Sumario

3. *Document Object Model*

El **DOM** (*Document Object Model*) es un modelo que permite navegar por los nodos existentes que forman la página pudiendo manipular sus atributos e incluso crear nuevos elementos. Usando *JavaScript* para navegar en el DOM es posible acceder a todos los elementos de una página lo que permite cambiarlos dinámicamente.

3.1. *Browser Object Model*

El **BOM** (*Browser Object Model*) define una serie de objetos que permiten interactuar con el navegador, como son **window**, **navigator** y **location**.

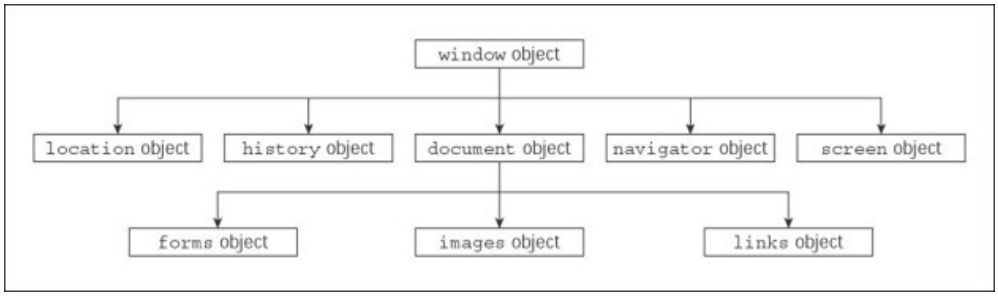

Figura 1: Esquema del modelo **BOM**

3.1.1. Objeto `window`

El objeto **window** es el objeto raíz (global) del navegador. Mediante él se puede:

- abrir una nueva ventana con **window.open()**. Si se hace de este modo el propio navegador bloqueará su apertura tal como sucede con las ventanas emergentes de publicidad. En la actualidad se realizan aplicaciones web con una única ventana por lo que no se utiliza demasiado,
- cerrar una ventana mediante **window.close()**,
- mostrar mensajes de alerta, confirmación y consulta mediante **window.alert(mensaje)**, **window.confirm(mensaje)** y **window.prompt(mensaje [,valorPorDefecto])**.

Ya no se utilizan muchos estos métodos y de hacerlo es posible omitir la palabra **window** ya que siempre hará referencia a la única ventana que está abierta.

3.1.2. Objeto `navigator`

Mediante el objeto **navigator** se accede a propiedades de información del navegador, tales como su nombre y versión (**navigator.appName** y **navigator.appVersion**). Resultan de utilidad si es necesario hacer ajustes entre navegadores y versiones ya que, desafortunadamente, el código no se comporta por igual en todos ellos.

3.1.3. Objeto `location`

Es uno de los objetos más útiles del **BOM**. Debido a la falta de estandarización, **location** es una propiedad tanto del objeto **window** como del objeto **document** y representa la **URL** de la página **HTML** que se muestra en la ventana del navegador. Proporciona varias propiedades útiles para su manejo:

- **href**, cadena que representa la URL completa,
- **protocol**, protocolo de la URL,
- **host**, nombre del host,
- **pathname**, trayectoria del recurso,

- **`search`**, parte que contiene los parámetros, incluido el símbolo **`?`**.

Suponiendo que se accede a la siguiente a la URL:

`http://localhost:63342/Pruebas/bom/location.html?alfa=beta&gama=delta`

```
console.log("href:" + location.href); // Muestra toda la dirección.
console.log("protocol:" + location.protocol); // Muestra http:.
console.log("host:" + location.host); // Muestra localhost:63342.
console.log("pathname:" + location.pathname); // Muestra /Pruebas/bom/ location.html.
console.log("search:" + location.search); // Muestra ?alfa=beta&gama=delta.
```

Si a **`location.href`** le asignamos una nueva URL el navegador realizará una petición a dicha URL y el navegador cargará el nuevo documento.

3.1.4. Objeto `document`

Cada objeto **`window`** contiene la propiedad **`document`** que da acceso al **`DOM`** de una página web y, a partir de él, acceder a los elementos que la forman. Es buena idea cargar los *scripts JavaScript* al final de un documento **`HTML`** para acelerar la carga de una página web y asegurarnos que el **`DOM`** ya está cargado cuando se hace referencia a un objeto. Aunque la mejor manera de hacer esto es a través de eventos, tema que se tratará con posterioridad.

3.2. Trabajando con el DOM

El 99% de las aplicaciones *JavaScript* están orientadas a la web y de ahí su integración con el **DOM**. El **DOM** permite interactuar con el documento **HTML** y cambiar el contenido y su estructura, los estilos **CSS** y gestionar los eventos mediante *listeners* a través de eventos (que se verán en la siguiente unidad). Se trata de un modelo que representa un documento mediante una jerarquía de objetos en árbol y que facilita su programación mediante métodos y propiedades.

Cada elemento, exceptuando el elemento **<html>**, forma parte de otro elemento que se le conoce como padre (*parent*). Un elemento a su vez puede contener elementos hijos (*child*) y/o hermanos (*sibling*).

Suponiendo el siguiente fragmento de código **HTML**:

```
<p><strong>Hello</strong> how are you doing?</p>
```

Cada porción de este fragmento se convierte en un nodo **DOM** con punteros a otros nodos que apuntan sus nodos relativos (padres, hijos o hermanos), del siguiente modo:

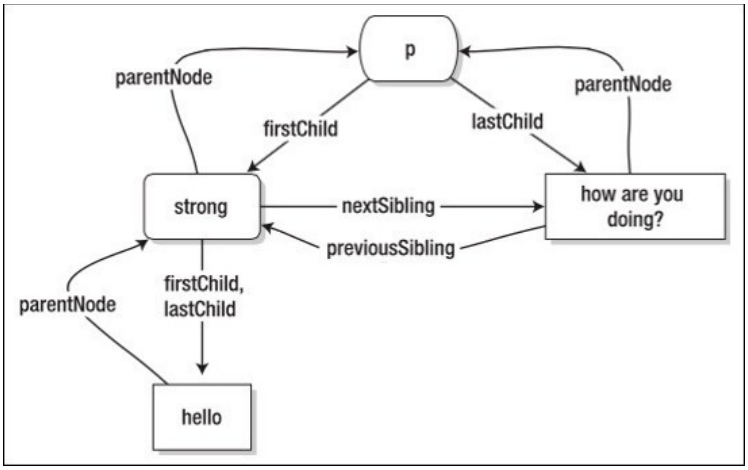

Figura 2: Jerarquía de objetos en el DOM

Cada objeto **DOM** contiene un conjunto de propiedades para acceder a los nodos con lo que mantiene alguna relación. Por ejemplo, cada nodo tiene una propiedad **parentNode** que referencia a su padre (si tiene). Estos padres, a su vez, contienen enlaces que devuelven la referencia a sus hijos, pero como puede tener más de un hijo se almacenan en un objeto iterable (*pseudo-array)* denominado **childNodes**.

Para poder realizar algunos ejemplos de manejo de **DOM**, se dispone de este sencillo documento **HTML**:

```
<!DOCTYPE html>
<html lang="es">
 <head>
    <title>Ejemplo DOM.</title>
    <meta charset="utf-8" />
 </head>
 <body>
    <h1>Encabezado Feo.</h1>
    <p>Primer párrafo.</p>
    <p>Segundo párrafo.</p>
    <div><p id="tres">Tercer párrafo dentro de un div muy feo.</p></div>
```

```
    <script src="dom.js" charset="utf-8"></script>
 </body>
</html>
```

En el documento anterior, **`document.body.childNodes`** contiene cinco elementos: el encabezado **`h1`**, los dos párrafos, la capa (div) y la referencia al *script*. Si el documento **`HTML`** contiene saltos de línea o tabuladores **`DOM`** interpreta estos como nodos de texto, con lo que el número de hijos de un nodo puede no ser el esperado.

Los otros enlaces que ofrece un nodo son:

- **`firstChild`**, primer hijo de un nodo o **`null`** si no tiene hijos,
- **`lastChild`**, último hijo de un nodo o **`null`** si no tiene hijos,
- **`nextSibling`**, siguiente hermano de un nodo o **`null`** si no tiene un hermano a continuación (es el último),
- **`previousSibling`**, anterior hermano de un nodo o **`null`** si no tiene un hermano previo (es el primero).

```
var encabezado = document.body.firstChild;
var scriptJS = document.body.lastChild;
var parrafo1 = encabezado.nextSibling;
var capa = scriptJS.previousSibling;
```

En resumen, los punteros que ofrece un nodo **`DOM`** son:

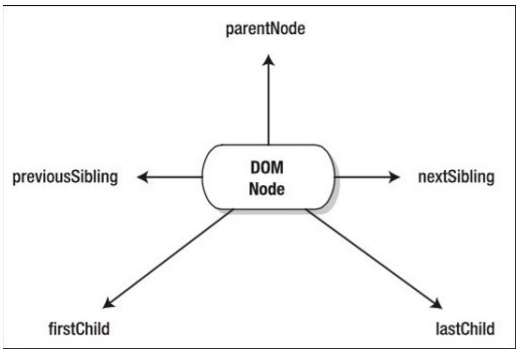

Figura 3: Punteros de un nodo DOM

3.3. Tipos de nodo

Cada uno de los elementos que conforman el **DOM** se conoce como nodo, que pueden ser de los siguientes tipos:

- **Element**, nodo que contiene una etiqueta **HTML**,
- **Attr**, nodo que forma parte de un elemento **HTML**,
- **Text**, nodo que contiene texto y que no puede tener hijos.

Si se observa el código de la capa:

```
 <div>
 <p id="tres">
    Tercer párrafo dentro de un div muy feo.
 </p>
</div>
```

La capa ** ** es un nodo de tipo **element**, el atributo **id** es un nodo de tipo **attr** y el contenido del párrafo es un nodo de tipo **text**. Es posible comprobar el tipo de un nodo a través de su propiedad **nodeType**, que devolverá un número: **1** si es un elemento (nodo **HTML**), **2** si es un atributo y **3** si es de texto. Existen hasta doce tipos de nodo, pero los más usados son estos tres.

Así pues, esta función averigua si es un nodo de texto:

```
function esNodoTexto(nodo) {
 return nodo.nodeType == document.TEXT_NODE; // 3
}
esNodoTexto(document.body); // false
esNodoTexto(document.body.firstChild.firstChild); // true
```

Los elementos contienen la propiedad **nodeName** que indica el tipo de etiqueta **HTML** que representa (siempre en mayúsculas). Los nodos de texto, en cambio, contienen **nodeValue** que obtiene el texto contenido.

```
document.body.firstChild.nodeName; // H1
document.body.firstChild.firstChild.nodeValue; // Encabezado Feo
```

3.4. Seleccionando elementos

Mediante el **DOM** se pueden usar dos métodos para seleccionar un determinado elemento. Para seleccionar un conjunto de elementos, por ejemplo todos los párrafos del documento, es necesario utilizar el método **document.getElementsByTagName(nombreDeTag)**. En cambio, si se accede a un elemento por su **id** (que debería ser único), usaremos el método **document.getElementById(nombreDeId)**.

```
var pElements = document.getElementsByTagName("p"); // NodeList
console.log(pElements.length); // 3
console.log(pElements[0]); // Primer párrafo
var divpElement = document.getElementById("tres");
console.log(divpElement); // "<p id="tres">Tercer párrafo dentro de un div</p>"
```

Destacar que si se necesita un único elemento se selecciona mediante su **id**, mientras que si se necesita más de un elemento se selecciona mediante su etiqueta (*tag*). Con la segunda opción se obtiene un **NodeList** que es un elemento iterable y corresponde con la representación de los elementos del **DOM** seleccionados.

El método de **getElementsByTagName** es antiguo y no se suele utilizar. En el año 2013 se definió el Selector API que define los métodos **querySelectorAll** y **querySelector**. Conviene señalar que **getElementById** es casi cinco veces más rápido que **querySelector**. Estos selectores permiten obtener elementos mediante consultas **CSS** que ofrecen mayor flexibilidad:

```
var pElements = document.querySelectorAll("p");
var divpElement = document.querySelector("div p");
var tresElement = document.querySelector("#tres");
```

3.5. Añadiendo contenido al DOM

Si se necesita añadir un párrafo al documento del ejemplo, primero hay que crear el contenido y luego decidir donde colocarlo. Para ello se usa el método **createElement** para crear el elemento y posteriormente añadir el contenido (por ejemplo, mediante **appendChild**).

```
var elem = document.createElement("p");
texto="<strong>Nuevo párrafo creado dinámicamente.</strong>";
contenido=  document.createTextNode(texto);
elem.appendChild(contenido);
elem.id = "conAppendChild";
document.body.appendChild(elem); // Lo añade como el último nodo detrás de script.
```

En este código las etiquetas **** no se han transformado y en lugar de mostrar el texto en negrita se muestra el código de la etiqueta. Además, no es posible ver el contenido creado dinámicamente y será necesario utilizar las herramientas de desarrollador que ofrecen los navegadores web.

Los métodos para añadir contenidos son:

- **appendChild(nuevoElemento)**, el nuevo nodo se incluye inmediatamente después de los hijos ya existentes (si hay alguno),
- **insertBefore(nuevoElemento, elementoExistente)**, permiten elegir un nodo existente del documento e incluir otro antes que él,
- **replaceChild(nuevoElemento, elementoExistente)**, reemplazar un nodo por otro,
- **removeChild(nodoABorrar)**, elimina un nodo,
- **cloneNode()**, permite clonar un nodo.

Un ejemplo en funcionamiento:

```
var doc = document,
elem = doc.createElement("p"),
contenido = doc.createTextNode("<strong>Nuevo párrafo creado dinámicamente.</strong>"),
pTres = doc.getElementById("tres");
elem.appendChild(contenido);
elem.id = "conAppendChild";
pTres.parentNode.appendChild(elem);
```

En el código anterior:

- se guarda en **doc** la referencia a **document** para evitar tener que salir del alcance y subir al alcance global con cada referencia. Se trata de una pequeña mejora que aumenta la eficiencia del código,
- se genera el nuevo elemento **<p>** y su contenido,

- se crea una referencia al nodo que contiene el párrafo a través de su **id** con **getElementById**,
- se añade el contenido al nuevo nodo,
- se inserta un hijo al padre de **#tres**, lo que lo convierte en su hermano.

Otra forma de añadir contenido es mediante la propiedad **innerHTML**, la cual sí que va a transformar el código incluido en etiquetas reales. Para ello, en vez de crear un elemento y añadirle contenido, el contenido se puede añadir como una propiedad del elemento.

```
var doc = document,
elem = doc.createElement("p"),
pTres = doc.getElementById("tres");
elem.innerHTML = "<strong>Nuevo párrafo reemplazado dinámicamente.</ strong>";
elem.id = "conInner";
pTres.parentNode.replaceChild(elem, pTres);
```

Mientras que **innerHTML** interpreta la cadena como **HTML**, **nodeValue** la interpreta como texto plano, por lo que los símbolos de **<** y **>** no aportan significado al contenido. Pese a que el método **document.write(txt)** permite añadir contenido a un documento, hay que tener mucho cuidado porque, al ejecutarlo en el **DOM**, ya ha cargado el texto y sustituirá todo el contenido que había previamente. Por lo tanto, si se añade contenido, es mejor hacerlo mediante **innerHtml**.

El problema de **innerHTML** es que reemplaza el contenido del nodo con uno nuevo. Para evitar esto se dispone de **insertAdjacentHTML** que no tan sólo respeta la información previa, sino que además ofrece la posibilidad de colocarla en diferentes partes del nodo:

```
<!-- beforebegin -->
<p id=”feo”>
  <!-- afterbegin -->
    Párrafo muy feo.
  <!-- beforeend -->
</p>
<!-- afterend -->
```

Por tanto, el siguiente código introduce un nuevo elemento después del párrafo con **id**=”feo”:

```
var d1 = document.getElementById('one');
d1.insertAdjacentHTML('afterend', '<div id="two">two</div>');
```

Otro dato a tener muy en cuenta es que las referencias con **getElementsBy** están vivas y **siempre contienen el estado actual del documento**, mientras que con **querySelector** obtenemos las referencias **existentes en el momento de ejecución** sin que cambios posteriores en el **DOM** afecten a las referencias obtenidas.

```
var getElements = document.getElementsByTagName("p"),
queryElements = document.querySelectorAll("p");
console.log("Antes con getElements:" + getElements.length);    // 3
console.log("Antes con querySelector:" + queryElements.length);       // 3
var elem = document.createElement("p");
elem.innerHTML = "getElements vs querySelector";
document.body.appendChild(elem);
console.log("Después con getElements:" + getElements.length); // 4
console.log("Después con querySelector:" + queryElements.length); // 3
```

3.6. Gestionando atributos

Una vez se ha recuperado un nodo se accede a sus atributos mediante el método **`getAttribute(nombreAtributo)`** y modificarlo mediante **`setAttribute(nombreAtributo, valorAtributo)`**.

```
var pTres = document.getElementById("tres");
pTres.setAttribute("align","right");
```

También se puede acceder a los atributos como propiedades de los elementos, con lo que se puede hacer lo mismo del siguiente modo:

```
var pTres = document.getElementById("tres");
pTres.align = "right";
```

La gestión de los atributos **está en desuso en favor de las hojas de estilo** para dotar a las páginas de un comportamiento predefinido y desacoplado en su archivo correspondiente. Por ese motivo para modificar los atributos de un nodo se modificará su **CSS**.

Para ello, se dispone del siguiente código **HTML**.

```
<!DOCTYPE html>
<html lang="es">
 <head>
    <meta charset="utf-8" />
    <title></title>
    <style>
       #batman {}
       .css-class{
          color: blue;
          border : 1px solid black;
       }
    </style>
 </head>
 <body>
    <div style="font-size:xx-large" id="batman">Batman siempre gana.</div>
    <script src="css.js"></script>
 </body>
</html>
```

La propiedad **`style`** de los elementos permite obtener/modificar los estilos. Por ejemplo, para cambiar mediante *JavaScript* el color del texto del párrafo a azul y añadirle un borde negro:

```
var divBatman = document.getElementById("batman");
divBatman.style.color = "blue";
divBatman.style.border = "1px solid black";
```

Si la propiedad **CSS** contiene un guion se usa la notación *camelCase*. Así pues, **`background-color`** pasará a **`backgroundColor`**.

Aunque parezca un mejor método para modificarlos, se están modificando los atributos de un elemento del mismo modo que con **`setAttribute()`**. Para conseguir separar la estructura del diseño definitivamente hay que trabajar con clases **CSS**. De hecho, **es el modo recomendado para trabajar con atributos**. Se usará la propiedad **`className`**.

```
var divBatman = document.getElementById("batman");
divBatman.className = "css-class";
```

Si se necesita añadir más de una clase se pueden separar con espacios o utilizar la propiedad **`classList`** que permite añadir clases mediante el método **`add`**.

```
var divBatman = document.getElementById("batman");
divBatman.classList.add("css-class");
divBatman.classList.add("css-class2");
```

Otros métodos útiles de **`classList`** son:

- **`remove`** para eliminar una clase,
- **`toggle`** para cambiar una clase por otra,
- **`length`** para averiguar la longitud de la lista de clases y
- **`contains`** para averiguar si una clase existe dentro de la lista.

De esta forma es posible añadir y eliminar clases **CSS** de un objeto para modificar sus atributos y, con ellos, su apariencia separando de forma definitiva la estructura del estilo.

3.7. Virtual DOM en React

El **DOM** (*Document Object Model*) es la representación gráfica del documento de la aplicación web creada por el navegador, sobre la cual, éste aplica los cambios necesarios en cada actualización de estado o evento.

El **DOM** posee una estructura de diagrama de árbol y al cambiar algún elemento o nodo todos su elementos hijos serán dibujados (renderizados) nuevamente independientemente de si han sido modificados o no. Este modo de actualización puede ser costoso desde el punto de vista del rendimiento si existen muchos elementos hijos y puede ser un problema para aplicaciones muy complejas, especialmente si son cargadas en equipos con bajo rendimiento.

El **Virtual DOM** es una representación del **DOM** guardada en memoria que actúa de intermediario entre los estados de la aplicación y los estados del **DOM** (vistos por el usuario). Cuando ocurre un cambio en la aplicación web, el **virtual DOM** interpreta dichos cambios y calcula la manera más eficiente de actualizar el **DOM** para dibujar la menor cantidad de cambios posibles.

En cada actualización de la aplicación el proceso sigue los siguientes pasos:

- cambio de estado, se produce un cambio en el estado de algún nodo o elemento lo que genera un **Virtual DOM**,
- cómputo de cambios, **React** compara la diferencia entre el estado del **Virtual DOM** y el **DOM** del navegador y detecta los cambios producidos,
- re-dibujado (re-render), se definen los cambios en el **DOM** y la interfaz es actualizada.

La ventaja principal de utilizar **Virtual DOM** como intermediario es que modifica sólo aquellos nodos donde se han producido los cambios ahorrando recursos de procesamiento y brindando una experiencia de usuario más fluida.

3.7.1. useRef

Fruto de la gestión que **React** realiza del **DOM** a través de su propia versión (**Virtual DOM**), el acceso a éste desde un componente está desaconsejado (**casi prohibido**) ya que se redibuja después de cada actualización (como se verá en lo sucesivo). Estos cambios hacen que la selección tan sólo sea precisa durante la carga inicial del código, pero tras unos cuantos cambios es muy probable que los objetos del **DOM** sean remplazados por los del **Virtual DOM** de **React** y se produzcan resultados inesperados.

Para controlar esta situación se dispone de **useRef()** que permite crear referencias dinámicas a elementos del **DOM**, de este modo no se perderá la referencia si se produce un redibujado del árbol que contiene el componente.

```
import { useRef } from react; // Se importa useRef.
function App() { // Se crea un componente App.
 const textoRef = useRef(null); // Se crea una referencia dinámica vacía.
 // Se añade un evento (se verá en el tema siguiente).
 const accion = () => {
    console.log(textoRef.current.value); // Se muestra en un alert lo escrito en el input.
 }
 return (
    <div>
       <input type=”text” ref={textoRef} /> // Referencia al DOM.
       <button onClick={accion}>Mostrar texto</button>
    </div>)
 }
```

```
export default App;
```

Para acceder a las propiedades de la referencia debe realizarse a través de **current**, de este modo si es necesario acceder al valor (**value**) de un **input** tipo **text** se realiza así:

```
textoRef.current.value
```

Esto ocurre con todas las características del objeto referenciado del mismo modo que como si se estuviera accediendo desde **ECMAScript6**.

También es posible modificar el **Virtual DOM** una vez hecha la referencia del mismo modo que se realizaba en el **DOM** original:

```
import { useRef } from react;
function App() {
 const textoRef = useRef(null);
 const resultadoRef = useRef(null); // Se crea otra referencia para contener el texto.

 const accion = () => {
    console.log(textoRef.current.value);
    resultadoRef.current.innerHTML = textoRef.current.value;
    // Se asigna el contenido del input al div.
 }
 return (
    <div>
          <input type=”text” ref={textoRef} />
          <button onClick={accion}>Mostrar texto</button>
          <div ref={resultadoRef} ></div> // Nueva referencia al Virtual DOM.
    </div>)
 }
export default App;
```

Estas referencias, al tratarse de un elemento del código, es posible moverla entre componentes para poder usarlas en componentes descendientes. Esto se realiza a través del objeto **props** o de un contexto (tema que se abordará en lo sucesivo) del siguiente modo:

```
<ComponenteFeo referenciaExterna={textoRef} />
```

Y accediendo a ella dentro del componente `<ComponenteFeo>` del siguiente modo:

```
<input type=”text” ref={props.referenciaExterna} />
```

En caso de necesitar un acceso directo al **DOM** siempre se realizará utilizando las herramientas que **React** dispone, de no ser así el resultado puede ser inesperado y crear muchos dolores de cabeza. Nunca es buena idea usar **getElement** para recorrer el **DOM** desde **React** (ni siquiera es buena idea hacerlo con **useRef**).

3.8. *React-router-dom*

React permite crear SPA (*Single Page Application*) y mostrar varias vistas dentro de la misma página. Aún así, navegación entre páginas, una URL específica para una vista o simplemente retroceder/avanzar en el historial de navegación son características que todo usuario debería poder hacer en un sitio web.

El uso de las rutas en **React** se simplifica mucho gracias a esta biblioteca y a su división en componentes y **hooks**. Se encarga de capturar la barra de navegación del navegador, evitar su funcionamiento por defecto e imponer uno nuevo que será gestionado a través de componentes. No se pretende en este manual estudiar en profundidad esta biblioteca, pero se ahondará lo suficiente para crear un sistema de rutas competente.

Y para todo esto se necesita:

- instalar los paquetes necesarios a través del siguiente comando situado dentro de la carpeta del proyecto:

```
npm install react-router-dom
```

- **<BrowserRoute>**, conecta la aplicación a la URL del navegador, es decir, mantiene la interfaz de usuario en sincronía con la **URL** del navegador mediante la **API History** de **HTML5**,
- **<Routes>**, genera un árbol de rutas que permite reemplazar la vista con el componente que coincide con la **URL** de nuestra barra de navegación y dibuja solamente dicho componente,
- **<Route>**, representa una ruta en el árbol y necesita, al menos, las propiedades **path** y **element**, para representar una ruta,
  - **path** indica la ruta que debe coincidir con la del navegador para cargar el componente especificado. Hay que tener en cuenta que las rutas aquí indicadas funcionan con direccionamiento absoluto y relativas de igual modo que el sistema **Unix** (esto no ocurre en el primer nivel de rutas):
    - si la ruta **comienza con una barra**, la ruta de la barra de direcciones se sustituirá por la que indica el atributo **path** del **<Link>**
    - si la ruta **no empieza con una barra**, la ruta especificada en **path** se añadirá a la ruta existente,
  - **element** especifica el componente que debe dibujar el enrutador,
- **<Outlet>**, sirve para indicar el destino de las rutas anidadas (submenús),
- **<Link>**, sustituye a la etiqueta **<a>** de **HTML** para prevenir el funcionamiento por defecto y evitar que se recargue la página. Recibe en su propiedad **to** el **path** de la ruta a dibujar.
- **useNavigate**, **hook** que permitirá controlar la **API History** del navegador. Se usará para acciones como redirigir al usuario a una ruta determinada en función de alguna condición o acción,
- **useLocation**, ofrece información de las rutas gestionadas por **React-router-dom**,
- **useParams**, permite gestionar los parámetros pasados en una ruta.

Lo ideal es crear las rutas en el componente raíz de la aplicación, que usualmente será **<App>**, ya que será dentro del componente **<Routes>** donde se carguen los componentes que se especifica en cada ruta.

En el siguiente código se define un árbol de rutas con un submenú incluido:

```
<BrowserRouter>
 <Routes>
    <Route path='/' element={<Inicio />} />
    <Route path='login' element={<Login />} />
    <Route path='notas' element={<Notas />}></Route>
    <Route path='herramientas' element={<Herramientas />}>
       <Route path='ciclos' element={<Ciclos />} />
       <Route path='modulos' element={<Modulos />} />
       <Route path='practicas' element={<Practicas />} />
    </Route>
    <Route path='acerca-de' element={<AcercaDe />} />
 </Routes>
</BrowserRouter>
```

El componente **<Routes>**, situado dentro de **<BrowserRoute>**, contendrá todas las rutas que sean necesarias para construir la aplicación. Además, es posible crear rutas anidadas para la creación de submenús incluyendo rutas dentro de un componente **<Route>**, tal y como se ve en el ejemplo.

Una vez creadas las rutas, será necesario crear el enlace que conduce hasta ellas a través del componente **<Link>** del siguiente modo:

```
<nav>
 <Link to="/">Inicio</Link>
 <Link to="/login">Login</Link>
 <Link to="/notas">Notas</Link>
 <Link to="/herramientas">Herramientas</Link>
 <Link to="/acerca-de">Acerca de</Link>
</nav>
```

El dibujado de este submenú se realizará allá donde se encuentre el componente **<Outlet>** más cercano, que suele situarse junto a los componentes **<Link>** que cargan las rutas de ese submenú, tal y como se puede ver en el siguiente componente:

```
const Herramientas = () => {
return (
<Fragment>
<nav>
 <Link to="herramientas/ciclos">Administrar ciclos</Link>
 <Link to="herramientas/modulos">Administrar módulos</Link>
 <Link to="herramientas/practicas">Administrar prácticas</Link>
</nav>
<div className=”contenido”>
 <Outlet />
</div>
</Fragment>
);
};
```

Si en la etiqueta **<Link>** no se hace uso de una ruta anidada, el componente que debe dibujar lo hará donde esté situado **<Routes>**.

Ruta por defecto.

Hay que tener en cuenta que cuando se usa el componente **<Link>** y se activa, éste buscará la ruta de su atributo **to** en la lista de rutas disponibles. Si no encuentra ninguna coincidencia no cargará ningún componente y la aplicación puede verse comprometida.

Para estos casos existe el **path="*"** para poder dibujar un componente por defecto que indique que la ruta no existe o muestre un mensaje de error, por ejemplo.

```
<Route path='*' element={<Error />} />
```

Rutas con parámetro

Para usar parámetros en una ruta debe especificarse en la propiedad **path** del componente **<Route>** del siguiente modo:

```
<Route path='/discente/:nia' element={<Discente />} />
```

De esta forma la ruta será dinámica y pasará el valor de la variable **nia** al componente que carga. En él, en esta ocasión **<Discente>**, se debe usar el **hook useParams** para recoger ese dato y actuar en consecuencia:

```
const { nia } = useParams();
```

Redirección de rutas manual

Para poder enviar a los usuarios a una ruta como resultado de alguna acción es posible utilizar el **hook useNavigate** indicando como parámetro la ruta que debe cargarse. Lo primero es crear el elemento que controlará la navegación:

```
const navigate = useNavigate();
```

Lo segundo usándolo a conveniencia:

```
( sesion_iniciada ? navigate("inicio") : navigate("login") );
```

Esta es una visión muy simple de este **hook** ya que también acepta un objeto como segundo parámetro para personalizar la redirección permitiendo, entre otras cosas, dar información al usuario. Algo a tener en cuenta es que para utilizar este **hook** es necesario hacerlo dentro del ámbito de **<BrowserRouter>** o no tendrá efecto alguno. Si es necesario utilizarlo una configuración adecuada sería convertir al componente **<App>** en *children* de **<BrowserRouter>** para que el ámbito de utilización del **hook** sea toda la aplicación:

```
<BrowserRouter>
 <App />
</BrowserRouter>
```

Utilizando esta biblioteca de esta forma se asegurará que el resultado siga siendo una SAP y evitar recargar la página con todo lo que ello supone. Queda a discreción del discente ahondar sobre las características de esta biblioteca.