Profesor: Borja Rey Seoane

TEMA 3

MODELO DE OBXECTOS DA LINGUAXE

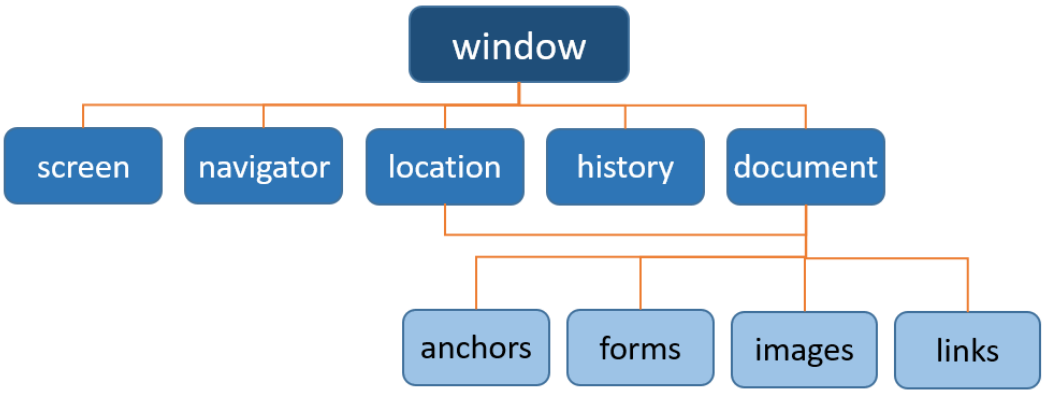
Módulo: **Desenvolvemento Web en Contorna Cliente**

Ciclo: **DAW**

Obxectos predefinidos

JS permite a comunicación co navegador co gaio de personalizar a contorna, de xeito que obteñamos aplicacións con maior interactividade. O anterior lógrase mediante o uso de BOM[1].

O devandito modelo non conta cun estándar que o defina e, polo tanto, os navegadores poden implementalo de distintos xeitos. Mesmo así, hai un certo consenso, existindo varios obxectos comúns aos navegadores principais, que forman a seguinte estrutura xerárquica:



Cando unha páxina web é cargada no navegador créanse os obxectos anteriores en base á estrutura da mesma, as características da aplicación cliente e a interacción do usuario.

A tarefa principal do BOM é a de xestionar as xanelas do navegador, permitindo a comunicación entre elas.

`Window`

Este obxecto, coma vimos no diagrama anterior, atópase na parte máis alta da xerarquía e considérase o obxecto por defecto, polo que non fai falta referencialo para acceder ás súas propiedades e métodos.

`Window` representa ao navegador, aínda que naqueles navegadores que permiten varias lapelas a cada unha delas correspóndelle un obxecto `window`, isto é, non se comparte o obxecto deste tipo entre as lapelas da mesma xanela de aplicación. De xeito semellante, se un documento contén `frames` (etiquetas `<iframe>`) a cada unha delas

[1] *Browser Object Model*.

corresponderá un obxecto `window` (ademais do principal que corresponda ao documento HTML que actúa coma contedor).

Ademais da área da xanela destinada a amosar a páxina web, este obxecto tamén permite traballar coas medidas da xanela, as barras de desprazamento, a barra de ferramentas, a barra de estado, etc.

Un aspecto importante a considerar cando programamos en JS é que todas as variables, obxectos e funcións globais que non sexan asignadas a ningún outro obxecto serán membros de `window`.

Referencia

Existen varias formas de facer referencia aos seus membros deste obxecto:

- **`window`:** coma con calquera obxecto, indícase o nome do mesmo e, separado por un punto (.), a propiedade ou método ao cal queremos acceder.
- **`self`:** pódese substituír o nome do obxecto pola palabra reservada `self`, de xeito que poidamos facer desde ao propio documento contido na xanela.
- **Por defecto:** aínda que fai menos comprensible o código, por tratarse do obxecto predefinido, pódese omitir a referencia ao mesmo e indicar simplemente a propiedade ou método a cal queremos acceder.

```
// Formas equivalentes de acceder á propiedade parent
window.parent;
self.parent;
parent;

// Formas equivalentes de acceder ao método close
window.close();
self.close();
close();
```

Propiedades

A continuación veremos unha enumeración das distintas propiedades de `window` e os valores que poden tomar:

- **Estado:**
  - `closed`: indica se a xanela foi pechada ou non.
  - `defaultStatus`: establece ou devolve o texto por defecto da barra de estado.
  - `name`: establece ou devolve o nome da xanela.
  - `status`: establece ou devolve o texto da barra de estado.

- **Tamaño:**
  - `innerHeight`: altura da área de visualización da xanela (incluíndo barras de desprazamento).
  - `innerWidth`: largura da área de visualización da xanela (incluíndo barras de desprazamento).
  - `outerHeight`: altura de toda a xanela do navegador, incluíndo a zona exterior á área de visualización (barras de ferramentas, de título, de estado, etc.).
  - `outerWidth`: largura de toda a xanela do navegador, incluíndo a zona exterior á área de visualización (barras de ferramentas, de título, de estado, etc.).
- **Posición:**
  - `pageXOffset`: cantidade de pixels desprazados horizontalmente dende a esquina superior-esquerda da xanela.
  - `pageYOffset`: cantidade de pixels desprazados verticalmente dende a esquina superior-esquerda da xanela.
  - `screenLeft`: coordenada horizontal da xanela relativa á pantalla.
  - `screenTop`: coordenada vertical da xanela relativa á pantalla.
  - `screenX`: coordenada horizontal da xanela relativa á pantalla.
  - `screenY`: coordenada vertical da xanela relativa á pantalla.
- **Frames:**
  - `frameElement`: elemento `<iframe>` no que se atopa a páxina actual.
  - `frames`: conxunto de elementos `<iframe>` que se atopan na páxina actual.
  - `length`: número de elementos `<iframe>` que se atopan na páxina actual.
- **Obxectos do BOM:**
  - `console`: referencia á consola do navegador que permite enviar mensaxes á mesma.
  - `document`: referencia ao obxecto `document` da xanela.
  - `history`: referencia ao obxecto `history` da xanela.
  - `location`: referencia ao obxecto `location` da xanela.
  - `navigator`: referencia ao obxecto `navigator` da xanela.
  - `screen`: referencia ao obxecto `screen` da xanela.
- **Almacenaxe no cliente:**
  - `localStorage`: permite almacenar pares clave/valor na aplicación cliente, sen data de expiración.

    - `sessionStorage`: permite almacenar pares clave/valor na aplicación cliente, durante a sesión actual.
- **Referencias:**
    - `opener`: referencia a xanela que abriu a actual.
    - `parent`: referencia a xanela nai do `frame` actual.
    - `self`: referencia a xanela actual.
    - `top`: referencia a xanela que está no tope da xerarquía.

```
// Obter información da xanela
var info = "Screen size = " + screen.availWidth +" x "+ screen.availHeight + "
px";
info += "<br/>Outer size = " + window.outerWidth +" x "+ window.outerHeight + "
px";
info += "<br/>Inner size = " + window.innerWidth +" x "+ window.innerHeight + "
px";
info += "<br/>Position = " + window.screenLeft +" x "+ window.screenTop + " px";
document.getElementById("info").innerHTML = info;
```

Métodos

- **`alert()`:** amosa unha xa nela cunha mensaxe e un botón de OK.
- **`atob()`:** decodifica unha *string* codificado en base 64.
- **`blur()`:** quita o foco da xanela actual.
- **`btoa()`:** codifica unha *string* en base 64.
- **`clearInterval()`:** borra o temporizador establecido con `setInterval()`.
- **`clearTimeout()`:** borra o temporizador establecido con `setTimeout()`.
- **`close()`:** pecha a xanela actual.
- **`confirm()`:** amosa unha xanela de diálogo cunha mensaxe e 2 botóns (OK e Cancelar).
- **`focus()`:** establece o foco na xanela actual.
- **`moveBy()`:** move a xanela relativa á posición actual.
- **`moveTo()`:** move a xanela a unha posición absoluta.
- **`open()`:** abre unha xanela do navegador nova.
- **`print()`:** imprime o contido da xanela actual.
- **`prompt()`:** amosa unha xanela de diálogo na que o usuario debe introducir información.
- **`resizeBy()`:** cambia o tamaño da xanela actual en base aos pixels da actual.
- **`resizeTo()`:** establece o tamaño (largo x alto) da xanela actual á cantidade de pixels indicada.
- **`scrollBy()`:** despraza o documento unha cantidade de pixels en base á posición actual.
- **`scrollTo()`:** despraza o documento ás coordenadas indicadas.

- **`setInterval()`:** chama a unha función ou avalía unha expresión a intervalos constantes (en milisegundos).
- **`setTimeout()`:** chama a unha función ou avalía unha expresión transcorridos uns milisegundos.
- **`stop()`:** detén a carga de contido na xanela.

```
// Pedimos un dato ao usuario
var seconds = window.prompt("Indica unha cantidade de segundos: ", "5");
// Executa unha función cada 1000 ms
var myTimer = setInterval(showTimer, 1000);

function showTimer() {
  document.getElementById("info").innerHTML = new Date().toLocaleTimeString();
  seconds--;
  if (seconds==0) {
    // Deixa de executar a función ao borrar o temporizador
    clearInterval(myTimer);
    // Amosa unha xanela emerxente cunha mensaxe
    window.alert("Rematou o tempo");
  }
}
```

Xestión

Desde o código JS nunca se poderá crear a xanela principal do navegador, xa que este código terá que estar cargado previamente nunha xanela inicial aberta polo usuario. A partires desa situación poderanse abrir subxanelas novas co método `open()`, o cal ten a seguinte sintaxe:

```
window.open(url, target, features, replace);
```

Parámetros

Todos os parámetros da función `open()` son opcionais:

- **`url`:** indica a URL do recurso que desexamos cargar na xanela. Se a deixamos en branco, abrirase unha xanela co contido baleiro `about:blank`.
- **`target`:** establece o destino no que se cargará a nova xanela ou o nome da mesma (para poder referila posteriormente):
  - `_blank`: a URL cárgase nunha nova xanela/lapela. É a opción por defecto.
  - `_parent`: a URL cárgase no marco pai da actual.
  - `_self`: a URL substitúe a páxina actual.
  - `_top`: a URL substitúe calquera conxunto de marcos que poidan estar cargados.
  - `name`: nome da nova xanela.

- **`features:`** trátase dunha listaxe de valores que permiten establecer propiedades da nova xanela. A listaxe debe ir delimitada por comiñas (""), separando cada par/valor mediante comas (,) e sen incluír espazos en branco entre elementos. A continuación algunhas das de uso máis habitual[2]:
  - **`height=pixels:`** alto da xanela en pixels. O mínimo é 100.
  - left=pixels. Posición con respecto á marxe esquerda da pantalla.
  - **`location=yes|no|1|0:`** amosar a barra de direccións (só en Opera).
  - **`menubar=yes|no|1|0:`** amosar a barra de menús.
  - **`resizable=yes|no|1|0:`** permitir redimensionar a xanela (só en Internet Explorer).
  - **`scrollbars=yes|no|1|0:`** amosar barras de desprazamento (só en Internet Explorer, Firefox e Opera).
  - **`status=yes|no|1|0:`** amosar a barra de estado.
  - **`titlebar=yes|no|1|0:`** amosar a barra de título.
  - **`toolbar=yes|no|1|0:`** amosar a barra de ferramentas (só en Internet Explorer e Firefox).
  - **`top=pixels:`** posición con respecto ao tope superior da pantalla.
  - **`width=pixels:`** largura da xanela en pixels. O mínimo é 100.
- **`replace:`** indica se creamos unha entrada nova no historial (`false`) ou se substitúe a actual (`true`).

Exemplo

A seguinte instrución abriría unha xanela nova de 800x600 pixels, sen barra de estado e na que se carga o contido dunha páxina HTML xa creada que se atopa na mesma ubicación que a actual (new.html).

```
var  subWindow = window.open("new.html", "new", "height=600,width=800");
```

Se desexamos omitir algún dos parámetros anteriores, basta con indicar comiñas dobres ("") no que corresponda.

Dentro do código pódese facer referencia á mesma mediante `new`, e así acceder ás súas propiedades e métodos co gaio manexala a través da variable `subWindow`. Así por exemplo, para pechar a xanela aberta, empregaríase o método `close()`:

```
subWindow.close();
```

[2] Algunhas das características anteriores non funcionan coa configuración por defecto de Chrome, Firefox, IE, Edge, Opera e Safari.

Un exemplo máis elaborado sería:

```
// Abre unha páxina xa creada nunha subxanela dun tamaño concreto
var  subWindow = window.open("new.html", "_blank",
"width=400,height=300,status=no,location=yes,scrollbars=0,titlebar=1", "true");

// Comproba se a subxanela foi creada (p.ex: non bloqueada por ser emerxente)
if (!subWindow.closed) {
  // Redimensiona a xanela á metade do tamaño da pantalla do usuario
  subWindow.moveTo(0, 0);
  subWindow.resizeTo(screen.availWidth/2, screen.availHeight/2);

  // Obtén información da subxanela
  var info = "Screen size = " + screen.availWidth +"x"+ screen.availHeight +" px"+
    "<br>Subwindow size = " + subWindow.outerWidth +"x"+ subWindow.outerHeight + "
px";
  document.getElementById("info").innerHTML = info;
}
```

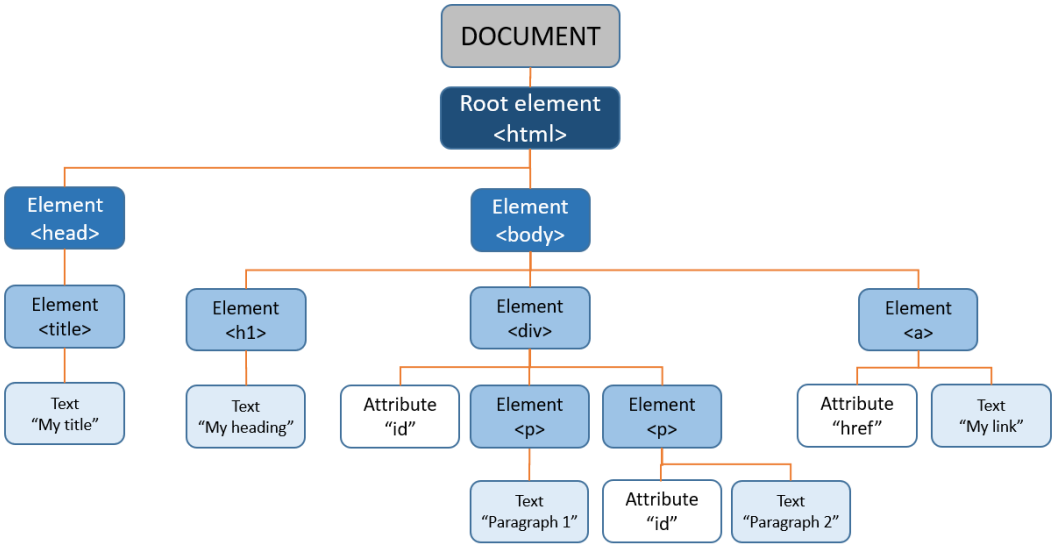
Document

Cando se carga unha páxina HTML no navegador, este crea unha estrutura xerárquica en forma de árbore que a representa, o que se coñece como a árbore DOM. Como xa sabemos, DOM é un estándar do W3C (World Wide Web Consortium) que fornece unha interface que permite, por medio de programación, acceder e actualizar dinamicamente o contido, estrutura e estilo dun documento, e que define:

- Os elementos HTML como obxectos.
- Propiedades dos elementos HTML.
- Métodos para manexar os elementos HTML.
- Eventos para os elementos HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      My title
    </title>
  </head>
  <body>
    <h1>My heading</h1>
    <div id="layer1">
      <p>Paragraph 1</p>
      <p class="p2">Paragraph 2</p>
    </div>
    <a href="#">My link</a>
  </body>
</html>
```

Para o documento HTML do exemplo anterior teriamos a seguinte árbore DOM:



Facendo cambios sobre os nodos da estrutura anterior mediante JS podemos fornecer capacidades dinámicas ao contido HTML, tales coma:

- Modificar os nodos da páxina HTML (elementos, atributos e textos).
- Modificar os estilos CSS da páxina.
- Engadir e eliminar nodos (elementos, atributos e textos).
- Reaccionar ante eventos producidos sobre a páxina HTML.
- Rexistrar eventos novos.

Propiedades

- **`activeElement`:** elemento que ten o foco actualmente no documento.
- **`anchors`:** colección de enlaces do documento (`<a>`) que teñen o atributo `name`.
- **`applets`:** colección de applets (`<applet>`) do documento.
- **`body`:** elemento `<body>` do documento.
- **`cookie`:** todos os pares nome/valor das cookies do documento.
- **`characterSet`:** xogo de carácteres en uso no documento.
- **`defaultView`:** obxecto `window` asociado ao documento.
- **`doctype`:** declaración do tipo de documento asociado ao actual.
- **`documentElement`:** elemento `<html>` do documento.
- **`domain`:** nome do dominio do servidor que serviu o documento.
- **`embeds`:** colección de elementos `<embed>` do documento.
- **`forms`:** colección de elementos `<form>` do documento.
- **`head`:** elemento `<head>` do documento.
- **`images`:** colección de elementos `<img>` do documento.
- **`inputEncoding`:** xogo de caracteres establecido para o documento.

- **links:** colección de elementos `<a>` e `<area>` do documento que teñan atributo `href`.
- **readyState:** estado do documento relativo á súa carga no navegador.
- **referrer:** URL do documento que cargou o actual.
- **scripts:** colección de elementos `<script>` do documento.
- **title:** elemento `<title>` do documento.
- **URL:** URL completa de acceso ao documento HTML.

Métodos

- **addEventListener():** asigna un manexador de eventos ao elemento especificado.
- **close():** pecha o fluxo de saída aberto previamente con `document.open()`.
- **createAttribute():** crea un nodo de tipo atributo.
- **createComment():** crea un nodo de tipo comentario co texto indicado.
- **createElement():** crea un nodo de tipo elemento.
- **createEvent():** crea un evento novo.
- **createTextNode():** crea un nodo de tipo texto.
- **getElementById():** devolve o elemento cuxo atributo ID é igual ao valor indicado.
- **getElementsByClassName():** devolve un `NodeList` que conterá todos os elementos co nome de clase igual ao valor indicado.
- **getElementsByName():** devolve un `NodeList` que conterá todos os elementos co nome igual ao valor indicado.
- **getElementsByTagName():** devolve un `NodeList` que conterá todos os elementos co nome da etiqueta igual ao valor indicado.
- **hasFocus():** indica se o documento ten o foco (se fai click dentro ou se selecciona algún elemento do mesmo co tabulador).
- **open():** abre un fluxo de saída HTML para captar a saída de `document.write()`.
- **querySelector():** devolve o primeiro elemento do documento que coincida co selector CSS indicado.
- **querySelectorAll():** devolve un `NodeList` con todos os elementos do documento que coinciden co selector CSS indicado.
- **removeEventListener():** elimina un manexador de eventos do documento (asignado previamente con `document.addEventListener()`).
- **renameNode():** renomea o nodo correspondente.
- **write():** escribe código HTML/JavaScript no documento.

- **`writeln()`:** escribe código HTML/JavaScript no documento, engadindo un caracter de nova liña ao final.

```
document.write("<html><head><title>JavaScript</title></head><body><h1>Document</h1
><p>Contido HTML xerado dende JS</p><button
onclick='crear()'>Crear!</button></body></html>");

function crear() {
  var par = document.createElement("P");
  var text = document.createTextNode("Nodo de parágrafo dinámico.");
  par.appendChild(text);
  document.body.appendChild(par);
}
```

Exemplo

Segundo se viu no modelo do BOM, o obxecto `document` ten unha serie de *nodelists* que actúan coma coleccións de elementos HTML. Para acceder a eles, pódense empregar tanto os métodos do DOM como os propios obxectos do BOM, do seguinte xeito:

- Se temos o seguinte documento HTML:

```
<html>
  <head>
    <title>Meu título</title>
  </head>
  <body>
    <h1>O meu título</h1>
    <a href="paxina_externa.html">O meu enlace</a>
    <img src="logo.png" name="logo" />
    <form method="post" name="rexistro">
      <input type="text" name="id" />
      <input type="submit" value="send" />
    </form>
  </body>
</html>
```

- Podemos acceder aos elementos dende JS, por exemplo así:

```
// Acceso empregando métodos do DOM
document.getElementByTagName("h1");

// Acceso empregando obxectos do BOM
document.links[0];
document.images[0];
document.images["logo"];
document.forms[0];
document.forms["rexistro"];

// Acceso ás propiedades dun obxecto
document.forms[0].method;
document.images["logo"].src;
```

Screen

Trátase dun obxecto que non posúe métodos de seu, senón que contén información referente á aparencia e características de visualización da pantalla do usuario, isto é, manéxase exclusivamente a través de propiedades.

Propiedades

- **availHeight:** altura da pantalla excluíndo a barra de tarefas (en pixels).
- **availWidth:** largura da pantalla excluíndo a barra de tarefas (en pixels).
- **colorDepth**[3]**:** profundidade de cor da paleta de cores do navegador (en bits).
- **height:** altura total da pantalla (en pixels).
- **pixelDepth:** profundidade de cor da pantalla (en bits).
- **width:** largura total da pantalla (en pixels).

```
// Redimensionar a xanela ao tamaño máximo segundo a pantalla do usuario
window.moveTo(0, 0);
window.resizeTo(screen.availWidth, screen.availHeight);
```

Navigator

O obxecto `navigator` fornece información relativa ao estado e identidade dos axentes de usuario (o navegador e o sistema operativo dende os cales é accedida a páxina actual).

Entre os seus usos habituais atopamos:

- Determinar o tipo/versión do navegador en aplicacións cuxo código difire segundo o cliente.
- Detectar se o navegador ten habilitadas características requiridas polas aplicacións: Java, *cookies*, plugins dispoñibles.

Hai que ter coidado coa información obtida a través desde obxecto posto que pode levar a conclusións erróneas polos seguintes motivos:

- Diferentes navegadores poden empregar o mesmo nome.
- A información do navegador pode ser cambiada polo propietario do mesmo.
- Algúns navegadores mudan a súa identificación intencionadamente para enganar aos servidores.

[3] Nos ordenadores modernos, o `colorDepth` e o `pixelDepth` son equivalentes.

- Os navegadores non poden informar do uso de sistemas operativos novos, lanzados despois do xurdimento da versión que esteamos a empregar do navegador.

Propiedades

- **`appCodeName`:** nome en código do navegador[4].
- **`appName`:** nome que representa o tipo de navegador.
- **`appVersion`:** información da versión do navegador.
- **`cookieEnabled`:** indica se as *cookies* están habilitadas no navegador.
- **`geolocation`:** este obxecto pode ser empregado para determinar a localización xeográfica do usuario.
- **`language`:** idioma do navegador.
- **`onLine`:** indica se o navegador está conectado.
- **`platform`:** indica a arquitectura de procesador para a cal foi compilado o navegador.
- **`plugins`:** listaxe de *plugins* instalados no navegador.
- **`product`:** nome do motor do navegador.
- **`userAgent`:** cabeceira enviada polo axente de usuario ao servidor.

```
var x = navigator.userAgent;
var y = navigator.platform;
```

Métodos

- **`javaEnabled()`:** indica se está habilitada no navegador a execución de código Java.

```
var z = navigator.javaEnabled();
```

Location

Este obxecto permite acceder e manipular as partes da URL que corresponden á paxina actual, así coma redirixir o navegador a unha ubicación nova. Por mor da falta de estandarización, pódese acceder a este obxecto tanto a través de `window` como de `document`:

```
window.location;
```

[4] Por motivos de retrocompatibilidade, todos os navegadores modernos devolven o código Mozilla para se referir aos navegadores de tipo Firefox.

```
document.location; // É equivalente ao anterior
```

Propiedades

Para entender mellor as propiedades deste obxecto, partiremos da seguinte URL de exemplo:

```
http:// www.dominio.gal:8080/ruta/subruta/paxina.php#seccion?key1=valor1
```

- **hash:** parte da URL que corresponde co *anchor* (`#`).

```
#seccion
```

- **host:** nome do *host* e número de porto.

```
www.dominio.gal:8080
```

- **hostname:** nome do *host*.

```
www.dominio.gal
```

- **href:** a URL completa. Se escribimos nesta propiedade, provocamos que o navegador se redirixa á nova ubicación.

```
http:// www.dominio.gal:8080/ruta/subruta/paxina.php#seccion?key1=valor1
```

- **origin:** protocolo, nome do *host* e número de porto da URL.

```
http://www.dominio.gal:8080
```

- **pathname:** ruta do recurso ao que se está a acceder no servidor.

```
/ruta/subruta/paxina.php
```

- **port:** número de porto da URL.

```
8080
```

- **protocol:** protocolo que se está a empregar para acceder ao recurso.

```
http:
```

- **search:** cadea de busca dentro da URL. Normalmente o que se indica despois dunha interrogación pechada (`?`).

```
?key1=valor1
```

Métodos

- **`assign()`:** carga un novo documento.
- **`reload()`:** recarga o documento actual, sendo equivalente a facer *click* no botón de actualizar do navegador.
- **`replace()`:** substitúe o documento actual cun novo. É semellante ao método `assign()`, pero o sitio substituído elimínase do historial da sesión.

```
location.assign("https:// www.xunta.gal/portada"); // Redirixe o navegador a outra
URL
location = "https:// www.xunta.gal/portada"; // Máis un xeito de facer o mesmo
```

`history`

Este obxecto rexistra e permite xestionar o historial da sesión do navegador, isto é, as páxinas visitadas polo usuario na xanela ou `frame` na que está cargada a páxina actual.

Propiedades

- **`length`:** número de URL visitadas, incluída a actual.

Métodos

- **`back()`:** carga a URL previa do historial.
- **`forward()`:** carga a URL seguinte do historial.
- **`go()`:** vai a unha URL concreta, relativa á posición no historial da actual.

```
var x = history.length; // 1 (ao abrir a xanela por vez primeira)
history.back(); // Vai á páxina anterior no historial
history.go(-1); // Máis un xeito de facer o mesmo
```

Almacenaxe en cliente

Como xa sabemos, HTTP é un protocolo non orientado a conexión, o que quere decir que unha vez que o servidor web remata de enviar a páxina solicitada, este pecha a conexión e non lembrará ao usuario en peticións posteriores. Para resolver esta característica do protocolo predominante na Web (que tivo os seus motivos fundacionais pero que hoxe en día limita notablemente a funcionalidade do servizo) e, por exemplo, manter a sesión de usuario namentres este navega por un sitio web, fíxose precisa a implementación de mecanismos de almacenaxe no lado do cliente co gaio de permitir gardar pequenas porcións de información asociadas ás peticións que se fixeron dende un navegador concreto cara un certo servidor.

Co paso do tempo foron aparecendo distintos métodos para gardar información do usuario entre distintas solicitudes de páxinas web, como poden ser:

- **Cookies:** foron a primeira solución de almacenaxe no cliente por parte do navegador. Tamén constitúen a solución máis limitada e rudimentaria, aínda que se manteñen en uso até hoxe en día (sendo o mecanismo máis habitual deste tipo) grazas á súa simplicidade.
- **WebStorage:** considérase o substituto natural das *cookies*, xa que tamén permite almacenar información en pares clave/valor, pero ampliando a cantidade de información que se pode almacenar á orde dos MB, no canto dos KB que permitían as Cookies.
- **Atributos HTML5:** a última versión de HTML permite anexar atributos de datos as elementos do DOM, de xeito que estes poden ser incluídos dende o servidor para ser accedidos dende o código cliente (p.ex.: JavaScript).
- **Indexed DB:** permite almacenar datos estruturados e realizar procuras eficientes sobre eles. Tamén amplía aínda máis a cantidade de información que se pode almacenar (chegando á orde dos GB).
- **Websockets:** solución incluída en HTML5 que permite establecer con JavaScript unha comunicación orientada a conexión bidireccional entre o cliente e o servidor. Con eles resólvense principalmente as situacións nas que a información debe de se transmitir en resposta a eventos producidos no servidor.

Cookies

As *cookies* son pequenos ficheiros de texto asociados á comunicación dun navegador cun mesmo dominio aloxado nun servidor. Neses ficheiros gárdanse pares clave/valor que poden ser accedidos dende JS mediante a propiedade `document.cookie`.

O seu obxectivo é o de permitir lembrar información acerca do usuario (nome, preferencias, etc.), resolvendo así a desvantaxe derivada do feito de que o protocolo HTTP sexa non orientado a conexión.

Entre outras características, as *cookies* admiten o establecemento dunha data de expiración que, no caso de non ser fixada, indicaría que a mesma caducará no intre en que se peche o navegador.

Cando un navegador solicita unha páxina web a un servidor, as *cookies* asociadas ao documento solicitado son engadidas á devandita petición. Destea maneira o servidor obtén os recursos necesarios para recordar a información do usuario ao cal pertence a sesión.

Así mesmo, as *cookies* presentan unha serie de limitacións, tales coma:

- **Tamaño máximo por *cookie*:** na maioría de navegadores é de 4 KB (se ben o soporte a 8KB estase a estender na actualidade).
- **Número máximo de *cookies* por sitio web:** na maioría de navegadores é de 20 *cookies*.
- **Número máximo de *cookies* por dominio:** na maioría de navegadores é de 50 *cookies*.
- **Número máximo de *cookies* por navegador:** este dato varía notablemente entre navegadores e dispositivos, pero adoita estar en torno ás 3.000 *cookies*.

En relación co anterior, o uso de *cookies* para a conservación de información de sesión conleva tamén unha serie de desvantaxes:

- A acumulación de *cookies*, ademais de consumir espazo de almacenaxe, pode resultar na ralentización do navegador.
- As *cookies*, en tanto ficheiros de texto plano, poden ser facilmente accedidas e empregadas para desvelar información confidencial ou alterar os datos contidos nas mesmas se non son manexadas e almacenadas coidadosamente.
- O cifrado de *cookies*, se ben mellora o manexo seguro das mesmas, afecta de xeito notable ao rendemento do navegador que ten que manexalas.

Operacións

Accedendo á propiedade `document.cookie`, pódense levar a feito as operacións típicas de xestión deste tipo de elementos:

- **Creación:** faise asignando unha *string* que conteña a información axeitada coa seguinte sintaxe:

```
document.cookie = cookieString;
```

  O parámetro `cookieString` será un texto que conterá unha lista de pares clave/valor separados por punto e coma (`;`), ou un só par clave/valor con algún dos seguintes valores opcionais:
  - **`expires=data`:** indica a data en formato GMT (ver método `Date.toUTCString()`). Se non se inclúe, a *cookie* borrarase cando se peche o navegador.
  - **`max-age=segundos`:** establece a duración máxima en segundos. Se non se inclúe, a *cookie* borrarase cando sexa pechado o navegador. Este parámetro ten preferencia sobreo anterior (`expires`).
  - **`path=ruta`:** sinala o directorio no que se almacenará a *cookie* (p.ex: "`/`", "`/cookies`"). A ruta debe indicarse de xeito absoluto. Se non se indica a *cookie* pertence á páxina actual.
  - **`domain=nomeDominio`:** especifica o dominio do sitio (p.ex: "`domain.com`", "`subdomain.domain.com`"). Se non se indica, interprétase que será o dominio da páxina actual.
  - **`secure`:** especifica que vai empregarse o protocolo web seguro HTTPS para enviar a *cookie* ao servidor.

```
document.cookie="username=Xiao; expires=Tue, 9 Nov 2021 10:39:00 UTC; path=/";
```

A propiedade `document.cookie` pode parecer unha *string* normal, pero ao lela só se visualizan os pares clave/valor:

```
"username=Allen;"
```

Ao crear unha *cookie* nova non se sobrescriben a existentes senón que esta é engadida ao resto en `document.cookie`:

```
document.cookie="cookie1=valor1;";
document.cookie="cookie2=valor2;";
console.log(document.cookie); // cookie1=valor1; cookie2=valor2;
```

NOTA: o valor dunha *cookie* non pode conter comas (,), puntos e coma (;) ou espaczos en branco ( ). É útil empregar o método `encodeURIComponent()` para evitar conflitos neste sentido.

- **Acceso:** para obter as *cookies* dun documento operaremos así:

```
var c = document.cookie;
```

O anterior devolverá unha *string* do estilo:

```
"cookie1=valor1; cookie2=valor2; cookie3=valor3;"
```

Como xa mencionamos, accedendo á propiedade `document.cookie` obtéñense todos os pares clave/valor. Para acceder a unha *cookie* concreta haberá que empregar código JavaScript (p.ex: empregando o método `String.split()`) para fragmentar a *string* completa e poder acceder a cada par de modo individualizado.

- **Modificación:** pódese mudar o valor dunha *cookie* xa existente do mesmo xeito que se creou, xa que se sobrescribirá o valor orixinal:

```
document.cookie="username=Dombodán; expires=Mon, 8 Nov 2021 12:05:57 UTC; path=/";
```

- **Borrado:** para eliminar unha *cookie* basta con coñecer o seu nome (non é necesario coñecer o seu valor) e, ou ben especificar unha data xa pasada no parámetro `expires`, ou ben unha duración de 0 segundos en `max-age`:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
document.cookie = "username=; max-age=0; path=/;";
```

WebStorage

Trátase dunha API (*Application Programming Interface*) que xurdiu canda HTML5 como alternativa ás *cookies* para almacenar datos no cliente asociados á sesión do usuario ou permanentes.

O uso de `WebStorage` mellora a seguridade con respecto ás *cookies*, ampliando ademais a cantidade de información que se pode almacenar, sen afectar ao rendemento dos sitios web.

Consiste na emprega dun repositorio común por orixe (mesmo dominio e protocolo), o que supón que todas as páxinas da mesma orixe poderán almacenar e acceder aos mesmos datos.

Tipos

`WebStorage` fornece dous obxectos de tipo `Storage` contidos en `Window`, para tratar a almacenaxe de distintos xeitos:

- **`sessionStorage`:** almacena os datos durante unha sesión (pérdense cando se pecha a ventá do navegador).
- **`localStorage`:** almacena os datos sen data de caducidade.

Denantes empregar `WebStorage` é preciso verificar que o navegador sopórtao:

```
if (typeof(Storage) !== "undefined") {
  // código usando WebStorage
} else {
  // Código se non hai soporte para WebStorage
}
```

Propiedades

- **`length`:** esta propiedade contén o número total de elementos almacenados no obxecto `Storage`.

Métodos

A continuación os métodos que o obxecto `WebStorage` pon á nosa disposición:

- **`setItem()`:** permite engadir un par clave/valor á almacenaxe cliente correspondente ou actualizala no caso de que a clave xa exista.
- **`getItem()`:** devolve o valor da clave indicada polo seu nome.
- **`key()`:** devolve o nome da clave indicada pola súa orde (comezando por 0).
- **`removeItem()`:** elimina o elemento cuxo nome coincide co indicado.
- **`clear()`:** baleira a almacenaxe cliente correspondente para a orixe actual.

```
localStorage.setItem("accesstime", new Date()); // localStorage.accesstime = new
Date();
localStorage.getItem("accesstime"); // Mon Nov 18 2019 14:06:42 GMT+0100
localStorage["accesstime"]; // Mon Nov 18 2019 14:06:42 GMT+0100
localStorage.accesstime; // Mon Nov 18 2019 14:06:42 GMT+0100
localStorage.key(0); // accesstime
localStorage.removeItem("accesstime");
localStorage.clear();
localStorage.length; // 0
```

Atributos HTML5

Un **atributo de datos** é un atributo HTML que funcionalmente actúa coma calquera outro atributo da árbore DOM, pero que é empregado para gardar información non necesariamente relacionada coa estrutura do documento HTML.

Os atributos de datos son accesibles dende JavaScript e CSS, o que nos permite o intercambio de información entre partes das nosas aplicacións web.

Operacións

Para manexar atributos de datos vamos proceder de xeito análogo a que se estiveramos a manipular calquera outro atributo HTML. Supoñamos que temos un obxecto de tipo parágrafo (`p`) con identificador `texto`, e queremos engadirlle un novo atributo:

- **Creación**: empregaremos o método `createAttribute` propio de `document`, con dúas posibilidades:
  - Reemprazando o valor do atributo[5]: creamos un novo atributo denominado `cousa` e engadímolo ao elemento mencionado anteriormente empregando o método `setAttribute`.

```
const novoatributo = document.createAttribute("cousa");
novoatributo.value = "Este é o contido do novo atributo";
document.getElementById("texto").setAttribute(novoatributo.name,
novoatributo.value);
```

  - Reemprazando o obxecto atributo[6]: o código é case idéntico ao do caso anterior, pero empregando o método `setAttributeNode`.

```
const novoatributo = document.createAttribute("cousa");
novoatributo.value = "Este é o contido do novo atributo";
document.getElementById("texto").setAttributeNode(novoatributo);
```

- **Acceso**: coma no caso de calquera outro atributo HTML, accederemos preferiblemente empregando o método `getAttribute`.

```
// Accedemos ao obxecto atributo
document.getElementById("texto").getAttributeNode("cousa");
// Accedemos ao valor do obxecto atributo
document.getElementById("texto"). getAttributeNode("cousa").value;
```

[5] No caso de que xa existira o devandito atributo.

[6] No caso de que xa existira o devandito atributo.

- **Modificación**: podemos asignar un novo valor ao atributo directamente sobre o campo `value` ou a través do método `setAttribute`.

```
// Modificación a través do valor do atributo
document.getElementById("texto").getAttributeNode("cousa").value = "Novo valor do
atributo";
// Modificación a través do método setAttribute
document.getElementById("texto").setAttribute("cousa", "Novo valor do atributo");
```

Restricións de uso

É importante salientar unha serie de restricións de uso con respecto aos atributos de datos:

- Non debemos xamais gardar información privada e/ou confidencial nos mesmos, xa que sempre serán accesibles dende o código da páxina.
- Os nomes dos atributos non poden nunca conter letras maiúsculas, xa que serán automaticamente interpretados coma minúsculas, o que pode implicar conflitos no código JavaScript.

Indexed DB

IndexedDB é unha API de baixo nivel que permite a almacenaxe de datos estruturados no lado do cliente. Co gaio diso, emprega índices que posibilitan a procura rápida de información, resolvendo un dos problemas inherentes a **WebStorage**.

Así mesmo, IndexedDB é unha base de datos asíncrona[7] e orientada a obxectos, polo que o xeito que ten de representar a información vai diferir notablemente das bases de datos relacionais SQL coas que traballamos normalmente. Internamente, IndexedDB representa os datos coma pares clave-valor. Así, as bases de datos vanse organizar en **almacéns de obxectos** (ás veces denominados **almacéns de datos**) onde poderemos establecer restricións estruturais que deberán cumprir os obxectos neles gardados (xeralmente mediante a definición dun atributo clave); de xeito que calquera obxecto que cumpra esas restricións poderá ser almacenado, con independencia de que a súa estrutura non sexa idéntica á do resto de obxectos do almacén.

Cada acción é realizada na forma dunha transacción, xa sexa unha lectura, unha escritura ou algún tipo de cambio. O anterior asegura que as modificacións efectuadas sobre a base de datos realízanse completamente ou non se realizan. Ademais do anterior, as devanditas transaccións son asíncronas, o que asegura que o navegador non

[7] Que as solicitudes á BD sexan asíncronas implica que o navegador non espera até que estean completas despois de realizalas, senón que as solicitudes son invocadas e execútanse nun fío aparte namentres o resto do código do *script* continúa a súa execución en paralelo.

estea deshabilitado durante a operación, podendo o usuario continuar co seu uso con independencia do traballo da base de datos.

Finalmente, e para garantir a seguridade dos datos contidos na súa base, IndexedDB limita o acceso á mesma a aqueles sitios nos que coincida o dominio, a capa de protocolos e o porto de acceso.

Ao longo dos seguintes puntos veremos o código necesario para desenvolver un **CRUD**[8] básico sobre IndexedDB.

Creación das base de datos

Coma ocorre con outros sistemas de bases de datos, o primeiro paso para traballar con IndexedDB é crear unha nova base. Partimos dunha páxina web que chama (no remate do `body`) ao noso *script*, e que posúe un formulario que unicamente contén catro botóns (**Engadir**, **Modificar**, **Ler** e **Eliminar**). No código seguinte podemos ver un exemplo da páxina web da que falamos:

```
<html>
    <head>
        <title>Exemplo T0311: Manexando IndexedDB</title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <form id="formulario">
            <button type="button" id="engadir">Engadir</button>
            <button type="button" id="modificar">Modificar</button>
            <button type="button" id="eliminar">Eliminar</button>
            <button type="button" id="ler">Ler</button>
        </form>
        <ul id="listaxe"></ul>
        <script language="JavaScript" src="indexedDB.js"></script>
    </body>
</html>
```

A continuación vemos o inicio do noso *script*, co código[9] correspondente á creación da BD (**NOSA_BASE**) e un almacén de obxectos (**novo_almacen**):

```
const novaBD = window.indexedDB; // Habilitamos IndexedDB no noso navegador.
const formulario = document.getElementById('formulario');
const btnEngadir = document.getElementById('engadir');
const btnModificar = document.getElementById('modificar');
const btnEliminar = document.getElementById('eliminar');
const btnLer = document.getElementById('ler');

if (novaBD && formulario) {
   let basedatos; // Esta variable vai empregarse para gardar unha referencia á
BD.
```

[8] *CREATE READ UPDATE DELETE*.

[9] Nótese que hai un `if` que figura sen chave de peche porque os seguintes exemplos de código JS do presente punto constitúen un único *script*.

```
   const solicitude = novaBD.open("NOSA_BASE", 1); // Abrimos a BD e gardamos unha
referencia á solicitude resultante.

   // Definimos, cunha función frecha, que hai que facer no caso de que a
solicitude anterior tivera éxito.
   solicitude.onsuccess = () => {
      basedatos = solicitude.result; // Gardamos o resultado da nosa solicitude.
      console.log('APERTURA', basedatos);
   };

   // Definimos o que hai que facer no caso de a BD requira actualización.
   solicitude.onupgradeneeded = () => {
      basedatos = solicitude.result;
      console.log('CREACIÓN', basedatos);
      // Creamos un novo almacén de datos na BD indicando que a clave do mesmo
será o campo 'dni'
      const almacenDatos = basedatos.createObjectStore('novo_almacen', { keyPath:
'dni' });
   };
```

Así, cando executemos por vez primeira o código, na consola do navegador veremos executarse `APERTURA` (que tentará abrir unha BD inexistente) seguido de `CREACIÓN`. A partires dese punto, como a BD xa estará creada, unicamente veremos `APERTURA`.

Inserción nas bases de datos

Neste punto veremos como engadir datos a IndexedDB, o cal faremos sempre en forma de obxectos xa que, coma explicamos ao inicio do apartado, tratamos dun sistema de BB.DD. orientadas a obxectos. Para isto engadiremos o seguinte código (situaríase a continuación do exemplo anterior, dentro do mesmo `if`):

```
   // Definimos unha función que permitirá engadir datos á BD.
   const engadirDatos = (datos) => {
      // Abrimos unha nova transacción contra o almacén de datos 'novo_almacen' da
nosa BD en modo lectura-escritura.
      const transaccion = basedatos.transaction(['novo_almacen'], 'readwrite');
      // Especificamos o almacén no que imos traballar.
      const almacenDatos = transaccion.objectStore('novo_almacen');

      // Engadimos os datos recibidos ao almacén (este método devolve unha
solicitude)
      almacenDatos.add(datos);

   };
```

E no remate do código engadimos o código que da funcionalidade ao escoitador do botón **Engadir**:

```
   // Engadimos o escoitador ao botón 'Engadir'
   btnEngadir.addEventListener('click', (evento) => {
      evento.preventDefault(); // Prevén que o evento quede inútil en caso de
cancelación.
      const persoa = {
         dni: '12345678A',
         nome: 'Perico de los Palotes',
      };
```

```
      engadirDatos(persoa); // Chamamos ao método para engadir datos á BD.

      const persoa2 = {
         dni: '98765432A',
         nome: 'Xoán Castro',
      };

      engadirDatos(persoa2); // Chamamos ao método para engadir datos á BD.

      const persoa3 = {
         dni: '33333333W',
         nome: 'Saínza Pereira',
      };

      engadirDatos(persoa3); // Chamamos ao método para engadir datos á BD.
   });
```

Acceso ás bases de datos

Unha vez creada a nosa BD e engadidos datos á mesma, podemos proceder a extraer información. Co gaio diso engadiremos ao noso *script* dúas funcións, unha para extraer todos os elementos do almacén de obxectos mediante un cursor (`lerDatos()`) e outra para consultar obxectos a partires do seu campo clave (`obterDatos()`).

```
   // Creamos unha función que permitirá ler datos dende a BD.
   const lerDatos = () => {
      // Abrimos unha nova transacción contra o almacén de datos 'novo_almacen' da
nosa BD en modo lectura.
      const transaccion = basedatos.transaction(['novo_almacen'], 'readonly');
      // Especificamos o almacén no que imos traballar.
      const almacenDatos = transaccion.objectStore('novo_almacen');
      // Facemos unha solicitude de datos, para o que empregaremos un cursor.
      const solicitudeLectura = almacenDatos.openCursor();

      // Executamos o seguinte código no caso de que o acceso ao cursor tivese
éxito.
      solicitudeLectura.onsuccess = (evento) => {
         const novoCursor = evento.target.result; // Gardamos o cursor nunha
variable

         // Comprobamos que o cursor existe (é dicir, que aínda ten datos por ler)
         if (novoCursor) {
            console.log(novoCursor.value); // Amosamos o contido do cursor
            novoCursor.continue(); // Avanzamos o cursor ao seguinte elemento
         } else {
            console.log('Rematouse de ler a base de datos');
         }
      };

   };

   // Creamos unha función que nos permitirá recuperar elementos partindo da súa
clave.
   const obterDatos = (clave) => {
      // Abrimos unha nova transacción contra o almacén de datos 'novo_almacen' da
nosa BD en modo só lectura.
      const transaccion = basedatos.transaction(['novo_almacen'], 'readonly');
      // Especificamos o almacén no que imos traballar.
      const almacenDatos = transaccion.objectStore('novo_almacen');
      // Facemos unha solicitude para obter o obxecto asociado á clave pasada coma
parámetro
      const solicitudeDatos = almacenDatos.get(clave);
```

```
      // No caso de que a solicitude sexa correcta executamos o seguinte
      solicitudeDatos.onsuccess = () => {
         if (solicitudeDatos.result) {
            console.log(solicitudeDatos.result);
         } else {
            console.log('Non hai ningunha persoa con ese DNI');
         }
      }
   };
```

E, coma no caso anterior, engadimos o código correspondente ao escoitador do botón na parte inferior do *script*:

```
      // Engadimos o escoitador ao botón 'Ler'
   btnLer.addEventListener('click', (evento) => {
      evento.preventDefault(); // Prevén que o evento quede inútil en caso de
cancelación.
      console.log("LISTAXE DE TODOS OS OBXECTOS CONTIDOS NO ALMACÉN:");
      lerDatos(); // Chamamos ao método que le todo o contido dun almacén.
      console.log("DATOS DO OBXECTO CON CLAVE '33333333A':");
      obterDatos('33333333A'); // Chamamos ao método que le os datos dun obxecto
concreto do almacén.
   });
```

Actualización das bases de datos

Para modificar datos dunha BD coma IndexedDB hai que ter en consideración unha diferenza importante con respecto ás bases de datos relacionais: se solicitamos actualizar os datos dun obxecto contido nun almacén pero resulta non existir ningún obxecto con esa clave no mesmo, o que fará o motor de IndexedDB será inserir o obxecto no almacén, isto é, a actualización de IndexedDB actualiza cando o dato existe no almacén, e insire cando non existe. Neste caso incorporaremos unha función que recibirá o obxecto que queremos actualizar na BD. Vexamos o código:

```
   // Creamos unha función que permitirá actualizar datos xa presentes na BD
   const actualizarDatos = (datos) => {
      // Abrimos unha nova transacción contra o almacén de datos 'novo_almacen' da
nosa BD en modo lectura-escritura.
      const transaccion = basedatos.transaction(['novo_almacen'], 'readwrite');
      // Especificamos o almacén no que imos traballar.
      const almacenDatos = transaccion.objectStore('novo_almacen');
      // Facemos unha solicitude de actualización de datos contra a BD.
      const solicitudeActualizacion = almacenDatos.put(datos);

      // No caso de que a solicitude anterior tivera éxito, executamos o seguinte
código
      solicitudeActualizacion.onsuccess = () => {
         obterDatos(datos.dni);
      };
   };
```

E, unha vez máis, velaquí temos o código do escoitador do botón correspondente:

```
   // Engadimos o escoitador ao botón 'Modificar'
   btnModificar.addEventListener('click', (evento) => {
```

```
      evento.preventDefault(); // Prevén que o evento quede inútil en caso de
cancelación.
      const persoa3 = {
         dni: '33333333A',
         nome: 'Lucía Pereira Pérez'
      };
      actualizarDatos(persoa3);
   });
```

Eliminación nas bases de datos

E xa, coma derradeiro paso no desenvolvemento do noso CRUD sobre IndexedDB, veremos como eliminar obxectos contidos nun almacén, para iso desenvolvemos unha función tal e coma se aprecia no código que figura a continuación:

```
   // Creamos unha función que permitirá eliminar obxectos contidos na BD.
   const eliminarDatos = (clave) =>  {
      // Abrimos unha nova transacción contra o almacén de datos 'novo_almacen' da
nosa BD en modo lectura-escritura.
      const transaccion = basedatos.transaction(['novo_almacen'], 'readwrite');
      // Especificamos o almacén no que imos traballar.
      const almacenDatos = transaccion.objectStore('novo_almacen');
      // Facemos unha solicitude de eliminación de datos contra a BD.
      const solicitudeEliminacion = almacenDatos.delete(clave);

      solicitudeEliminacion.onsuccess = () => {
        lerDatos();
      };
   }
```

E o código[10] do escoitador do botón:

```
   // Engadimos o escoitador ao botón 'Modificar'
   btnEliminar.addEventListener('click', (evento) => {
      evento.preventDefault(); // Prevén que o evento quede inútil en caso de
cancelación.
      eliminarDatos('33333333A');
   });
}
```

Websockets

WebSockets é unha API que permite a comunicación bidireccional, interactiva e síncrona entre navegador e servidor en base a paquetería TCP. Así, empregando WebSockets podemos enviar mensaxes a un servidor e recibir respostas baseadas en eventos (isto permite iniciar a comunicación dende calquera dos extremos, fronte a HTTP, onde só os clientes poden solicitar nova información).

Unha conexión WebSocket é iniciada mediante o envío dunha solicitude de *handshake*[11] ao servidor a través de HTTP co gallo de tornar a conexión nun novo tipo. A

[10] Obsérvese que neste fragmento de código hai unha chave de peche adicional que corresponde ao

partires de que o servidor resposta, a conexión é establecida pero xa non baixo HTTP, senón baixo un protocolo propio (WS). O servidor manterá «vivas» todas as conexións WebSocket que teña cos distintos clientes, podéndose comunicar de xeito independente con cada un deles.

O servidor

No noso caso empregaremos **node.js** para construír o noso servidor WebSocket, tanto pola facilidade que aporta como pola súa grande implantación no mercado actual.

Con ese fin tomaremos os seguintes pasos:

1. Creamos un novo directorio para o noso servidor: no noso caso chamámoslle `servidorWS`.
2. Na terminal situámonos no directorio anterior e creamos un novo proxecto de **node** executando:

```
npm init -y
```

3. Acto seguido instalamos un servidor WebSocket mediante:

```
npm i ws
```

4. Creamos no directorio `servidorWS` un novo *script* de JS para conter o código do noso servidor. Neste exemplo chamarémoslle `servidorWS.js`.
5. Finalmente, incorporamos o seguinte código ao noso *script* do servidor:

```
const WebSocket = require("ws");

const servidorWS = new WebSocket.Server({
    port: 8082
});

// Nesta liña "ws" fai referencia a cada conexión particular
servidorWS.on("connection", socketServidor => {
    console.log("Conectouse un cliente.")

    // Nesta función realízase un intercambio de mensaxes entre cliente e
servidor.
    socketServidor.on("message", mensaxe => {
        // Recibimos a mensaxe do cliente.
        console.log("O cliente enviou a seguinte mensaxe: "+mensaxe)
        // Enviamos unha mensaxe ao cliente.
        socketServidor.send("Ola, Cliente!");
    });

    // Nesta liña indicamos o código a executar cando a conexión pecha dende o
lado do cliente.
```

[11] Empregamos o termo orixinal en inglés dada a habitualidade do seu uso no contexto de redes informáticas.

```
    socketServidor.on("close", () => {
        console.log("O cliente desconectou.")
    });
});
```

O cliente

O noso cliente, pola contra, pode ser desenvolvido enteiramente en JS, sen necesidade de recorrer a librarías e *frameworks* externos (se ben sería posible).

Así, creamos un novo documento HTML co código (por motivos de simplicidade o código JS figura embebido, aínda que recoméndase que, en proxectos maiores, se traballe con ficheiros de *script* independentes):

```
<!DOCTYPE html>
<html lang="es-ES">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Cliente WebSockets</title>
    </head>
    <body>
        <h1>Cliente WebSockets</h1>
        <script language="JavaScript">
            // Créase unha nova conexión ao servidor a través dun WebSocket
            const socketCliente = new WebSocket("ws://localhost:8082");

            // Esta liña execútase cada vez que se produce o evento de conexión a
un servidor.
            socketCliente.addEventListener("open", () => {
                console.log("Conectado ao servidor.");

                // Enviamos unha mensaxe ao servidor.
                socketCliente.send("Ola, Servidor!");
            });

            // Nesta función recibimos mensaxes do servidor.
            socketCliente.addEventListener("message", mensaxe => {
                console.log("O    servidor    enviou    a    seguinte    mensaxe:
"+mensaxe.data);
            });
        </script>
    </body>
</html>
```

Promesas

As **promesas** ou ***promises*** en JavaScript son un tipo de estruturas de código que permiten manexar operacións asíncronas[12] de xeito estruturado en base a diversas características:

- **Uso de estados:** permiten crear estruturas de código máis claras mediante a aplicación de tres estados posibles (pendente -*pending*-, cumprida -*fullfilled*- e rexeitada -*rejected*-).
- **Manexo de erros eficiente:** mediante o uso do método `catch()` no remate dunha cadea de promesas evítase a propagación non controlada de erros, ao tempo que se facilita a depuración de código.
- **Encadeado de operacións:** as promesas poden ser encadeadas nunha secuencia lóxica, o que permite forzar a unha sucesión de operacións que doutro xeito serían asíncronas, a amosar un comportamento síncrono.
- **Programación concorrente:** as promesas permiten executar código en paralelo mediante o uso do método `all()`.
- **Lexibilidade do código:** o uso de promesas produce código máis lexible, escalable e mantible, en comparación co aniñamento de *callbacks*, o cal pode dar no fenómeno coñecido coma *CallBack Hell*[13].

Creación

As promesas créanse mediante a emprega do construtor correspondente da clase **Promise**, o cal vai recibir como parámetro unha función (**`executor`**) que é executada inmediatamente no intre de creación da promesa e que, ao seu tempo, recibe dous argumentos:

- **`resolve`**: función empregada para marcar a promesa coma «cumprida» ou «resolta» (*fullfilled*). O que pasemos neste lugar será o valor de resolución da promesa.
- **`reject`:** función empregada para marcar a promesa como «rexeitada» (*rejected*). Vaise chamar cando ocorra un erro e/ou a operación asíncrona

[12] Unha operación asíncrona e aquela que non se executa en secuencia co fluxo principal do programa, é dicir, que no canto de bloquear a execución do código en espera a que a operación sexa completada, permítese que o programa continúe a executarse, non manexándose o resultado até que a operación asíncrona finaliza.

[13] Un xeito tradicional de programar en JS para forzar a execución secuencial de operacións asíncronas era encadear a execución de funcións pasándoas como parámetro para ser posteriormente chamadas (*callback*). A aplicación en exceso desta técnica produce código difícil de manter e ler (*CallBack Hell*).

non sexa completada correctamente. O que pasemos neste lugar adoitará ser unha mensaxe de erro ou outro tipo de obxecto que describa o motivo do rexeitamento da promesa.

A continuación vemos un exemplo sinxelo de creación dunha promesa que simula unha demora de 2 segundos, devolvendo un texto ao rematar:

```
const minhaPromese = new Promise((resolve, reject) => {
  setTimeout(() => {
    const resultado = "A operación completouse correctamente";
    resolve(resultado);
  }, 2000);
});
```

Uso

Supoñamos agora que temos unha función chamada **obterDatos()** que realiza unha operación asíncrona (como por exemplo unha solicitude a un servidor Web ou a un servidor de BB.DD.). A continuación o código de exemplo:

```
// Definimos a funcion obterDatos
const obterDatos = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { mensaxe: "Os datos foron obtidos correctamente" };
      resolve(datos); // Indicamos o éxito da promesa e retornamos os datos
      // Se quixeramos simular un erro, empregariamos "reject"
      // reject("Erro na obtención de datos");
    }, 2000); // Simulamos unha demora de 2 segundos
  });
};

// Uso da promesa
obtenerDatos()
  .then((resultado) => {
    console.log(resultado.mensaxe);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Vexamos polo tanto que é o que fai o segmento de código anterior:

- Definimos a función **obterDatos()**, a cal devolve unha promesa. Así, na súa cláusula **return** creamos unha nova promesa á cal pasámoslle o **executor** por nós definido.
- No executor simulamos unha operación asíncrona con **setTimeout** e, dentro desa chamada, marcamos a operación como correcta chamando a **resolve** e pasándolle coma argumento os datos obtido.
    - Comentado atopamos un fragmento de código que emula o comportamento en caso de erro, no que chamariamos a **reject**.

- Para empregar o método coa súa promesa utilizamos o método **`then()`**, que manexará a mesma no caso de que teña éxito (en cuxo caso imprimimos os datos obtidos a través da consola).
- Finalmente, usamos o método **`catch()`** para manexar calquera erro que puidera ter ocorrido durante a execución da operación asíncrona. En caso de se producir o tal erro, e se a promesa é rexeitada, imprimiríase a mensaxe de erro correspondente na consola.

Encadeado

Coma mencionamos anteriormente, as promesas poden ser encadeadas para realizar múltiples operacións asíncronas en secuencia, evitando a necesidade dos *callbacks* empregados tradicionalmente. O encadeado realízase mediante múltiples chamadas ao método **`then()`**, e funciona porque o valor devolto por un **`then()`** mediante `return` é pasado automaticamente como argumento do seguinte **`then()`**.

Vexámolo nun exemplo:

```
obterDatos()
  .then((datos) => {
    console.log("Paso 1:", datos.mensaxe);
    return "Máis algún dato";
  })
  .then((maisDatos) => {
    console.log("Paso 2:", maisDatos);
  })
  .catch((erro) => {
    console.error("Erro:", erro);
  });
```

Concorrencia

As promesas permiten execución concorrente de código mediante o uso de **`Promise.all`**, coma introducimos no inicio. Esta estrutura permite manexar múltiples promesas en paralelo e esperar que todas sexan completadas denantes continuar. Con ese fin, devolve unha nova promesa que será resolta cun *array* de resultados cando todas as promesas sexan resoltas correctamente, ou rexeitada se algunha delas é rexeitada.

Vexamos un exemplo:

```
const promesa1 = obterDatos();
const promesa2 = obterDatos2();
const promesa3 = obterDatos3();
```

```
Promise.all([promesa1, promesa2, promesa3])
  .then((resultados) => {
    console.log("Todas as promesas foron completadas:", resultados);
  })
  .catch((erro) => {
    console.error("Alomenos unha promesa foi rexeitada:", erro);
  });
```

Exemplo

A continuación vemos un exemplo práctico de uso de promesas para o encadeamento de tres solicitudes **`fetch`**.

```
// Declaración das URL sobre as que realizaremos cada unha das peticións
const url1 = 'https://jsonplaceholder.typicode.com/posts/1';
const url2 = 'https://jsonplaceholder.typicode.com/posts/2';
const url3 = 'https://jsonplaceholder.typicode.com/posts/3';

// Ésta función realiza unha solicitude (fetch) e devolve unha promesa
function recuperarDatos(url) {
  return fetch(url)
    .then(response => {
      if (!response.ok) {
        throw new Error('Erro na solicitude);
      }
      return response.json();
    });
}

// Realizamos as solicitudes das tres URL anteriores de xeito secuencial
recuperarDatos(url1)
  .then(data1 => {
    console.log('Resposta 1:', data1);
    return recuperarDatos (url2);
  })
  .then(data2 => {
    console.log('Resposta 2:', data2);
    return recuperarDatos (url3);
  })
  .then(data3 => {
    console.log('Resposta 3:', data3);
  })
  .catch(erro => {
    console.error('Error:', erro);
  });
```

Bibliografía

- Curso de JavaScript desde cero (Dorian Desings): https://www.youtube.com/watch?v=tGtxX5x8pKU&list=PLROIqh_5RZeBAnmi0rqLkyZIAVmT5lZxG
- Eloquent JavaScript: https://eloquentjavascript.net/
- Flanagan, D. «JavaScript: the definitive guide». O'Reilly.
- IndexedDB: tutorial for Web Storage: https://www.ionos.com/digitalguide/websites/web-development/indexeddb/
- Introducción a los atributos de datos HTML5: https://ourcodeworld.co/articulos/leer/70/como-guardar-informacion-en-un-elemento-dom-introduccion-a-los-atributos-de-datos-html5
- Osmani, Addy. «Learning JavaScript design patterns». O'Reilly.
- Promise – JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Simpson, K. «You don't know JS: async & Performance». O'Reilly.
- Working with IndexedDB through JS Promises: https://web.dev/indexeddb/
- What is a WebSocket? : https://www.techtarget.com/whatis/definition/WebSocket

ÍNDICE