una `clase`. Solo puede haber un método especial con el nombre "constructor" en una clase. Si esta contiene mas de una ocurrencia del método **`constructor`**, se arrojará un *Error* "SyntaxError"

Un **constructor** puede usar la *palabra reservada* **`super`** para llamar al **constructor** de una *superclase*

1.3 Métodos estáticos

La *palabra clave* `static` define un método estático para una clase. Los métodos estáticos son llamados sin instanciar su clase y **no** pueden ser llamados mediante una instancia de clase. Los métodos estáticos son a menudo usados para crear funciones de utilidad para una aplicación.

```
class Punto {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static distancia(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;

    return Math.sqrt(dx * dx + dy * dy);
  }
}

const p1 = new Punto(5, 5);
const p2 = new Punto(10, 10);

console.log(Punto.distancia(p1, p2)); // 7.0710678118654755
```

1.4 Subclases con `extends`

La palabra clave `extends` es usada en *declaraciones de clase* o *expresiones de clase* para crear una clase hija.

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hablar() {
    console.log(this.nombre + " hace un ruido.");
  }
}

class Perro extends Animal {
  hablar() {
    console.log(this.nombre + " ladra.");
  }
}
```

Extender objetos implementados por funciones

También se pueden extender las clases tradicionales basadas en funciones:

```
function Animal(nombre) {
  this.nombre = nombre;
}
Animal.prototype.hablar = function () {
  console.log(this.nombre + "hace un ruido.");
};

class Perro extends Animal {
  hablar() {
    super.hablar();
    console.log(this.nombre + " ladra.");
  }
}

var p = new Perro("Trosky");
p.hablar();
```

1.5 Propiedades privadas.

Las propiedades de la clase son públicas de forma predeterminada y se pueden examinar o modificar fuera de la clase. Sin embargo, existe una propuesta experimental para permitir la definición de campos de clase privados utilizando un # prefijo hash .

Sintaxe

```
class ClassWithPrivateField {
  #privateField;
}

class ClassWithPrivateMethod {
  #privateMethod() {
    return "hello world";
  }
}

class ClassWithPrivateStaticField {
```

```
  static #PRIVATE_STATIC_FIELD;
}
```

Los getters y setters privados

```
class Caja {
    #contenido;

    constructor(cont) {
        this.#contenido = cont;
    }

    // Getter privado para #contenido
    get #contenido() {
        return this.#contenido;
    }

    // Setter privado para #contenido
    set #contenido(nuevoContenido) {
        this.#contenido = nuevoContenido;
    }
}
```

Exemplo

```
class Persona {
    #nombre;
    #edad;
    #mote;

    constructor(nombre, edad) {
        this.#nombre = nombre;
        this.#edad = edad;
        this.#mote=nombre+edad;
    }

    // Getter para nombre
    get nombre() {
        return this.#nombre;
    }

    // Setter para nombre
    set nombre(nuevoNombre) {
        this.#nombre = nuevoNombre;
    }

    // Getter para edad
    get edad() {
        return this.#edad;
    }

    // Setter para edad
    set edad(nuevaEdad) {
        if (nuevaEdad < 0) {
```

```
            throw new Error("La edad no puede ser negativa.");
        }
        this.#edad = nuevaEdad;
    }

    // Método que devuelve una descripción de la persona
    descripcion() {
        return `Nombre: ${this.#nombre}, Edad: ${this.#edad}`;
    }
}

// Uso de la clase
const persona = new Persona("Juan", 28);
console.log(persona.descripcion()); // Muestra: Nombre: Juan, Edad: 28

// Modificando propiedades con setters
persona.nombre = "Ana";
persona.edad = 30;
console.log(persona.descripcion()); // Muestra: Nombre: Ana, Edad: 30

// Accediendo a propiedades con getters
console.log(persona.nombre); // Muestra: Ana
console.log(persona.edad);   // Muestra: 30

// Intentando acceder directamente a propiedades privadas (esto producirá un error)
// console.log(persona.#nombre); // Error: Syntax error
```

2. JSON

El objeto JSON contiene métodos para analizar JavaScript Object Notation (JSON) y convertir valores a JSON.

2.1 JavaScript Object Notation

JSON es una sintaxis para serializar objetos, arrays, números, cadenas, booleanos y nulos. Está basado sobre sintaxis JavaScript pero es diferente a ella: algo JavaScript no es JSON, y algo JSON no es JavaScript. Mira también: JSON: The JavaScript subset that isn't.

Tipo JavaScript	Diferencia JSON
Objetos y Arrays	**Los nombres de las propiedades deben tener doble comilla**; las comas finales están prohibidas.
Números	**Los ceros a la izquierda están prohibidos**; un punto decimal debe ser seguido al menos por un dígito.
Cadenas	Solo un limitado conjunto de caracteres pueden ser de escape; **ciertos caracteres de control estan prohibidos**; los caracteres de separador de linea Unicode (U+2028) y el separador de parrafo (U+2029) son permitidos; las cadenas deben estar entre comillas dobles.

El objeto JSON no es soportado por navegadores antiguos.

A continuación móstrase unha táboa resumo dalgúns elementos a ter en conta nas conversións a JSON.

String en JavaScript	Resultado JSON	Descripción
`"null"`	`null`	Representa a ausencia de valor.
`"Infinity"`	`null`	`Infinity` non é válido en JSON, convértese en `null`.
`"NaN"`	`null`	`NaN` (Not a Number) tamén se converte en `null` en JSON.
`"undefined"`	non incluído	`undefined` non é válido en JSON e non se incluirá na saída.

String en JavaScript	Resultado JSON	Descripción
`"0"`	`0`	Un número representado como string convértese no número correspondente.
`"\"0\""`	`"0"`	Unha cadea que contén un número queda como cadea.

2.2 Método JSON.parse()

El método JSON.parse() analiza una cadena de texto como JSON, transformando opcionalmente el valor producido por el análisis.

Sintaxis

```
JSON.parse(text[, reviver])
```

Parámetros

- `text`: El texto que se convertirá a JSON.
- `reviver` {opcional}
    - : La función pasada indica cómo se transforma el valor original por el parsing.

Returns

Retorna el objeto que se corresponde con el texto JSON entregado.

Exceptions

Lanza una excepción "SyntaxError" si la cadena a transformar no es un JSON válido.

Ejemplos

Ejemplo: Usando `JSON.parse()`

```
JSON.parse("{}"); // {}
JSON.parse("true"); // true
JSON.parse('"foo"'); // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse("null"); // null
```

Ejemplo: `Usando el parámetro reviver`

Supongamos que tienes una cadena JSON que incluye fechas en formato de cadena y **quieres convertirlas en objetos de tipo Date** de JavaScript cuando parses el JSON.

```
const jsonData = '{"meetingDate": "2023-05-08T15:00:00Z"}';

const data = JSON.parse(jsonData, (key, value) => {
  // Revisa si la clave es 'meetingDate' y convierte su valor a objeto Date
  if (key === 'meetingDate') {
    return new Date(value);
  }
  return value;
});
```

```
console.log(data.meetingDate); // Muestra un objeto Date: 2023-05-08T15:00:00.000Z
console.log(data.meetingDate instanceof Date); //true
```

Cómo funciona:

- **El JSON inicial:** Es una cadena que representa un objeto con una propiedad meetingDate que es una fecha en formato de cadena.
- **Función reviver:** La función que pasas como segundo argumento de JSON.parse se llama reviver. Esta función se llama para cada clave y valor del objeto. En este caso, cuando la función encuentra la clave meetingDate, convierte el valor asociado (una cadena de fecha) en un objeto Date.
- **Resultado:** El objeto data resultante tiene una propiedad meetingDate que es un objeto Date, no una cadena.

2.3 Método JSON.stringify()

El método JSON.stringify() convierte un objeto o valor de JavaScript en una cadena de texto JSON, opcionalmente reemplaza valores si se indica una función de reemplazo, o si se especifican las propiedades mediante un array de reemplazo.

Sintaxis

```
JSON.stringify(value[, replacer[, space]])
```

Parámetros

- `value`
  - : El valor que será convertido a una cadena JSON.
- `replacer`{Opcional}
  - : Una función que altera el comportamiento del proceso de conversión a cadena de texto, o un array de objetos . Si este valor es **null** o **no se define**, todas las propiedades del objeto son incluidas en la cadena de texto JSON resultante.
- `space`{{Optional_Inline}}
  - : Un objeto de tipo String o Number que se utiliza para **insertar un espacio en blanco dentro de la cadena de salida JSON para mejorar su legibilidad**.Si es de tipo `Number`, indica el número de espacios a usar como espacios en blanco; este **número está limitado a 10** (si es mayor, el valor es sólo `10`). Los valores inferiores a 1 indican que no se deben utilizar espacios.Si es de tipo `String`, la cadena de texto (o sus 10 primeros caracteres, si es mayor) se utiliza como espacios en blanco. Si este parámetro no se define o es "null", no se utilizará ningún espacio en blanco.

Valor devuelto

Una cadena de texto JSON que representa el valor dado.

Excepciones

Lanza una excepción "TypeError ("cyclic object value") cuando encuentra una referencia circular.

Descripción

`JSON.stringify` convierte un valor a notación JSON representándolo:

- Si el valor tiene un método toJSON(), es responsable de definir qué será serializado.
- Los objetos "Boolean" , "Number" , and "String" se convierten a sus valores primitivos, de acuerdo con la conversión semántica tradicional.
- Si durante la conversión se encuentra un "undefined", una "Function" o un Symbol" se omite (cuando se encuentra en un objeto) o se censura a "null" (cuando se encuentra en un array). `JSON.stringify()` puede devolver `undefined` cuando

se pasan valores "puros" como `JSON.stringify(function(){}` o `JSON.stringify(undefined)`.
- Las instancias de "Date" implementan la función `toJSON()` devolviendo una cadena de texto (igual que `date.toISOString()`). Por lo que son tratadas como strings.
- Los números Infinity" y "NaN", así como el valor "null" , se consideran `null`.
- El resto de instancias de "Object" (incluyendo "Map" , "Set" , "WeakMap" , y "WeakSet") sólo tendrán serializadas sus propiedades enumerables.

```
//1. Serializar un objeto simple

const person = {
  name: "John",
  age: 30,
  city: "New York"
};

//2. Excluir propiedades específicas
//Utilizar un array para especificar solo las propiedades que deseas serializar.


const person = {
  name: "Alice",
  age: 25,
  city: "Los Angeles"
};

const json = JSON.stringify(person, ["name", "city"]);
console.log(json); // {"name":"Alice","city":"Los Angeles"}

//3. Transformar valores durante la serialización
// Utilizar una función de reemplazo para modificar los valores mientras se serializan.


const person = {
  name: "Bob",
  age: 20,
  birthYear: 2003
};

function replacer(key, value) {
  if (key === "birthYear") {
    return 2023 - value; // Calcula la edad a partir del año de nacimiento
  }
  return value;
}

const json = JSON.stringify(person, replacer);
console.log(json);
// {"name":"Bob","age":20,"birthYear":20}


//4. Formatear la salida
//Utilizar el tercer argumento de JSON.stringify() para añadir espacios e indentación a
la cadena JSON resultante, haciéndola más legible.
const data = {
  id: 1,
  title: "Example",
  completed: false
};

const json = JSON.stringify(data, null, 4);
console.log(json);
```

```
/*
{
    "id": 1,
    "title": "Example",
    "completed": false
}
*/

//5. Serializar fechas
//Las fechas son automáticamente convertidas a su representación en cadena ISO cuando se serializan.
const meeting = {
  subject: "Team Sync",
  time: new Date()
};

const json = JSON.stringify(meeting);
console.log(json); // {"subject":"Team Sync","time":"2023-08-01T12:34:56.789Z"}
```

3. Módulos

Son el estándar oficial para trabajar con módulos en JavaScript moderno y son soportados nativamente en la mayoría de los navegadores modernos y en entornos como Node.js (a partir de la versión 12, con algunas configuraciones).

- Exportar: Para hacer que funciones, objetos o primitivas estén disponibles fuera del módulo, se utilizan las declaraciones export.
- Importar: Para acceder a las funcionalidades de otro módulo, se utiliza la declaración import.

3.0 Motivos para emplear módulos.

1. Encapsulación: Los módulos ayudan a evitar la contaminación del espacio de nombres global y permiten que los desarrolladores mantengan piezas de código privadas, excepto aquellas que deciden exportar. **2. Reutilización:** Los módulos son reutilizables en diferentes partes de una aplicación o incluso en diferentes proyectos. **3. Mantenibilidad:** dAl estar organizado en módulos, el código es más fácil de mantener y actualizar.

3.1 `export`

La declaración export se utiliza al crear módulos de JavaScript para exportar funciones, objetos o tipos de dato primitivos del módulo para que puedan ser utilizados por otros programas con la sentencia import.

Sintaxis

```
export { name1, name2, …, nameN };
export { variable1 as name1, variable2 as name2, …, nameN };
export let name1, name2, …, nameN; // también var
export let name1 = …, name2 = …, …, nameN; // también var, const
export function FunctionName(){...}
export class ClassName {...}

export default expression;
export default function (…) { … } // también class, function*
export default function name1(…) { … } // también class, function*
export { name1 as default, … };

export * from …;
```

```
export { name1, name2, …, nameN } from …;
export { import1 as name1, import2 as name2, …, nameN } from …;
export { default } from …;
```

Ejemplo export

Definimos os seguintes ficheiros:

```
// archivo: mensaje.js
export function saludar(nombre) {
    return `Hola, ${nombre}!`;
}

// archivo: app.js
import { saludar } from './mensaje.js';

console.log(saludar('Ana'));  // Output: Hola, Ana!
```

Archivo index.html que hace la referencia al módulo. En este caso hay que indicar en la etiqueta script el tipo módulo (`<script type="module">`).

```
  <script type="module">
        import { saludar } from './mensaje.js';

        document.body.innerHTML = saludar('Ana'); // Utiliza la función importada
    </script>
```

Debido a las **políticas de seguridad** de los navegadores modernos (específicamente la política del mismo origen), los módulos JavaScript no funcionarán si simplemente abres el archivo index.html directamente en un navegador desde tu sistema de archivos local (file://). **Necesitas servir tus archivos a través de un servidor HTTP**. Puedes hacer esto de varias maneras:

```
php -S localhost:9090

python3 -m http.server 9099
```

Fichero index.html

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script type="module">
        import { saludar } from './mensaje.js';
```

```
        document.body.innerHTML = saludar('Ana'); // Utiliza la función importada
    </script>
</body>
</html>
```

Ejemplo export con nombre

Fíjate en la última linea donde se definen los elementos "exportables" del fichero.

```
// module "my-module.js"
function cube(x) {
  return x * x * x;
}
const foo = Math.PI + Math.SQRT2;
var graph = {
  options: {
    color: "white",
    thickness: "2px",
  },
  draw: function () {
    console.log("From graph draw function");
  },
};
//Elementos que serán visibles
export { cube, foo, graph };
```

Para luego ser empleamos en otro fichero:

```
import { cube, foo, graph } from "my-module";
graph.options = {
  color: "blue",
  thickness: "3px",
};
graph.draw();
console.log(cube(3)); // 27
console.log(foo); // 4.555806215962888
```

3.2 `import`

La sentencia import se usa para importar funciones que han sido exportadas desde un módulo externo.

Sintaxis

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export } from "module-name";
import { export as alias } from "module-name";
import { export1 , export2 } from "module-name";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
```

Importa todo el contenido del módulo

Esto inserta myModule en el ámbito actual, que contiene todos los elementos exportados en el archivo ubicado en /modules/my-module.js.

```
import * as myModule from "/modules/my-module.js";
```

Importa un solo miembro de un módulo

```
import { myExport } from "/modules/my-module.js";
```

Importa un miembre con un alias

```
import { reallyReallyLongModuleExportName as shortName } from "/modules/my-module.js";
```

4. ¿Qué es CORS?

CORS (Cross-Origin Resource Sharing) es un mecanismo o política de seguridad que permite controlar las peticiones HTTP asíncronas que se pueden realizar desde un navegador a un servidor con un dominio diferente de la página cargada originalmente. Este tipo de peticiones se llaman peticiones de origen cruzado (cross-origin).

Por defecto, los navegadores permiten enlazar hacia documentos situados en todo tipo de dominios si lo hacemos desde el HTML o desde Javascript utilizando la API DOM (que a su vez está construyendo un HTML). Sin embargo, no ocurre lo mismo cuando se trata de peticiones HTTP asíncronas mediante Javascript (AJAX), sea a través de XMLHttpRequest, de fetch o de librerías similares para el mismo propósito.

Utilizando este tipo de peticiones asíncronas, los recursos situados en dominios diferentes al de la página actual no están permitidos por defecto. Es lo que se suele denominar protección de CORS. Su finalidad es dificultar la posibilidad de añadir recursos ajenos en un sitio determinado.

4.1 Access-Control-Allow-Origin

Como hemos comentado, las peticiones HTTP asíncronas de origen cruzado no están permitidas, pero existen formas de permitirlas. La más básica, probablemente, sea la de **incluir una cabecera Access-Control-Allow-Origin** en la respuesta de la petición, donde debe indicarse el dominio al que se le quiere dar permiso:

```
Access-Control-Allow-Origin: https://domain.com/
```

De esta forma, el navegador comprobará dichas cabeceras y si coinciden con el dominio de origen que realizó la petición, esta se permitirá. En el ejemplo anterior, la cabecera tiene el valor https://domain.com/, pero en algunos casos puede interesar indicar el valor *.

5. El objeto XMLHttpRequest

6. fetch().

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

El modo más habitual de manejar las promesas es utilizando .then(), aunque también se puede utilizar **async/await**. Esto se suele reescribir de la siguiente forma, que queda mucho más simple y evitamos constantes o variables temporales de un solo uso:

```
fetch("/robots.txt")
  .then(function(response) {
    /** Código que procesa la respuesta **/
  });
```

Una petición básica de fetch es realmente simple de realizar.

```
<!DOCTYPE html>
<html lang="gl">
<head>
<meta charset="UTF-8">
<title>Fetch API Example</title>
</head>
<body>
<script>
  fetch('https://api.sampleapis.com/futurama/info')
    .then(response => {
      if (!response.ok) {
          throw new Error('Network response was not ok');
      }
      return response.json();
    })
    .then(data => console.log(data))
    .catch(error => console.error('There was a problem with the fetch operation:',
error));
</script>
</body>
</html>
```

A continuación vamos intentar mudar o exemplo para que mostre a información recuperada nunha táboa. El flujo del código es:

1. El script hace una solicitud a la API y procesa la respuesta como antes.
2. En lugar de imprimir los datos en la consola, este script inserta filas en la tabla. Para cada elemento en los datos, crea una fila y tres celdas correspondientes a los encabezados.
3. Llena cada celda con los datos apropiados: título, temporada y directores (este último se maneja como un array, así que se unen los nombres con comas).
4. Manejo de errores: Captura y muestra en la consola cualquier error que pueda ocurrir durante la operación de fetch.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Fetch API Table Display</title>
<style>
  table {
    width: 100%;
    border-collapse: collapse;
  }
  th, td {
    border: 1px solid black;
    padding: 8px;
    text-align: left;
```

```
  }
</style>
</head>
<body>
<h1>Información de Futurama</h1>
<table id="infoTable">
  <thead>
    <tr>
      <th>Años</th>
      <th>Descripción</th>
    </tr>
  </thead>
  <tbody>
  </tbody>
</table>

<script>
  fetch('https://api.sampleapis.com/futurama/info')
    .then(response => {
      if (!response.ok) {
          throw new Error('Network response was not ok');
      }
      return response.json();
    })
    .then(data => {
      const table = document.getElementById('infoTable').getElementsByTagName('tbody')
[0];
      data.forEach(item => {
        let row = table.insertRow();
        let cell1 = row.insertCell(0);
        let cell2 = row.insertCell(1);
        //Campos do json peticionado
        cell1.textContent = item.yearsAired;
        cell2.textContent = item.synopsis;
        //Tamén ten outro array no interior con datos dos directores:
        //item.creators
      });
    })
    .catch(error => console.error('There was a problem with the fetch operation:',
error));
</script>
</body>
</html>
```

6.1 Opciones de fetch()

A la función fetch(), al margen de la url a la que hacemos petición, se le puede pasar un segundo parámetro de opciones de forma opcional, un con opciones de la petición HTTP:

```
const options = {
  method: "GET"
};

fetch("/robots.txt", options)
  .then(response => response.text())
  .then(data => {
    /** Procesar los datos **/
  });
```

Opción	Tipo	Descripción
`method`	`string`	El método HTTP a utilizar, como `GET`, `POST`, `PUT`, `DELETE`, etc.
`headers`	`Headers`	Encabezados HTTP para enviar con la solicitud. Se pueden usar objetos de tipo `Headers` o un objeto literal con pares clave-valor.
`body`	`Blob`/`BufferSource`/`FormData`/`URLSearchParams`/`ReadableStream`/`string`	El cuerpo de la solicitud. No es permitido en solicitudes `GET` o `HEAD`.
`mode`	`string`	El modo de la solicitud, que puede ser `cors`, `no-cors`, `same-origin`, o `navigate`. Esto determina la política de CORS que se aplicará a la solicitud.
`credentials`	`string`	Controla las credenciales de la solicitud. Puede ser `omit`, `same-origin` o `include` para controlar el envío de cookies y datos de autenticación.
`cache`	`string`	Controla cómo debe interactuar la solicitud con el caché del navegador. Puede ser `default`, `no-store`, `reload`, `no-cache`, `force-cache`, o `only-if-cached`.

Opción	Tipo	Descripción
`redirect`	`string`	Controla qué hacer en caso de redirecciones. Puede ser `follow`, `error`, o `manual`.
`referrer`	`string`	Un referente para la solicitud. Puede ser un URL completo, `client`, o un string vacío.
`referrerPolicy`	`string`	Controla el envío de información del referente. Opciones incluyen `no-referrer`, `no-referrer-when-downgrade`, `origin`, `origin-when-cross-origin`, `same-origin`, `strict-origin`, `strict-origin-when-cross-origin`, y `unsafe-url`.
`integrity`	`string`	Contiene información de integridad subresource para la solicitud (por ejemplo, para verificar un hash de contenido de scripts y estilos).
`keepalive`	`boolean`	Indica que la solicitud puede mantenerse viva incluso cuando la página se cierre, útil para logs y analytics.
`signal`	`AbortSignal`	Se puede usar para abortar la solicitud. Se pasa un objeto `AbortSignal`.

Opción	Tipo	Descripción
`window`	`WindowProxy`	Este campo solo debería ser usado si estás trabajando con `ServiceWorkers`. Por defecto es `null` y no debe usarse en contexto de ventana.

6.2 headers

Los encabezados (headers) en una solicitud fetch son utilizados para enviar **información adicional** al servidor. Esta información puede incluir el tipo de contenido que se está enviando (como JSON, formulario, etc.), tokens de autenticación, instrucciones sobre cómo el servidor debe procesar la solicitud, y más.

Ejemplo

```
fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'GET', // Método HTTP
    headers: {
        'Content-Type': 'application/json' // Tipo de contenido
    }
})
.then(response => response.json()) // Procesar la respuesta como JSON
.then(data => console.log(data)) // Acciones a realizar con los datos
.catch(error => console.error('Error:', error)); // Manejo de errores
```

Header	Descripción
`Content-Type`	Indica el tipo de contenido del cuerpo de la solicitud.
`Authorization`	Contiene las credenciales para autenticar al usuario.
`Accept`	Indica los tipos de contenido que el cliente puede procesar.
`User-Agent`	Identifica el agente del usuario que hace la solicitud.
`Accept-Language`	Indica el idioma preferido del cliente.
`Cache-Control`	Directivas para los mecanismos de caching en la respuesta.
`Cookie`	Envía las cookies del cliente al servidor.

6.3 Cuerpo de la Solicitud (Body)

En JavaScript, cuando se utiliza la función fetch para realizar solicitudes HTTP, el término "body" se refiere **a los datos que se envían en el cuerpo de la solicitud**. Estos datos pueden ser cualquier tipo de información que necesites enviar al servidor, como formularios, archivos, JSON, etc. El cuerpo de la solicitud es opcional y se utiliza principalmente para métodos como POST, PUT o PATCH, donde estás enviando datos al servidor para que los procese.

Ejemplo con el esquema de la petición:

```
fetch('https://example.com/api/data', {
    method: 'POST', // Método HTTP
    headers: {
        'Content-Type': 'application/json'
```

```
    },
    body: JSON.stringify({
        key: 'value',
        anotherKey: 'anotherValue'
    })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Ejemplo real que retorna los valores enviados. Aquí, estamos enviando un objeto con tres propiedades: title, body, y userId.

```
fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST', // Método HTTP
    headers: {
        'Content-Type': 'application/json', // Indicando que el cuerpo de la solicitud es
un JSON
    },
    body: JSON.stringify({
        title: 'Ejemplo de título',
        body: 'Este es el cuerpo del post.',
        userId: 1
    }) // Datos que se envían al servidor en formato JSON
})
.then(response => response.json()) // Convierte la respuesta a JSON
.then(data => console.log(data)) // Maneja los datos de la respuesta
.catch(error => console.error('Error:', error)); // Captura y muestra errores
```

6.4 ¿Qué son async y await?

- **async**: La palabra clave async se utiliza para declarar una **función como asíncrona**. Esta declaración permite que **dentro de la función se utilice await**, y cambia la forma en que funciona la función: en lugar de devolver un valor directamente, devuelve una promesa.
- **await**: La palabra clave await se utiliza para **pausar la ejecución de la función asíncrona** y esperar la resolución de una Promesa. **Se utiliza principalmente para esperar el resultado de una llamada asíncrona** como una solicitud de red o una operación de base de datos. await solo puede ser usado dentro de funciones declaradas con async.

Ejemplo de fetch con async/await**

```
//ASYNC : : Declara una función asíncrona llamada enviarDatos.
async function enviarDatos() {
    try {
      //Aquí, await pausa la ejecución de la función hasta que la Promesa devuelta por
fetch se resuelva.
        const response = await fetch('https://jsonplaceholder.typicode.com/posts', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({
                title: 'Ejemplo de título',
                body: 'Este es el cuerpo del post.',
                userId: 1
            })
        });
```

```
        //Una vez que la solicitud ha completado y tenemos la respuesta, convertimos el
cuerpo de la respuesta a JSON, también de forma asíncrona, esperando a que esta operación
se complete.
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
}

enviarDatos();
```

Ventajas de usar async/await

- **Claridad y legibilidad**: El código se lee de una manera más lineal y es más fácil de entender, especialmente en flujos complejos de lógica asíncrona.
- **Manejo de errores**: El uso de bloques try/catch permite un manejo de errores más intuitivo y directo comparado con las cadenas de .then() y .catch().

En resumen, async/await hace que el código asíncrono se vea y se comporte un poco más como código síncrono tradicional, facilitando su lectura y mantenimiento.