UT06.1: Browser Object Model

Contenido

1. Introducción

Browser Object Model (BOM) es el un conjunto de objeto que nos permiten obtener información sobre el navegador que estamos utilizando. No se un estándar oficial, por lo que podemos encontrar diferencias en la implementación de estos objetos en cada navegador.

2. Política de seguridad mismo origen

Los navegadores implementan está política por la cual restringen cómo un documento o un script solamente puede interactuar con recursos que comparten el mismo **origen**, evitando con ello la interacción con scripts potencialmente maliciosos.

Dos recursos comparten el mismo origen, es decir, están ubicados en la misma máquina, si ambos coinciden en:

- **Protocolo**: Ambos recursos deben ser accedidos o utilizando HTTP o HTTPS.
- **Dominio**: El dominio debe ser el mismo para ambos recursos.
- **Puerto**: El número de puerto debe ser el mismo.

La ruta interna de acceso al recurso si que puede variar lógicamente.

Veamos un ejemplo de cómo podemos compartir recursos con la siguiente URL: `http://www.mitienda.com/ruta/index.html`.

De la URL obtenemos la siguiente información:

- Protocolo: HTTP.
- Dominio: www.mitienda.com
- Puerto: 80, al ser el puerto por defecto.

En la siguiente tabla hacemos la comparación con varios recursos.

URL	Resultado	Razón
`http://www.mitienda.com/dir/pagina.html`	Mismo origen	Solo varía la ruta
`http://www.mitienda.com/ruta/dir/index.html`	Mismo origen	Solo varía la ruta
`https://www.mitienda.com/secure.html`	Fallo	Diferente protocolo
`http://www.mitienda.com:81/dir/etc.html`	Fallo	Diferente puerto
`http://www.mitienda.es/ruta/index.html`	Fallo	Diferente dominio

Cumplir con la política de seguridad de mismo origen implica que debemos hacer uso de un **servidor web**. El protocolo `FILE` no permite cumplir con la política de seguridad, ya que cada recurso se muestra en su propio dominio.

En este documento haremos uso de **funcionalidades y objetos que hacen uso de la política de seguridad del mismo origen**, por para poder implementarlas **será necesario utilizar un servidor web**.

Para más información podemos consultar:

https://developer.mozilla.org/es/docs/Web/Security/Same-origin_politica

3. Objeto `window`

El objeto `window` representa la ventana del navegador. Este objeto contiene todas las variables y funciones globales que estamos definiendo en nuestra aplicación, además del resto de objetos predefinidos en el intérprete.

En el siguiente ejemplo vemos una sentencia típica de selección de un objeto utilizando DOM.

```
document.getElementById('message');
```

Podemos crear una sentencia equivalente utilizando el objeto `window`, aunque es el objeto es sobrentendido por el intérprete.

```
window.document.getElementById('message');
```

La referencia la puedes encontrar en: https://developer.mozilla.org/es/docs/Web/API/Window

3.1. Propiedades

Tenemos propiedades para comprobar las dimensiones de la ventana.

- `innerHeight`: Obtenemos la altura de la ventana del navegador.
- `innerWidth`: Obtenemos la anchura de la ventana del navegador.

Vamos a mostrar las dimensiones de la ventana del navegador.

```
console.log("Altura: " + window.innerHeight);
console.log("Anchura: " + window.innerWidth);
```

3.2. Métodos

Veamos algunos de los métodos que podemos utilizar con el objeto. Para ver los ejemplos, creamos una nueva página denominada `auxPage.html` y la dejamos en la raíz de la aplicación junto al `index.html`. La página tan solo contiene el banner de la página principal para poder interactuar con ella.

```
<!DOCTYPE html>
<html lang="es">

<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>Página auxiliar</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min
.css" rel="stylesheet"
        integrity="sha384-
T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN"
crossorigin="anonymous" />
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.11.1/font/bootstrap-icons.css">
    <link
href="https://fonts.googleapis.com/css?family=Open+Sans:300,400,600,700"
rel="stylesheet" />
    <!-- Estilos locales -->
    <link href="css/main.css" rel="stylesheet" />
</head>

<body>
    <h1 class="display-1">Página auxiliar</h1>
```

```
    <div class="banner">
        <div class="jumbotron jumbotron-fluid d-flex">
            <div
                class="text-dark container d-flex flex-column justify-
content-center align-items-sm-stretch align-items-md-center">
                <h1 id="message" class="col-sm-12">Obtén un
descuento</h1>
                <h2 id="discount">20%</h2>
                <p id="message2">Solo está temporada</p>
            </div>
        </div>
    </div>

    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/umd/popper.m
in.js"
        integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc2pM8ODewa9r"
        crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.min.j
s"
        integrity="sha384-
BBtl+eGJRgqQAUMxJ7pMwbEyER4l1g+O15P+16Ep7Q9Q+zqX6gSbd85u4mG4QzX+"
        crossorigin="anonymous"></script>
    <script src="https://code.jquery.com/jquery-3.7.1.min.js"
        integrity="sha256-/JqT3SQfawRcv/BIHPThkBvs0OEvtFFmqPF/lYI/Cxo="
crossorigin="anonymous"></script>

    <script src="./js/main.js"></script>
    <script src="./js/result.js"></script>
    <script src="./js/auxPage.js"></script>

</body>

</html>
```

3.2.1. Métodos `open()`, `close()` y `focus()`

Estos métodos permiten abrir, cerrar o pasar el foco a una ventana respectivamente.

Comenzamos con el método `open()`, el cual puede recibir tres parámetros:

- **URL del recurso**: Debemos indicar cuál será el recurso que queremos abrir.
- **Nombre de la ventana**: de manera opcional podemos darle un nombre a la ventana. Podemos usar este nombre para redirigir el tráfico de formularios y de enlaces utilizando el atributo `target`.

- **Opciones**: Se trata de un string donde una serie de pares nombre de propiedad valor separados por comas, donde configuramos las características de la ventana. No todos los navegadores implementan las mismas propiedades.

Vamos a agregar un botón en la página principal de nuestra aplicación que permita abrir la página auxiliar que hemos creado anteriormente. En el manejador de eventos ejecutamos el método `open()`. Como vemos le hemos pasado la URL del recurso y le hemos asignado un nombre a la nueva ventana. En cuanto a las opciones de la ventana, le hemos asignado las dimensiones y su posición, le decimos que nos muestra la barra de título, pero que no muestre la barra de herramientas ni el menú, tampoco muestra la URL del recurso cargado en la ventana. Por último, el método devuelve una referencia a la ventana que asignamos a una variable global que nos permita manipularla posteriormente.

```
let mywindow = null;
const examples = document.getElementById('examples');
const bOpen = document.createElement('button');
bOpen.classList.add('btn');
bOpen.classList.add('m-1');
bOpen.innerHTML = 'Abrir ventana';
bOpen.addEventListener('click', (event) => {
    mywindow = window.open('auxPage.html', 'Mywindow', 'width=800,
height=600, top=250, left=250, titlebar=yes, toolbar=no, menubar=no,
location=no');
});
examples.append(bOpen);
```

Referencia: https://developer.mozilla.org/es/docs/Web/API/Window/open

Si observamos la nueva ventana, cada vez que cliqueamos el botón estamos recargando el contenido. Un buena práctica es chequear si la ventana ya está abierta pasarle el foco en lugar de abrirla con el método `focus()`. La propiedad `closed` devuelve un booleano indicando si la ventana está abierta o se ha cerrado.

```
if (!mywindow || mywindow.closed) {
  mywindow = window.open('auxPage.html', 'Mywindow', 'width=800,
height=600, top=250, left=250, titlebar=yes, toolbar=no, menubar=no,
location=no');
} else {
  mywindow.focus();
}
```

Referencia: https://developer.mozilla.org/en-US/docs/Web/API/Window/focus

Por último, queremos añadir un botón que cierre la ventana. Antes de cerrarla, debemos comprobar que no se haya cerrado previamente.

```
const bClose = document.createElement('button');
bClose.classList.add('btn');
bClose.classList.add('btn-danger');
bClose.classList.add('m-1');
bClose.innerHTML = 'Cerrar ventana';
bClose.addEventListener('click', (event) => {
  if (mywindow && !(mywindow.closed)) {
    mywindow.close();
    console.log('Acabas de cerrar la ventana.');
  } else {
    console.log('La ventana está cerrada.');
  }
});
bOpen.after(bClose);
```

Referencia: https://developer.mozilla.org/es/docs/Web/API/Window/close

3.2.2. Interacción entre ventanas

Ambas ventanas pueden interactuar entre sí siempre y cuando mantengan la **política de seguridad del mismo origen**.

Vamos a crear un segundo fichero JavaScript denominado **auxPage.js** y lo importamos en la página auxiliar. La ventana auxiliar dispone de la referencia `opener` que apunta a la ventana padre o abridora, que podemos utilizar para ejecutar código entre ventanas.

Si estuviéramos utilizando un `iframe`, la propiedad `parent` haría referencia a la ventana donde se ha cargado la `iframe`.

Tanto en la página principal como en la auxiliar vamos a crear una función global *greeting* que devuelva una cadena de caracteres con un texto de saludo.

La función en la principal sería:

```
function greeting(){
  return "La ventana propietaria dice hola.";
}
```

En la página auxiliar, en el fichero *auxPage.js* tenemos:

```
function greeting(){
  return "La ventana auxiliar dice hola.";
}
```

En primer lugar, vamos a acceder al árbol DOM de la página auxiliar. Creamos un botón para mostrar el texto de la cabecera de la auxiliar en la página principal. Como vemos en el código, hacemos un chequeo para ver si la ventana está abierta, y accedemos al `document` de la ventana auxiliar.

```
const bTitle = document.createElement('button');
bTitle.classList.add('btn');
bTitle.classList.add('m-1');
bTitle.innerHTML = 'Título';
bTitle.addEventListener('click', (event) => {
  if (mywindow && !(mywindow.closed)) {
    console.log(mywindow.document.querySelector('h1').textContent);
  } else {
    console.log('La ventana está cerrada.');
  }
});
bClose.after(bTitle);
```

En la ventana auxiliar añadimos el mismo botón para mostrar el título de la ventana principal. Utilizamos la referencia `window.opener.document` para acceder al árbol DOM de la principal.

```
const message = document.getElementById('message');
const bTitle = document.createElement('button');
bTitle.classList.add('btn');
bTitle.classList.add('btn-primary');
bTitle.classList.add('m-1');
bTitle.innerHTML = 'Título';
bTitle.addEventListener('click', (event) => {
  if (window.opener) {
    message.innerText = window.opener.document.title;
  } else {
    message.innerText = 'La ventana se ha abierto directamente.';
  }
});
document.body.append(bTitle);
```

Vamos a enviar un saludo desde la ventana principal a la auxiliar manipulando el árbol DOM.

```
const bGreeting = document.createElement('button');
bGreeting.classList.add('btn');
bGreeting.classList.add('m-1');
bGreeting.innerHTML = 'Saludar';
bGreeting.addEventListener('click', (event) => {
  if (mywindow && !(mywindow.closed)) {
    mywindow.document.getElementById('message').textContent = greeting();
    mywindow.focus();
  } else {
    console.log('La ventana está cerrada.');
  }
});
bTitle.after(bGreeting);
```

Repetimos la misma operación desde la ventana auxiliar.

```
const bGreeting = document.createElement('button');
bGreeting.classList.add('btn');
bGreeting.classList.add('btn-primary');
bGreeting.classList.add('m-1');
bGreeting.innerHTML = 'Saludar';
bGreeting.addEventListener('click', (event) => {
  if (window.opener) {
    window.opener.document.getElementById('message').textContent =
greeting();
  } else {
    message.innerText = 'La ventana se ha abierto directamente.';
  }
});
document.body.append(bGreeting);
```

Podemos invocar funciones de otras ventanas. Vamos a invocar la función *greeting()* de la ventana auxiliar en la principal para mostrar el mensaje.

```
const bGetGreeting = document.createElement('button');
bGetGreeting.classList.add('btn');
bGetGreeting.classList.add('m-1');
bGetGreeting.innerHTML = 'Obtener Saludo';
bGetGreeting.addEventListener('click', (event) => {
  if (mywindow && !(mywindow.closed)) {
    console.log(mywindow.greeting());
  } else {
    console.log('La ventana está cerrada.');
  }
});
bGreeting.after(bGetGreeting);
```

Repetimos la misma funcionalidad en la auxiliar.

```
const bGetGreeting = document.createElement('button');
bGetGreeting.classList.add('btn');
bGetGreeting.classList.add('btn-primary');
bGetGreeting.classList.add('m-1');
bGetGreeting.innerHTML = 'Obtener Saludo';
bGetGreeting.addEventListener('click', (event) => {
  if (window.opener) {
    message.innerText = window.opener.greeting();
  } else {
    message.innerText = 'La ventana se ha abierto directamente.';
  }
});
$(document.body).append(bGetGreeting);
```

Por último, añadimos un botón para cerrar la ventana auxiliar.

```
const bClose = document.createElement('button');
bClose.classList.add('btn');
bClose.classList.add('btn-primary');
bClose.classList.add('m-1');
bClose.innerHTML = 'Cerrar ventana';
bClose.addEventListener('click', (event) => {
  if (window.opener) {
    window.close();
  } else {
    message.innerText = 'Solo podemos cerrar una ventana abierta por
código.';
  }
});
$(document.body).append(bClose);
```

3.2.3. Abrir enlace en nueva ventana. Atributo `target`

El atributo `target` nos sirve para indicar en qué ventana queremos abrir un enlace o redirigir un formulario. En el siguiente código añadimos un enlace que debe abrirse en la ventana auxiliar al indicar su nombre en el atributo `target`.

```
const link = document.createElement('a');
link.classList.add('btn');
link.classList.add('m-1');
link.innerHTML = 'Enlace a ventana';
link.target = 'Mywindow';
link.href = 'https://developer.mozilla.org/es/';
bGetGreeting.after(link);
```

3.2.4. Métodos `moveTo()` y `moveBy()`

Estos métodos permiten mover la ventana del navegador, tanto con posiciones absolutas como con posiciones relativas a la posición actual. Sus argumentos son la posición `left` y `top`.

En el siguiente ejemplo movemos la ventana auxiliar a una posición fija a través de un botón.

```
const bMoveTo = document.createElement('button');
bMoveTo.classList.add('btn');
bMoveTo.classList.add('m-1');
bMoveTo.innerHTML = 'Mover';
bMoveTo.addEventListener('click', (event) => {
  if (mywindow && !(mywindow.closed)) {
    mywindow.moveTo(1250, 1250);
    mywindow.focus();
  } else {
    console.log('La ventana está cerrada.');
  }
});
link.after(bMoveTo);
```

Ahora añadimos un manejador de eventos para mover la ventana auxiliar con las teclas flechas utilizando el posicionamiento relativo.

```
document.addEventListener('keydown', (event) => {
  event.preventDefault();
  switch (event.code) {
    case 'ArrowUp':
      mywindow.moveBy(0, -10);
      break;
    case 'ArrowDown':
      mywindow.moveBy(0, 10);
      break;
    case 'ArrowLeft':
      mywindow.moveBy(-10, 0);
      break;
    case 'ArrowRight':
      mywindow.moveBy(10, 0);
      break;
  }
});
```

3.2.5. Métodos `resizeTo()` y `resizeBy()`

Estos métodos permiten redimensionar la ventana del navegador, tanto con dimensiones absolutas como con dimensiones relativas. Los parámetros son `width` y `height`.

Añadimos un botón para redimensionar la ventana auxiliar.

```
const bResize = document.createElement('button');
bResize.classList.add('btn');
bResize.classList.add('m-1');
bResize.innerHTML = 'Redimensionar';
bResize.addEventListener('click', (event) => {
  if (mywindow && !(mywindow.closed)) {
    mywindow.resizeTo(1024, 768);
    mywindow.focus();
  } else {
    console.log('La ventana está cerrada.');
  }
});
bMoveTo.after(bResize);
```

Para el movimiento relativo, lo asociamos a las teclas de del *numpad*.

```
document.addEventListener('keydown', (event) => {
  event.preventDefault();
  switch (event.code) {
    case 'Numpad8':
      mywindow.resizeBy(0, 10);
      break;
    case 'Numpad2':
      mywindow.resizeBy(0, -10);
      break;
    case 'Numpad4':
      mywindow.resizeBy(-10, 0);
      break;
    case 'Numpad6':
      mywindow.resizeBy(10, 0);
      break;
  }
});
```

4. Objeto `screen`

Este objeto hace referencia a la pantalla donde se ha cargado la ventana. No suele ser muy relevante en una aplicación, salvo por el uso de mostrar un contenido a tamaño completo. Las principales propiedades son:

- `screen.width`: Ancho de la pantalla en píxeles.
- `screen.height`: Alto de la pantalla en píxeles.
- `screen.availWidth`: Ancho disponible, no tiene en cuenta barra de tareas.
- `screen.availHeight`: Alto disponible, no tiene en cuenta barra de tareas.

La siguiente función muestra estas propiedades.

```
console.log(`Width: ${screen.width}`);
console.log(`Height: ${screen.height}`);
console.log(`availWidth: ${screen.availWidth}`);
console.log(`availHeight: ${screen.availHeight}`);
```

La referencia la podemos encontrar en: https://developer.mozilla.org/en-US/docs/Web/API/Window/screen

5. Objeto `location`

Con este objeto podemos obtener información de la URL actual del recurso cargado.

Referencia: https://developer.mozilla.org/en-us/docs/Web/API/Window/location

5.1. Propiedades

Las propiedades que ofrece este objeto son:

- `window.location.href`: Devuelve la URL de las página actual.
- `window.location.hostname`: Muestra el nombre del host de la página.
- `window.location.pathname`: Muestra el path al recurso.

- `window.location.protocol`: Obtenemos el protocolo que hemos utilizado para acceder.
- `window.location.port`: Puerto sobre el que estamos accediendo.

La siguiente función muestra estas propiedades.

```
console.log(`href: ${location.href}`);
console.log(`hostname: ${location.hostname}`);
console.log(`pathname: ${location.pathname}`);
console.log(`protocol: ${location.protocol}`);
console.log(`puerto: ${location.port}`);
```

5.2. Métodos

Veamos algunos de los principales métodos con los que podemos trabajar con este objeto.

5.2.1. Método `reload()`

Este método recarga la página actual, mostrando las posibles modificaciones que hayamos realizado en el fuente.

```
function reload(){
  window.location.reload();
}
```

5.2.2. Método assign()

Con este método podemos modificar la página actual indicando la URL del nuevo recurso que queremos cargar.

```
function assign(){
  window.location.assign('https://developer.mozilla.org/es/');
}
```

El método `replace()` tiene la misma función, pero no almacena la URL en el objeto `history` del navegador.

6. Objeto `history`

Representa la navegación que ha realizado el usuario en la sesión que tiene abierta. Por cada página visita, el navegador guarda la referencia en una pila, pudiendo volver hacía atrás en la navegación de la página que hemos visitado, para posteriormente recuperar las últimas páginas que hemos visitado, correspondiendo con los botones *atrás* y *adelante* del navegador.

La referencia al objeto la podemos encontrar:

https://developer.mozilla.org/en-US/docs/Web/API/History

El objeto dispone de la propiedad `length` que indica el número de entradas que tenemos en el historial.

```
console.log(`Length: ${window.history.length}`);
```

6.1. Métodos `back()`, `forward()` y `go()`

Los métodos `back()` y `forward()` la URL que hemos visitado previamente o avanzar a la siguiente respectivamente.

El siguiente código añade dos botones en la sección de categorías que nos permiten navegar por las URLs de la historia del usuario.

```
const examples = document.getElementById('examples');
examples.append(document.createElement('hr'));

const bBack = document.createElement('button');
bBack.classList.add('btn');
bBack.classList.add('m-1');
bBack.innerHTML = 'Atrás';
bBack.addEventListener('click', (event) => {
  window.history.back();
});
examples.append(bBack);

const bForward = document.createElement('button');
bForward.classList.add('btn');
bForward.classList.add('m-1');
bForward.innerHTML = 'Adelante';
bForward.addEventListener('click', (event) => {
  window.history.forward();
});
bBack.after(bForward);
```

El método `go()` tiene la misma función pero indicamos cuantos saltos queremos dar a partir de la posición actual en la historia. Un argumento de -1 indica que queremos dar un salto hacia atrás equivalente al método `back()`, con un argumento de -2 daríamos dos saltos. De la misma forma, `go(1)` damos un salto adelante equivalente a `forward()`. Por último, un argumento de 0 o sin argumento, recarga la página actual.

```
const bGoBack = document.createElement('button');
bGoBack.classList.add('btn');
bGoBack.classList.add('m-1');
bGoBack.innerHTML = 'go (-1)';
bGoBack.addEventListener('click', (event) => {
  window.history.go(-1);
});
bForward.after(bGoBack);

const bGoReload = document.createElement('button');
bGoReload.classList.add('btn');
bGoReload.classList.add('m-1');
bGoReload.innerHTML = 'go ()';
bGoReload.addEventListener('click', (event) => {
  window.history.go();
});
bGoBack.after(bGoReload);
```

```
const bGoForward = document.createElement('button');
bGoForward.classList.add('btn');
bGoForward.classList.add('m-1');
bGoForward.innerHTML = 'go (1)';
bGoForward.addEventListener('click', (event) => {
  window.history.go(1);
});
bGoReload.after(bGoForward);
```

6.2. Manipulando el objeto `history`

Con la llegada de HTML5, se ha incluido un API que permite interactuar con el objeto `history`. Este API permite añadir nuevas URLs a pila de `history` sin necesidad de haber sido invocadas, lo que nos facilita la interacción con aplicación que utilicen comunicación asíncrona AJAX.

El uso tradicional de la web implica que la acción de cliquear en un enlace desencadena la descarga de una página completa y todos sus recursos asociados. Imaginemos la navegación de la página A, a la página B, donde ambas páginas comparten el 90% del contenido. Por qué debemos descargar nuevamente todo el contenido si ya lo tenemos en el cliente. No será mucho más lógico descargar 10% de las diferencias e incluirlas en la nueva página.

El **API de History** permite incluir una entrada en el objeto history que represente esa interacción, sin necesidad de haber recargado la página. Esta entrada representará el cambio entre la página A, a la página B, no tiene por qué ser una página real, tan solo no servirá para poder restaurar la página que acabamos de cargar.

Vamos a ver un ejemplo de uso, aunque reducido porque no vamos a utilizar ninguna comunicación AJAX. En primer lugar, vamos a añadir un manejador de eventos para el evento `click` de los enlaces de las categorías. El manejador invoca una función pasando como argumento el enlace cliqueado, y cancela el comportamiento por defecto.

```
const categories = document.getElementById('categories');
let links = categories.getElementsByTagName('a');
links = [...links];
for (const link of links) {
  link.addEventListener('click', function (event) {
    selectCategory(this);
    event.preventDefault();
  });
}
```

La función *selectCategory()* recoge el enlace cliqueado para añadirle un borde a través de CSS a modo de selección, y busca el resto de enlaces para eliminar el borde si lo tuvieran.

```
function selectCategory(cat) {
  cat.style.border = '3px solid red';
  cat.style.display = 'block';
  for (const link of links) {
    if (link !== cat) link.style.border = 'none';
  }
}
```

El API de History dispone del método para añadir una nueva URL al objeto `history` siempre y cuando garanticemos la política del mismo origen. El método `pushState()` puede recibir tres argumentos:

- **Estado**: es un objeto literal que nos permita guardar el estado actual que la URL que estamos apilando. El tamaño de este objeto queda limitado a 640 caracteres, por lo que solamente podemos guardar la información que nos permita recuperar objetos utilizando el **API de Persistencia** o **IndexDB**.
- **Título**: Se trata de un título para la URL, aunque realmente ningún navegador lo está implementando.
- **URL**: La URL que estamos apilando. Es opcional, si no lo pasamos estaremos apilando la URL actual del navegador.

Vamos a hacer uso de este método. Cada vez que cliqueemos en una categoría tenemos que apilar la URL actual, y pasamos como estado la posición de la categoría cliqueada.

```
link.addEventListener('click', function (event) {
  selectCategory(this);
  const index = links.indexOf(link);
  history.pushState({ category: index }, null);
  event.preventDefault();
});
```

Aunque no haya transición entre páginas, estamos añadiendo una nueva URL, en este caso la actual, a la pila de `history`.

El último paso es restaurar el estado cada vez que nos movamos por el objeto `history`. Al pulsar el botón de *atrás* o *adelante* se produce el evento `popstate`. Podemos asociar un manejador de eventos al objeto `window` sobre este evento, en el cual podremos restaurar el estado que teníamos en la página. El evento tiene acceso al objeto literal con estado que habíamos almacenado en `pushState()` en la propiedad `state`, lo recuperamos, y en base a su información restauramos el estado invocando a *selectCategory()*.

```
window.addEventListener('popstate', (event) => {
  if (event.state) {
    selectCategory(links[event.state.category]);
  }
});
```

El método `replaceState()` tiene la misma función que `pushState()` pero en lugar de apilar una nueva URL, sustituye la actual.

En este enlace podemos ver la referencia del API de History: https://developer.mozilla.org/es/docs/DOM/Manipulando_el_historial_del_navegador

7. Alertas

Además de la función `alert()` tenemos otras dos funciones que nos permiten interactuar con el usuario.

La función `confirm()` genera una alerta con la botonear para confirma o cancelar una acción. La función devuelve un booleano para indicar cuál ha sido el botón que se ha pulsado.

```
let value = confirm('Confirma o cancela la acción');
console.log(value);
```

La función `prompt()` sigue la misma filosofía pero la utilizamos con una caja de texto. El segundo parámetro recibe el valor por defecto para la caja de texto.

```
let value = prompt("Dime tu nombre","anónimo");
console.log(value);
```

Al no poder modificar la ubicación, ni el estilo de las alertas, **no son un buen recurso para integrar en una aplicación**. Solo se deberían utilizar a nivel de debugado de una aplicación, implementando la recogida de datos a través de formulario o ventanas modales creadas en a través de DOM.

8. Timing

Las funciones de timing nos permiten ejecutar funciones utilizando un parámetro temporal en milisegundos.

8.1. Función `setTimeOut()`

Esta función invoca una función pasada como primer argumento una vez transcurridos el número de milisegundo pasados como segundo argumento. En el siguiente código generamos un botón que invoca la función `setTimeout()`. Los argumentos son una función arrow que será invocada después de 3000 milisegundos.

```
const examples = document.getElementById('examples');
examples.append(document.createElement('hr'));
const bStartTimeOut = document.createElement('button');
bStartTimeOut.classList.add('btn');
bStartTimeOut.classList.add('m-1');
bStartTimeOut.innerHTML = 'Inicio TimeOut';

let controlTimeOut = null;

bStartTimeOut.addEventListener('click', (event) => {
  controlTimeOut = setTimeout(() => {
    alert('Función Timeout');
    console.log('Fin del Time out.');
  }, 3000);
  console.log('Inicio del Time out.');
});
examples.append(bStartTimeOut);
```

Un detalle que tenemos que tener en cuenta es que `setTimeout()` genera un segundo hilo de ejecución, es decir, el código no se detiene a la espera a que finalice la función pasada como argumento, sino que continua de forma normal. Una vez pasado el tiempo de espera, la función pasada como argumento es invocada.

Como podemos observa en el código, hemos utilizado una variable global para recibir el valor de retorno de `setTimeout()`. Esta variable la podemos utilizar para cancelar la invocación de `setTimeout()`. La función `clearTimeout()` recibe como argumento la variable global con lo que permite la cancelación. El siguiente código cancela la invocación de la función `setTimeout()` al cliquear el botón que hemos añadido.

```
const bStopTimeOut = document.createElement('button');
bStopTimeOut.classList.add('btn');
bStopTimeOut.classList.add('m-1');
bStopTimeOut.innerHTML = 'Parar TimeOut';

bStopTimeOut.addEventListener('click', (event) => {
  clearTimeout(controlTimeOut);
  console.log('Timeout cancelado.');
});
bStartTimeOut.after(bStopTimeOut);
```

8.2. Función `setInterval()`

El funcionamiento de esta función es similar a la anterior, pero en lugar de ejecutarse una única vez la función pasada como argumento, se ejecuta a modo de bucle en función del número de milisegundos indicados en el segundo argumento.

El siguiente código genera dos botones. El primero inicia un reloj que se incrementa cada segundo. El segundo permite detener el reloj.

```
const examples = document.getElementById('examples');
const message = document.getElementById('message');
examples.append(document.createElement('hr'));

let controlInterval = null;
const bStartInterval = document.createElement('button');
bStartInterval.classList.add('btn');
bStartInterval.classList.add('m-1');
bStartInterval.innerHTML = 'Iniciar Reloj';

bStartInterval.addEventListener('click', (event) => {
  controlInterval = setInterval(() => {
    const date = new Date();
    message.innerHTML =
      `${date.getHours()}:${date.getMinutes()}:${date.getSeconds()}`;
  }, 1000);
});
examples.append(bStartInterval);

const bStopInterval = document.createElement('button');
bStopInterval.classList.add('btn');
bStopInterval.classList.add('m-1');
bStopInterval.innerHTML = 'Parar Reloj';

bStopInterval.addEventListener('click', (event) => {
  clearTimeout(controlInterval);
});
bStartInterval.after(bStopInterval);
```