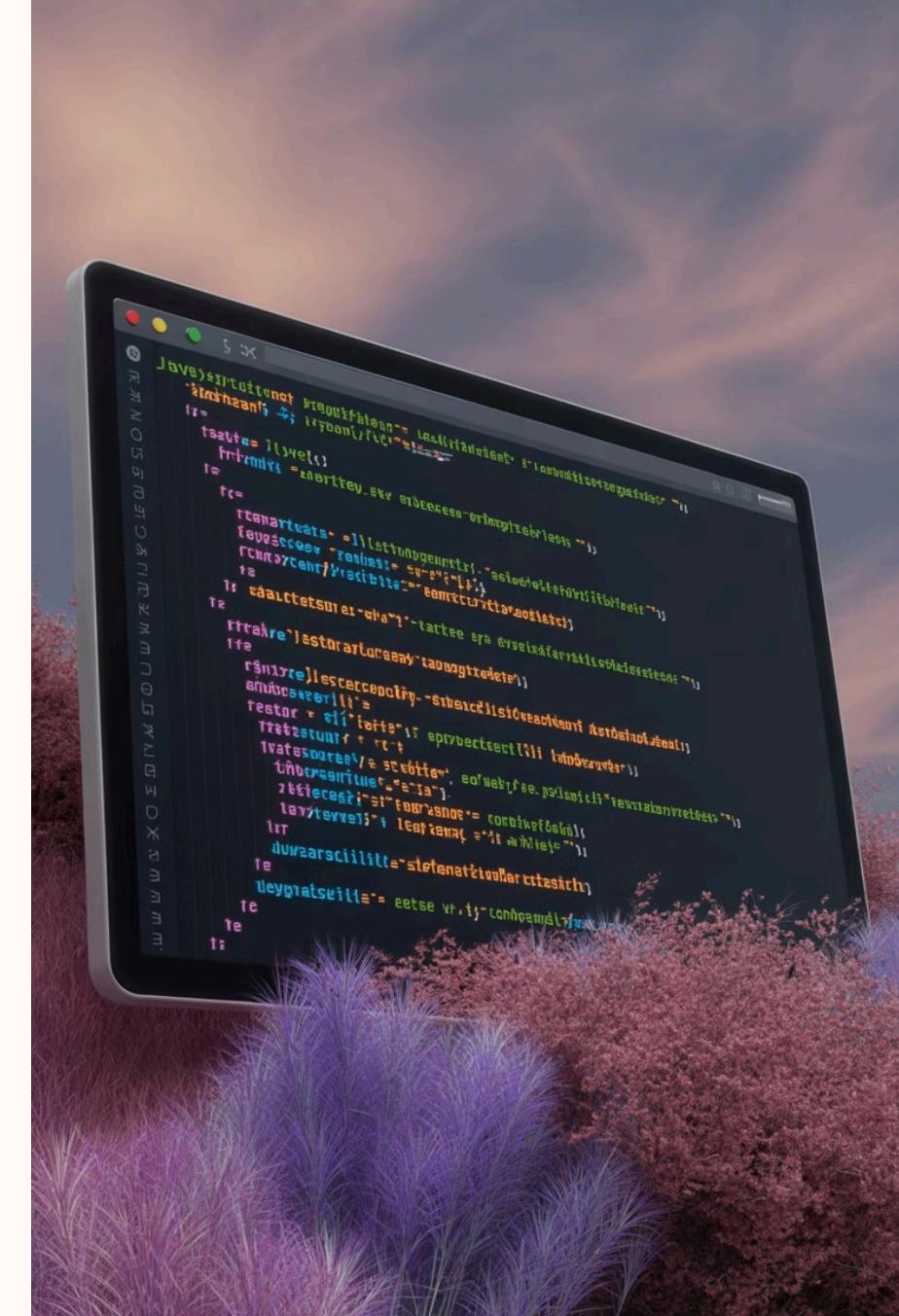


JavaScript Avanzado: APIs, Storage y Manipulación de Datos

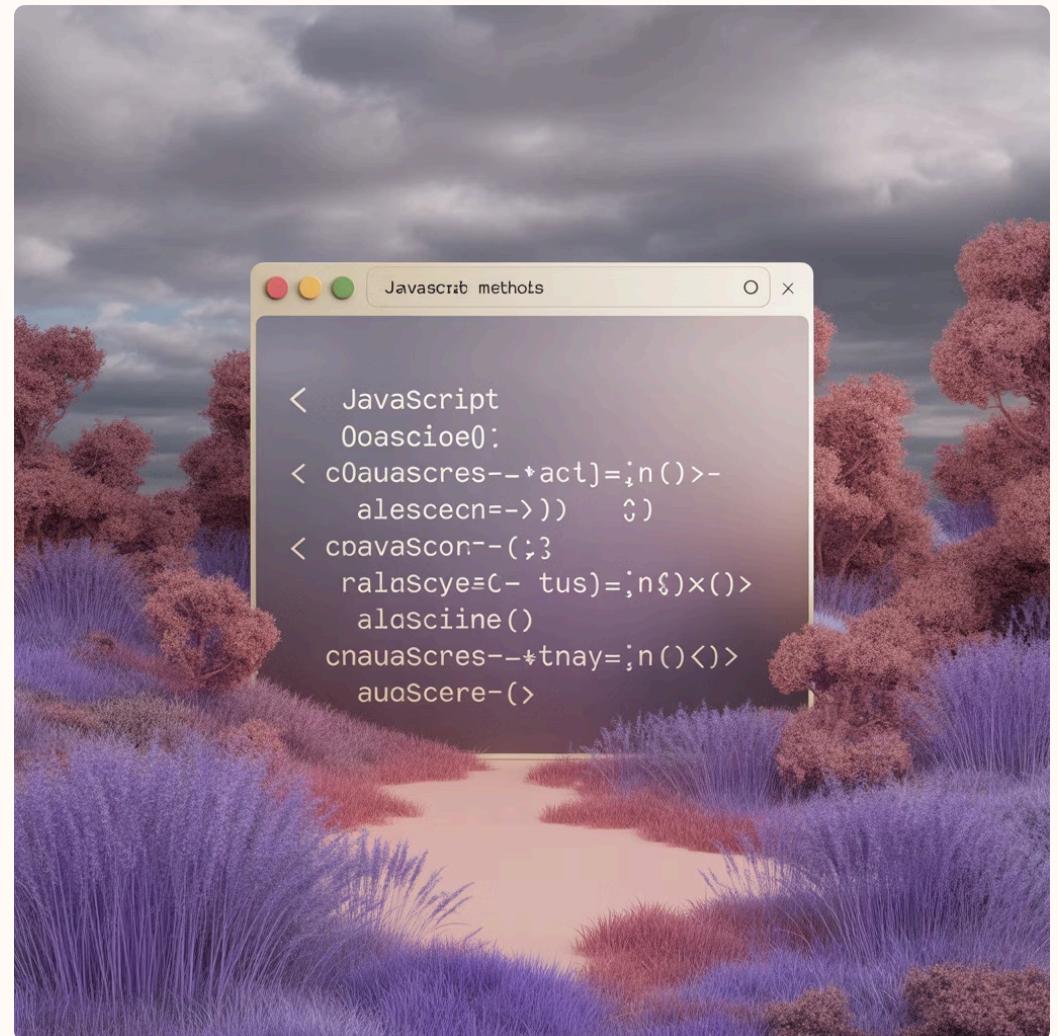
Domina las herramientas esenciales para crear aplicaciones web dinámicas e interactivas. En esta sesión exploraremos técnicas avanzadas de manipulación de arrays, gestión de almacenamiento local, y comunicación con APIs externas.



Métodos Avanzados de Arrays

Los arrays en JavaScript ofrecen métodos poderosos que transforman cómo procesamos datos. Estos métodos nos permiten filtrar, transformar y reducir colecciones de información de manera elegante y eficiente.

Dominar estos métodos es fundamental para trabajar con datos de APIs, como los productos que consumiremos en nuestro proyecto práctico.



map()

Transforma cada elemento del array y devuelve un nuevo array con los resultados.

```
products.map(p => p.title)
```

filter()

Crea un nuevo array con elementos que cumplan una condición específica.

```
products.filter(p => p.price > 50)
```

reduce()

Reduce el array a un único valor acumulando resultados.

```
products.reduce((sum, p) => sum +  
p.price, 0)
```



Operaciones Esenciales con Arrays

Más allá de los métodos de transformación, JavaScript ofrece herramientas específicas para buscar, ordenar y manipular arrays. Estas operaciones son cruciales cuando trabajamos con listados de productos o cualquier colección de datos.

1

find() y findIndex()

Localiza el primer elemento que cumple una condición o su posición en el array.

```
const producto = products.find(p => p.id === 5)
```

2

sort()

Ordena los elementos del array según un criterio personalizado.

```
products.sort((a, b) => a.price - b.price)
```

3

forEach()

Ejecuta una función para cada elemento, ideal para renderizar HTML.

```
products.forEach(p => container.innerHTML += createCard(p))
```

4

some() y every()

Verifican si alguno o todos los elementos cumplen una condición.

```
products.some(p => p.rating.rate > 4.5)
```

Manipulación de Strings en JavaScript

Los strings son fundamentales cuando trabajamos con datos de APIs. Necesitamos formatear títulos, validar entradas, extraer información y crear contenido dinámico para nuestra interfaz.

JavaScript proporciona métodos potentes para manipular cadenas de texto de forma eficiente y elegante.

Template Literals

Permiten crear strings dinámicos con interpolación de variables y expresiones.

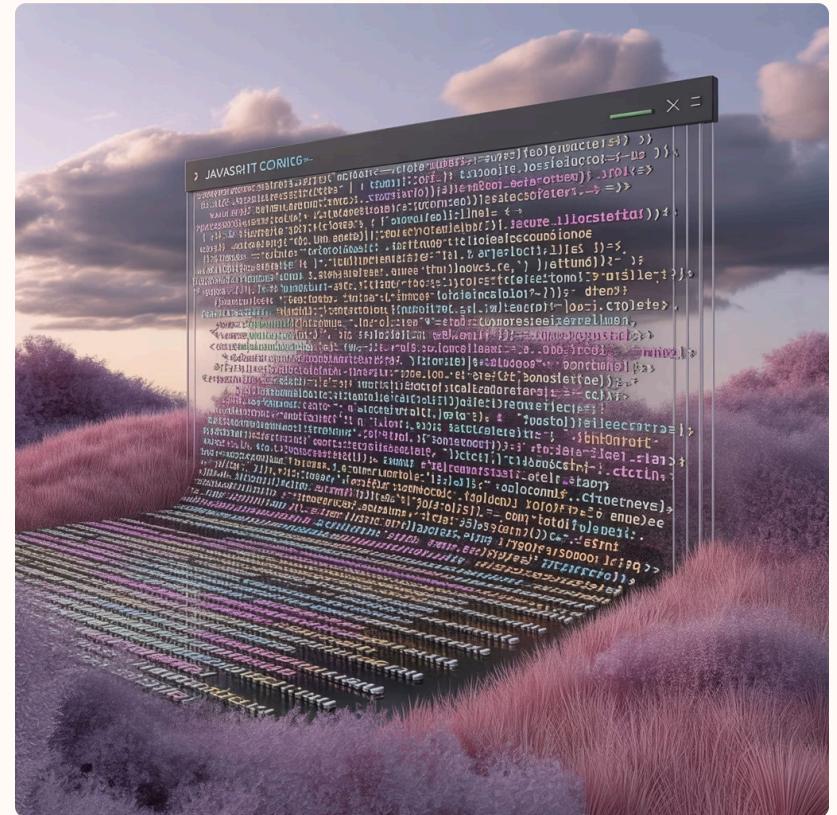
```
`Precio: ${product.price}`
```

Métodos de transformación

toUpperCase(), toLowerCase(), trim(), slice(), substring() y replace() para modificar strings.

Búsqueda y validación

includes(), startsWith(), endsWith() y indexOf() para verificar contenido.



Fetch API: Consumiendo Datos Externos

La Fetch API es la forma moderna de realizar peticiones HTTP en JavaScript. Nos permite comunicarnos con servidores externos, consumir APIs REST y obtener datos dinámicamente para nuestras aplicaciones.

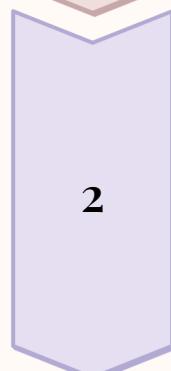


Realizar petición

1

Iniciamos la solicitud HTTP con `fetch()` pasando la URL del endpoint.

```
fetch('https://fakestoreapi.com/products')
```



Procesar respuesta

2

Convertimos la respuesta a JSON usando el método `.json()` que retorna una Promise.

```
.then(response => response.json())
```



Usar los datos

3

Accedemos a los datos obtenidos y los procesamos según nuestras necesidades.

```
.then(data => renderProducts(data))
```

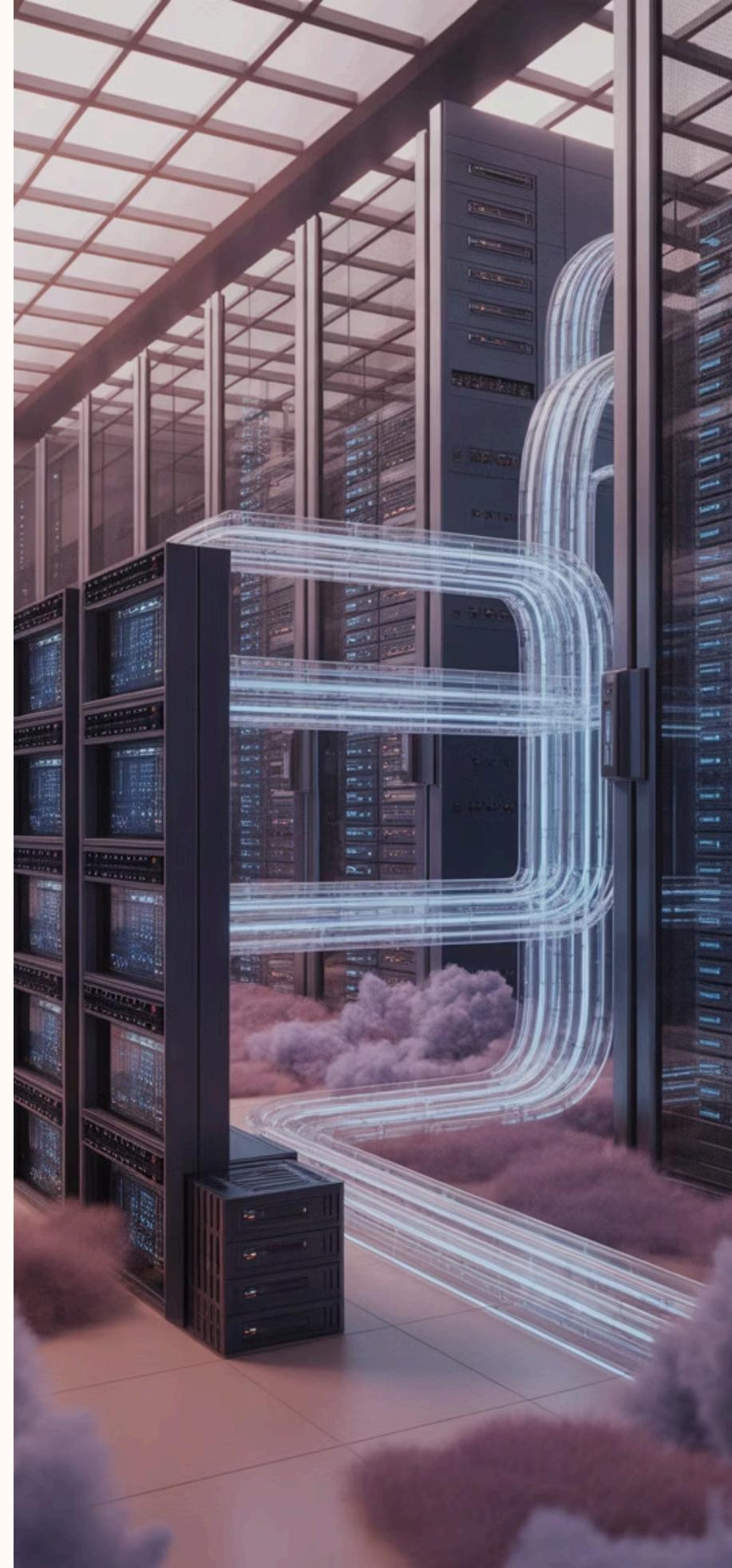


Manejo de errores

4

Capturamos posibles errores de red o procesamiento con `.catch()`

```
.catch(err => console.error(err))
```



Métodos Abstractos de Promises

Las Promises tienen métodos estáticos poderosos que nos permiten gestionar múltiples operaciones asíncronas simultáneamente. Estos métodos son esenciales cuando necesitamos coordinar varias peticiones a APIs o procesos paralelos.

Promise.all()

Espera a que **todas** las promesas se resuelvan. Si una falla, toda la operación falla.

```
Promise.all([fetch(url1), fetch(url2)])
  .then(([res1, res2]) => ...)
```

Ideal para cargar múltiples recursos simultáneamente.

Promise.any()

Se resuelve con la **primera promesa exitosa**. Ignora los fallos hasta que todas fallen.

```
Promise.any([fetch(backup1), fetch(backup2)])
  .then(firstSuccess => ...)
```

Perfecto para APIs con múltiples endpoints de respaldo.

Promise.race()

Se resuelve con la **primera** promesa que se complete, ya sea exitosa o fallida.

```
Promise.race([fetchAPI1(), fetchAPI2()])
  .then(fastestResult => ...)
```

Útil para timeouts o usar el servidor más rápido.

Promise.allSettled()

Espera a que **todas terminen**, sin importar el resultado (éxito o fallo).

```
Promise.allSettled([p1, p2, p3])
  .then(results => ...)
```

Ideal cuando necesitas el estado de todas las operaciones.

LocalStorage: Persistencia en el Navegador



LocalStorage permite almacenar datos en el navegador del usuario de forma permanente. Los datos persisten incluso después de cerrar el navegador, lo que lo hace perfecto para guardar preferencias, carritos de compra o información de productos seleccionados.

01

Guardar datos

Usa **setItem()** con una clave y valor. Los objetos deben convertirse a JSON.

```
localStorage.setItem('product',  
  JSON.stringify(productData))
```

02

Recuperar datos

Usa **getItem()** con la clave. Parsea el JSON de vuelta a objeto.

```
const product =  
  JSON.parse(localStorage.getItem('product'))
```

03

Eliminar datos

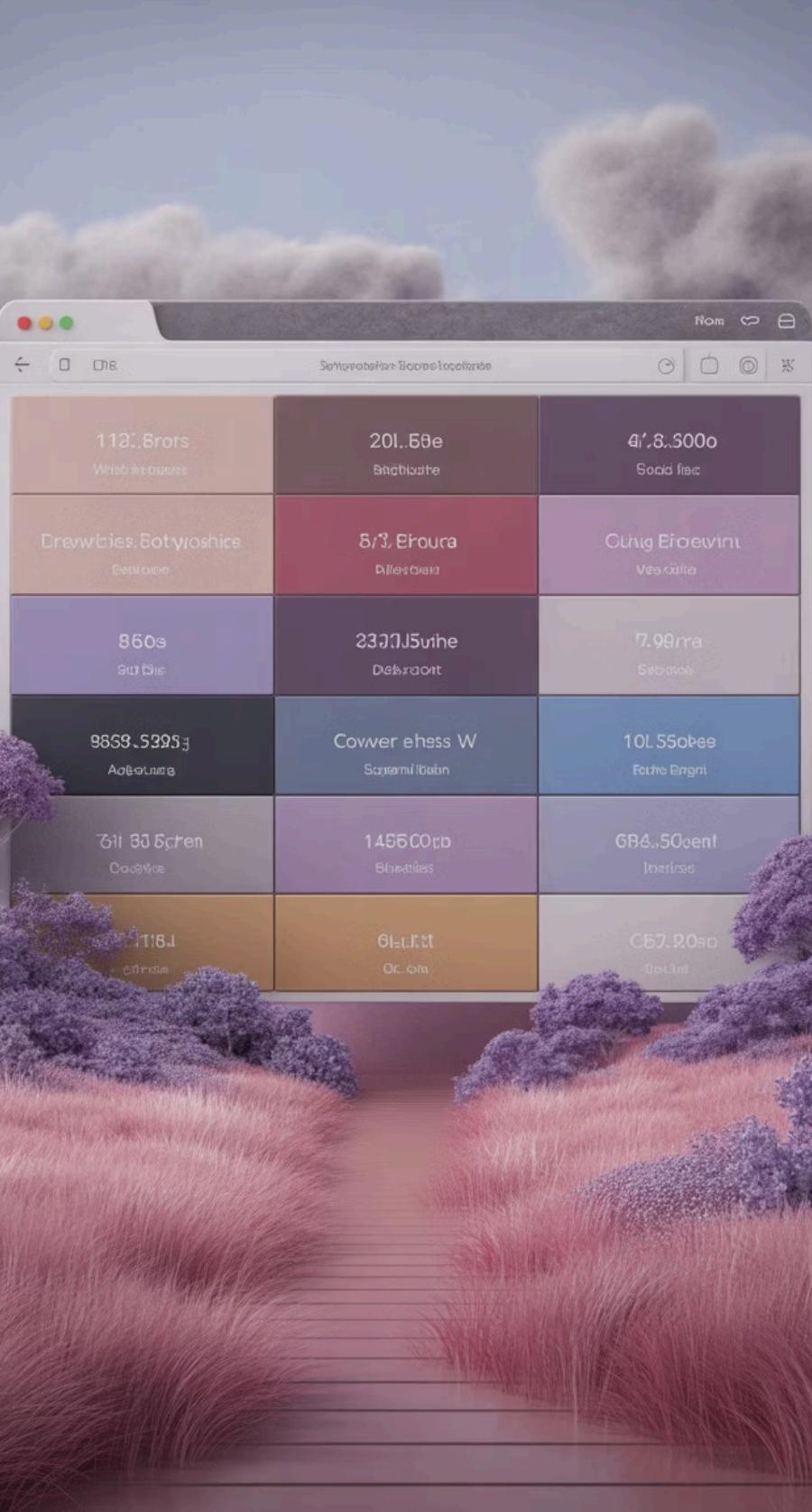
Usa **removeItem()** para borrar una clave o **clear()** para todo.

```
localStorage.removeItem('product')
```

- ❑ **Importante:** LocalStorage solo puede almacenar strings. Siempre usa **JSON.stringify()** para guardar objetos y **JSON.parse()** para recuperarlos.

SessionStorage vs LocalStorage

Ambos ofrecen almacenamiento en el navegador, pero con diferentes ciclos de vida. Elegir el correcto depende de cuánto tiempo necesitas conservar los datos y el contexto de tu aplicación.



LocalStorage

- Los datos **persisten indefinidamente** hasta que se borren explícitamente
- Compartido entre todas las pestañas y ventanas del mismo origen
- Capacidad: aproximadamente 5-10 MB
- Perfecto para: preferencias de usuario, carrito de compra, datos de sesión persistentes

```
localStorage.setItem('theme', 'dark')
```

SessionStorage

- Los datos **se borran al cerrar la pestaña** o ventana del navegador
- Aislado por pestaña: cada pestaña tiene su propio storage
- Capacidad: similar a localStorage
- Perfecto para: datos temporales, formularios en progreso, estado de navegación

```
sessionStorage.setItem('tempData', data)
```

Flujo Completo: De API a Detalles

Veamos cómo integrar todos estos conceptos en un flujo real de aplicación. Este patrón es fundamental en el desarrollo web moderno y representa cómo funcionan la mayoría de las aplicaciones e-commerce.

1. Consumir la API

Realizamos un fetch a la FakeStore API para obtener el listado completo de productos.

```
fetch('https://fakestoreapi.com/products')
    .then(res => res.json())
    .then(products => renderProducts(products))
```

3. Guardar en localStorage

Al hacer click, capturamos el producto seleccionado y lo guardamos en localStorage.

```
card.addEventListener('click', () => {
  localStorage.setItem('selectedProduct',
    JSON.stringify(product))
  window.location.href = 'product-details.html'
})
```

1

2

3

4

2. Iterar y renderizar

Usamos **forEach()** o **map()** para generar el HTML de cada producto y agregarlo al DOM.

```
products.forEach(product => {
  container.innerHTML += `<div class="card">
    
    <h3>${product.title}</h3>
    <p>${product.price}</p>
  </div>`})
```

4. Recuperar y mostrar detalles

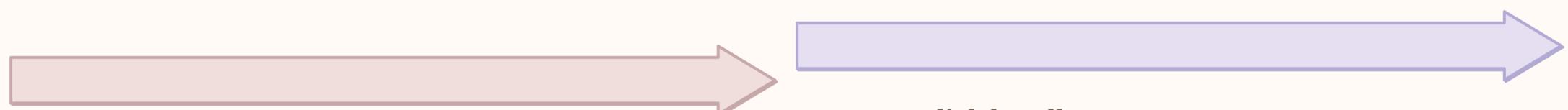
En la página de detalles, recuperamos el producto de localStorage y renderizamos la información completa.

```
const product = JSON.parse(
  localStorage.getItem('selectedProduct'))
document.getElementById('title').textContent =
  product.title
```

Trabajo Práctico Final

Tienda de Productos con FakeStore API

Es momento de aplicar todo lo aprendido. Crearás una tienda online completa que consume datos reales de una API, implementa navegación entre páginas y utiliza almacenamiento local para persistir información.



Paso 1: Listado de productos

consume la FakeStore API con fetch y renderiza todos los productos en cards HTML. Usa **forEach()** o **map()** para iterar.

```
fetch('https://fakestoreapi.com/products')
  .then(response => response.json())
  .then(data => console.log(data))
```

Paso 2: Click handler

Agrega un event listener a cada producto. Al hacer click, guarda la información completa del producto en **localStorage** usando **JSON.stringify()**.



Paso 3: Navegación

Después de guardar, navega programáticamente a **product-details.html** usando **window.location.href**.

Paso 4: Página de detalles

En **product-details.html**, recupera los datos de **localStorage** con **JSON.parse()** y renderiza toda la información: título, imagen, descripción, precio, rating, etc.

- Bonus:** Implementa filtros por categoría usando **filter()**, ordenamiento por precio con **sort()**, y un buscador con **find()**. ¡Sé creativo con el diseño!