3 de julio de 2022

Modificando el documento

La modificación del DOM es la clave para crear páginas "vivas", dinámicas.

Aquí veremos cómo crear nuevos elementos "al vuelo" y modificar el contenido existente de la página.

Ejemplo: mostrar un mensaje

Hagamos una demostración usando un ejemplo. Añadiremos un mensaje que se vea más agradable que un `alert`.

Así es como se verá:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert">
  <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
</div>
```

¡Hola! Usted ha leído un importante mensaje.

Eso fue el ejemplo HTML. Ahora creemos el mismo `div` con JavaScript (asumiendo que los estilos ya están en HTML/CSS).

Creando un elemento

Para crear nodos DOM, hay dos métodos:

`document.createElement(tag)`

Crea un nuevo *nodo elemento* con la etiqueta HTML dada:

```
let div = document.createElement('div');
```

`document.createTextNode(text)`

Crea un nuevo *nodo texto* con el texto dado:

```
let textNode = document.createTextNode('Aquí estoy');
```

La mayor parte del tiempo necesitamos crear nodos de elemento, como el `div` para el mensaje.

Creando el mensaje

Crear el div de mensaje toma 3 pasos:

```
// 1. Crear elemento <div>
let div = document.createElement('div');

// 2. Establecer su clase a "alert"
div.className = "alert";

// 3. Agregar el contenido
div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje.'
```

Hemos creado el elemento. Pero hasta ahora solamente está en una variable llamada `div`, no aún en la página, y no la podemos ver.

Métodos de inserción

Para hacer que el `div` aparezca, necesitamos insertarlo en algún lado dentro de `document`. Por ejemplo, en el elemento `<body>`, referenciado por `document.body`.

Hay un método especial `append` para ello: `document.body.append(div)`.

El código completo:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

```

```
<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje
  document.body.append(div);
</script>
```

Aquí usamos el método `append` sobre `document.body`, pero podemos llamar `append` sobre cualquier elemento para poner otro elemento dentro de él. Por ejemplo, podemos añadir algo a `<div>` llamando `div.append(anotherElement)`.

Aquí hay más métodos de inserción, ellos especifican diferentes lugares donde insertar:

- `node.append(...nodos o strings)` – agrega nodos o strings *al final* de `node`,
- `node.prepend(...nodos o strings)` – insert nodos o strings *al principio* de `node`,
- `node.before(...nodos o strings)` –- inserta nodos o strings *antes* de `node`,
- `node.after(...nodos o strings)` –- inserta nodos o strings *después* de `node`,
- `node.replaceWith(...nodos o strings)` –- reemplaza `node` con los nodos o strings dados.

Los argumentos de estos métodos son una lista arbitraria de lo que se va a insertar: nodos DOM o strings de texto (estos se vuelven nodos de texto automáticamente).

Veámoslo en acción.

Aquí tenemos un ejemplo del uso de estos métodos para agregar items a una lista y el texto antes/después de él:

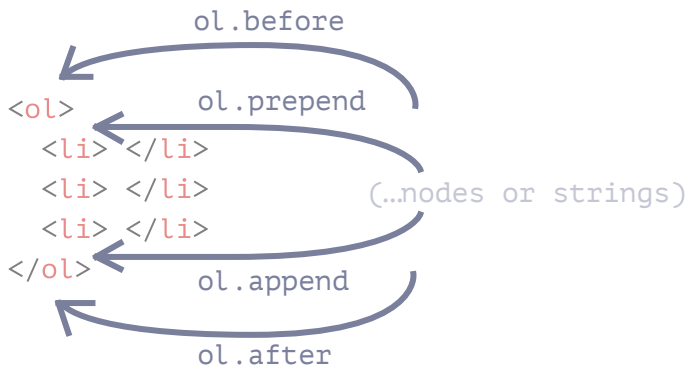
```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before'); // inserta el string "before" antes de <ol>
  ol.after('after'); // inserta el string "after" después de <ol>

  let liFirst = document.createElement('li');
  liFirst.innerHTML = 'prepend';
  ol.prepend(liFirst); // inserta liFirst al principio de <ol>

  let liLast = document.createElement('li');
  liLast.innerHTML = 'append';
  ol.append(liLast); // inserta liLast al final de <ol>
</script>
```

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

Aquí la representación visual de lo que hacen los métodos:



Entonces la lista final será:

```
before
<ol id="ol">
  <li>prepend</li>
  <li>0</li>
  <li>1</li>
  <li>2</li>
  <li>append</li>
</ol>
after
```

Como dijimos antes, estos métodos pueden insertar múltiples nodos y piezas de texto en un simple llamado.

Por ejemplo, aquí se insertan un string y un elemento:

```
<div id="div"></div>
<script>
  div.before('<p>Hola</p>', document.createElement('hr'));
</script>
```

Nota que el texto es insertado “como texto” y no “como HTML”, escapando apropiadamente los caracteres como `<`, `>`.

Entonces el HTML final es:

```
&lt;p&gt;Hola&lt;/p&gt;
<hr>
<div id="div"></div>
```

En otras palabras, los strings son insertados en una manera segura, tal como lo hace `elem.textContent` .

Entonces, estos métodos solo pueden usarse para insertar nodos DOM como piezas de texto.

Pero ¿y si queremos insertar un string HTML "como html", con todas las etiquetas y demás funcionando, de la misma manera que lo hace `elem.innerHTML` ?

insertAdjacentHTML/Text/Element

Para ello podemos usar otro métodos, muy versátil: `elem.insertAdjacentHTML(where, html)` .

El primer parámetro es un palabra código que especifica dónde insertar relativo a `elem` . Debe ser uno de los siguientes:

- `"beforebegin"` – inserta `html` inmediatamente antes de `elem`
- `"afterbegin"` – inserta `html` en `elem` , al principio
- `"beforeend"` – inserta `html` en `elem` , al final
- `"afterend"` – inserta `html` inmediatamente después de `elem`

El segundo parámetro es un string HTML, que es insertado "como HTML".

Por ejemplo:

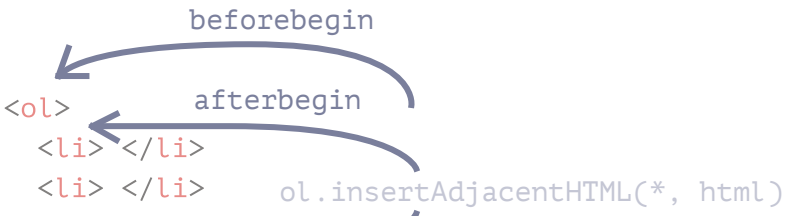
```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Hola</p>');
  div.insertAdjacentHTML('afterend', '<p>Adiós</p>');
</script>
```

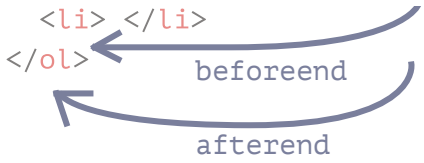
...resulta en:

```
<p>Hola</p>
<div id="div"></div>
<p>Adiós</p>
```

Así es como podemos añadir HTML arbitrario a la página.

Aquí abajo, la imagen de las variantes de inserción:





Fácilmente podemos notar similitudes entre esta imagen y la anterior. Los puntos de inserción son los mismos, pero este método inserta HTML.

El método tiene dos hermanos:

- `elem.insertAdjacentText(where, text)` – la misma sintaxis, pero un string de `texto` es insertado "como texto" en vez de HTML,
- `elem.insertAdjacentElement(where, elem)` – la misma sintaxis, pero inserta un elemento.

Ellos existen principalmente para hacer la sintaxis "uniforme". En la práctica, solo `insertAdjacentHTML` es usado la mayor parte del tiempo. Porque para elementos y texto, tenemos los métodos `append/prepend/before/after` : son más cortos para escribir y pueden insertar piezas de texto y nodos.

Entonces tenemos una alternativa para mostrar un mensaje:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
    <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
  </div>`);
</script>
```

Eliminación de nodos

Para quitar un nodo, tenemos el método `node.remove()` .

Hagamos que nuestro mensaje desaparezca después de un segundo:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
```

```
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje
  document.body.append(div);
  setTimeout(() => div.remove(), 1000);
</script>
```

Nota que si queremos *mover* un elemento a un nuevo lugar, no hay necesidad de quitarlo del viejo.

Todos los métodos de inserción automáticamente quitan el nodo del lugar viejo.

Por ejemplo, intercambiemos elementos:

```
<div id="first">Primero</div>
<div id="second">Segundo</div>
<script>
  // no hay necesidad de llamar "remove"
  second.after(first); // toma #second y después inserta #first
</script>
```

Clonando nodos: cloneNode

¿Cómo insertar un mensaje similar más?

Podríamos hacer una función y poner el código allí. Pero la alternativa es *clonar* el `div` existente, y modificar el texto dentro si es necesario.

A veces, cuando tenemos un elemento grande, esto es más simple y rápido.

- La llamada `elem.cloneNode(true)` crea una clonación "profunda" del elemento, con todos los atributos y subelementos. Si llamamos `elem.cloneNode(false)` , la clonación se hace sin sus elementos hijos.

Un ejemplo de copia del mensaje:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert" id="div">
```

```
  <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
</div>

<script>
  let div2 = div.cloneNode(true); // clona el mensaje
  div2.querySelector('strong').innerHTML = '¡Adiós!'; // altera el clon

  div.after(div2); // muestra el clon después del div existente
</script>
```

DocumentFragment

`DocumentFragment` es un nodo DOM especial que sirve como contenedor para trasladar listas de nodos.

Podemos agregarle nodos, pero cuando lo insertamos en algún lugar, lo que se inserta es su contenido.

Por ejemplo, `getListContent` de abajo genera un fragmento con items `<li>`, que luego son insertados en `<ul>`:

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>
```

Nota que a la última línea `(*)` añadimos `DocumentFragment`, pero este despliega su contenido. Entonces la estructura resultante será:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

Es raro que `DocumentFragment` se use explícitamente. ¿Por qué añadir un tipo especial de nodo si en su lugar podemos devolver un array de nodos? El ejemplo reescrito:

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append +  el operador "..." = ¡amigos!
</script>
```

Mencionamos `DocumentFragment` principalmente porque hay algunos conceptos asociados a él, como el elemento template, que cubriremos mucho después.

Métodos de la vieja escuela para insertar/quitar

> **Vieja escuela**
>
> Esta información ayuda a entender los viejos scripts, pero no es necesaria para nuevos desarrollos.

Hay también métodos de manipulación de DOM de "vieja escuela", existentes por razones históricas.

Estos métodos vienen de realmente viejos tiempos. No hay razón para usarlos estos días, ya que los métodos modernos como `append`, `prepend`, `before`, `after`, `remove`, `replaceWith`, son más flexibles.

La única razón por la que los listamos aquí es porque podrías encontrarlos en viejos scripts:

`parentElem.appendChild(node)`

Añade `node` como último hijo de `parentElem`.

El siguiente ejemplo agrega un nuevo `<li>` al final de `<ol>`:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = '¡Hola, mundo!';

```

```
  list.appendChild(newLi);
</script>
```

`parentElem.insertBefore(node, nextSibling)`

Inserta `node` antes de `nextSibling` dentro de `parentElem`.

El siguiente código inserta un nuevo ítem de lista antes del segundo `<li>`:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = '¡Hola, mundo!';

  list.insertBefore(newLi, list.children[1]);
</script>
```

Para insertar `newLi` como primer elemento, podemos hacerlo así:

```
list.insertBefore(newLi, list.firstChild);
```

`parentElem.replaceChild(node, oldChild)`

Reemplaza `oldChild` con `node` entre los hijos de `parentElem`.

`parentElem.removeChild(node)`

Quita `node` de `parentElem` (asumiendo que `node` es su hijo).

El siguiente ejemplo quita el primer `<li>` de `<ol>`:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  let li = list.firstElementChild;
  list.removeChild(li);
</script>
```

Todos estos métodos devuelven el nodo insertado/quitado. En otras palabras, `parentElem.appendChild(node)` devuelve `node`. Pero lo usual es que el valor no se use y solo ejecutemos el

método.

Una palabra acerca de “document.write”

Hay uno más, un método muy antiguo para agregar algo a una página web: `document.write` .

La sintaxis:

```
<p>En algún lugar de la página...</p>
<script>
  document.write('<b>Saludos de JS</b>');
</script>
<p>Fin</p>
```

El llamado a `document.write(html)` escribe el `html` en la página “aquí y ahora”. El string `html` puede ser generado dinámicamente, así que es muy flexible. Podemos usar JavaScript para crear una página completa al vuelo y escribirla.

El método viene de tiempos en que no había DOM ni estándares... Realmente viejos tiempos. Todavía vive, porque hay scripts que lo usan.

En scripts modernos rara vez lo vemos, por una importante limitación:

El llamado a `document.write` solo funciona mientras la página está cargando.

Si la llamamos después, el contenido existente del documento es borrado.

Por ejemplo:

```
<p>Después de un segundo el contenido de esta página será reemplazado...</p>
<script>
  // document.write después de 1 segundo
  // eso es después de que la página cargó, entonces borra el contenido exister
  setTimeout(() => document.write('<b>...Por esto.</b>'), 1000);
</script>
```

Así que es bastante inusable en el estado “after loaded” (después de cargado), al contrario de los otros métodos DOM que cubrimos antes.

Ese es el punto en contra.

También tiene un punto a favor. Técnicamente, cuando es llamado `document.write` mientras el navegador está leyendo el HTML entrante (“parsing”), y escribe algo, el navegador lo consume como si hubiera estado inicialmente allí, en el texto HTML.

Así que funciona muy rápido, porque no hay una “modificación de DOM” involucrada. Escribe directamente en el texto de la página mientras el DOM ni siquiera está construido.

Entonces: si necesitamos agregar un montón de texto en HTML dinámicamente, estamos en la fase de carga de página, y la velocidad es importante, esto puede ayudar. Pero en la práctica estos requerimientos raramente vienen juntos. Así que si vemos este método en scripts, probablemente sea solo porque son viejos.

Resumen

- Métodos para crear nuevos nodos:
  - `document.createElement(tag)` – crea un elemento con la etiqueta HTML dada
  - `document.createTextNode(value)` – crea un nodo de texto (raramente usado)
  - `elem.cloneNode(deep)` – clona el elemento. Si `deep==true`, lo clona con todos sus descendientes.
- Inserción y eliminación:
  - `node.append(...nodes or strings)` – inserta en `node`, al final
  - `node.prepend(...nodes or strings)` – inserta en `node`, al principio
  - `node.before(...nodes or strings)` –- inserta inmediatamente antes de `node`
  - `node.after(...nodes or strings)` –- inserta inmediatamente después de `node`
  - `node.replaceWith(...nodes or strings)` –- reemplaza `node`
  - `node.remove()` –- quita el `node`.

  Los strings de texto son insertados "como texto".
- También hay métodos "de vieja escuela":
  - `parent.appendChild(node)`
  - `parent.insertBefore(node, nextSibling)`
  - `parent.removeChild(node)`
  - `parent.replaceChild(newElem, node)`

  Todos estos métodos devuelven `node`.
- Dado cierto HTML en `html`, `elem.insertAdjacentHTML(where, html)` lo inserta dependiendo del valor `where`:
  - `"beforebegin"` – inserta `html` inmediatamente antes de `elem`
  - `"afterbegin"` – inserta `html` en `elem`, al principio
  - `"beforeend"` – inserta `html` en `elem`, al final
  - `"afterend"` – inserta `html` inmediatamente después de `elem`

  También hay métodos similares, `elem.insertAdjacentText` y `elem.insertAdjacentElement`, que insertan strings de texto y elementos, pero son raramente usados.
- Para agregar HTML a la página antes de que haya terminado de cargar:
  - `document.write(html)`

  Después de que la página fue cargada tal llamada borra el documento. Puede verse principalmente en scripts viejos.

Tareas

createTextNode vs innerHTML vs textContent

importancia: 5

Tenemos un elemento DOM vacio `elem` y un string `text`.

¿Cuáles de estos 3 comandos harán exactamente lo mismo?

1. `elem.append(document.createTextNode(text))`
2. `elem.innerHTML = text`
3. `elem.textContent = text`

solución

Limpiar el elemento

importancia: 5

Crea una función `clear(elem)` que remueva todo del elemento.

```
<ol id="elem">
  <li>Hola</li>
  <li>mundo</li>
</ol>

<script>
  function clear(elem) { /* tu código */ }

  clear(elem); // borra la lista
</script>
```

solución

Por qué "aaa" permanece?

importancia: 1

En el ejemplo de abajo, la llamada `table.remove()` quita la tabla del documento.

Pero si la ejecutas, puedes ver que el texto "aaa"` es aún visible.

¿Por qué ocurre esto?

```
<table id="table">
  aaa
  <tr>
    <td>Test</td>
  </tr>
</table>

<script>
  alert(table); // la tabla, tal como debería ser
```

```
10
11    table.remove();
12    // ¿Por qué aún está "aaa" en el documento?
13  </script>
```

solución

Crear una lista

importancia: 4

Escribir una interfaz para crear una lista de lo que ingresa un usuario.

Para cada item de la lista:

1. Preguntar al usuario acerca del contenido usando `prompt`.
2. Crear el `<li>` con ello y agregarlo a `<ul>`.
3. Continuar hasta que el usuario cancela el ingreso (presionando `Esc` o con un ingreso vacío).

Todos los elementos deben ser creados dinámicamente.

Si el usuario ingresa etiquetas HTML, deben ser tratadas como texto.

Demo en nueva ventana

solución

Crea un árbol desde el objeto

importancia: 5

Escribe una función `createTree` que crea una lista ramificada `ul/li` desde un objeto ramificado.

Por ejemplo:

```
1  let data = {
2    "Fish": {
3      "trout": {},
4      "salmon": {}
5    },
6
7    "Tree": {
8      "Huge": {
9        "sequoia": {},
10       "oak": {}
11     },
12     "Flowering": {
13       "apple tree": {},
14       "magnolia": {}
```

```
      }
    }
};
```

La sintaxis:

```
let container = document.getElementById('container');
createTree(container, data); // crea el árbol en el contenedor
```

El árbol resultante debe verse así:

- Fish
  - trout
  - salmon
- Tree
  - Huge
    - sequoia
    - oak
  - Flowering
    - apple tree
    - magnolia

Elige una de estas dos formas para resolver esta tarea:

1. Crear el HTML para el árbol y entonces asignarlo a `container.innerHTML`.
2. Crear los nodos del árbol y añadirlos con métodos DOM.

Sería muy bueno que hicieras ambas soluciones.

P.S. El árbol no debe tener elementos "extras" como `<ul></ul>` vacíos para las hojas.

Abrir un entorno controlado para la tarea.

solución

Mostrar descendientes en un árbol

importancia: 5

Hay un árbol organizado como ramas `ul/li`.

Escribe el código que agrega a cada `<li>` el número de su descendientes. No cuentes las hojas (nodos sin hijos).

El resultado:

- Animals [9]
  - Mammals [4]
    - Cows
    - Donkeys
    - Dogs
    - Tigers
  - Other [3]
    - Snakes
    - Birds
    - Lizards
- Fishes [5]
  - Aquarium [2]
    - Guppy
    - Angelfish
  - Sea [1]
    - Sea trout

Abrir un entorno controlado para la tarea.

solución

Crea un calendario

importancia: 4

Escribe una función `createCalendar(elem, year, month)` .

Su llamado debe crear un calendario para el año y mes dados y ponerlo dentro de `elem` .

El calendario debe ser una tabla, donde una semana es `<tr>` , y un día es `<td>` . Los encabezados de la tabla deben ser `<th>` con los nombres de los días de la semana: el primer día debe ser "lunes" y así hasta "domingo".

Por ejemplo, `createCalendar(cal, 2012, 9)` debe generar en el elemento `cal` el siguiente calendario:

MO	TU	WE	TH	FR	SA	SU
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

P.S. Para esta tarea es suficiente generar el calendario, no necesita aún ser cliqueable.

Abrir un entorno controlado para la tarea.

solución

Reloj coloreado con setInterval

importancia: 4

Crea un reloj coloreado como aquí:

hh:mm:ss
Start Stop

Usa HTML/CSS para el estilo, JavaScript solamente actualiza la hora en elements.

Abrir un entorno controlado para la tarea.

solución

Inserta el HTML en la lista

importancia: 5

Escribe el código para insertar `<li>2</li><li>3</li>` entre dos `<li>` aquí:

```
<ul id="ul">
  <li id="one">1</li>
  <li id="two">4</li>
</ul>
```

solución

Ordena la tabla

importancia: 5

Tenemos una tabla:

```
<table>
<thead>
  <tr>
    <th>Name</th><th>Surname</th><th>Age</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>John</td><td>Smith</td><td>10</td>
  </tr>
```

```
    <tr>
      <td>Pete</td><td>Brown</td><td>15</td>
    </tr>
    <tr>
      <td>Ann</td><td>Lee</td><td>5</td>
    </tr>
    <tr>
      <td>...</td><td>...</td><td>...</td>
    </tr>
</tbody>
</table>
```

Puede haber más filas en ella.

Escribe el código para ordenarla por la columna `"name"`.

Abrir un entorno controlado para la tarea.

solución

Lección anterior | Próxima lección

Compartir

Mapa del Tutorial

Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor enviar una propuesta de GitHub o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>`, para varias líneas – envolverlas en la etiqueta `<pre>`, para más de 10 líneas – utilice una entorno controlado (sandbox) (plnkr, jsbin, codepen...)

acerca del proyecto contáctenos