DOM

Курс: JavaScript, часть 2: прототипы и асинхронность

19 февраля 2018 г.

Оглавление

Глава 5

DOM

5.1. Поиск элементов

> **DOM (Document Object Model) – это программный интерфейс для работы с XML- и HTML-документами. Это дерево, узлами которого могут быть атрибуты, элементы, свойства и многие другие.**

Простейшее DOM-дерево представляет собой вот такую обычную HTML страницу.

```
<!DOCTYPE html>
<html>
<head>
 <title>DOM</title>
</head>
<body>
<form class="auth form" id="auth" data-form-value="123"
 ↪  action="/login/">
 <input type="text" value="" name="login">
 <input type="password" value="" name="password">
 <button>Войти</button>
</form>
</body>
</html>
```

DOM позволяет искать элементы, позволяет производить обход дерева, позволяет работать с атрибутами и свойствами узлов, а также добавлять, удалять их и обрабатывать на них события.
Работа с DOM-деревом всегда начинается с поиска элементов.
Метод `getElementById` возвращает нам элемент по его идентификатору. В данном случае мы ищем по идентификатору form и находим наш элемент формы. Объект, который возвращается нам, имеет класс HTMLFormElement.

```
1  document.getElementById('auth') // [object HTMLFormElement]
```

Метод `getElementById` всегда возвращает ровно один элемент, так как по спецификации элемент не может быть повторен с одинаковым идентификатором на странице. И в DOM-дереве всегда должен быть ровно один элемент с одним идентификатором. Как следствие, этот метод самый быстрый.
Следующий метод, который мы рассмотрим — это `getElementByTagName`. Он возвращает элементы с тем тегом, который указан в качестве параметра. В данном случае мы находим все элементы input. И этот метод всегда возвращает объект класса HTMLCollection.

```
1  document.getElementsByTagName('input') // [object HTMLCollection]
```

Этот метод достаточно устаревший и не очень удобный для применения, так как вам редко нужно находить все теги какого-то вида на странице.
На смену устаревшим методам для поиска были добавлены более современные, и прежде всего это метод `querySelector`, который позволяет искать внутри DOM-дерева элементы в зависимости от CSS-селектора.
В данном случае мы ищем нашу форму по ее идентификатору.

```
1  document.querySelector('#auth') // [object HTMLFormElement]
```

Здесь же мы ищем эту же самую форму, но уже по ее классу.

```
1  document.querySelector('.auth') // [object HTMLFormElement]
```

Механизм поиска элементов по CSS-селектору очень гибкий, так как с помощью различных CSS правил и их комбинаций вы можете задавать очень

сложные правила для поиска. Например, в данном случае мы находим все элементы, у которых есть атрибут `name`, значение которого — `login`.

```
document.querySelector('[name="login"]') // [object HTMLInputElement]
```

`querySelector` всегда возвращает первый элемент из всех возможных, найденных внутри DOM-дерева. Для поиска множества элементов разработан метод `querySelectorAll`, который принимает точно также на вход CSS-селектор, но возвращает уже все DOM-элементы, которые этому селектору соответствуют. В данном случае мы ищем все `input` и все `button` на странице и получаем объект класса `NodeList`.

```
document.querySelectorAll('input,button') // [object NodeList]
```

`HTMLCollection`, про которую мы говорили в связи с методом `getElementByTagName`, и `NodeList`, про который мы говорили только что вместе с методом `querySelectorAll`, не являются массивами. Как следствие, у них нет методов для итерирования, таких как forEach, map, reduce или любых других методов работы с массивами. Итак, что же мы можем сделать для итерирования? Самый простой способ для итерирования по коллекциям, и `HTMLCollection`, и `NodeList` — это цикл `for`. И тот и другой являются подобиями массива. И у того и у другого есть свойство `len`, которое говорит о количестве элементов внутри коллекции. И к любому элементу внутри коллекции можно обратиться по индексу.

```
var collection = document.querySelectorAll('input,button');

for (var i = 0, len = collection.length; i < len; i++) {
    var elem = collection[i];
}
```

Следующий вариант — это с помощью метода `Array.prototype.slice` преобразовать нашу коллекцию к настоящему массиву и уже воспользоваться всеми методами, которые есть у массива.

```
var elems = document.querySelectorAll('input,button');

var elemsList = Array.prototype.slice.call(elems);

elemsList.forEach(function(elem) {
    ...
});
```

И последний вариант, который предоставляют нам современные браузеры — это метод `Array.from`, который на вход получает что-то, подобное массиву, и преобразует это в полноценный, настоящий массив. В нашем случае мы получаем массив с элементами и уже можем применить любые методы для работы с массивами к ним.

```
var elemsList =
 ↪  Array.from(document.querySelectorAll('input,button'));

elemsList.forEach(function(elem) {
    ...
});
```

5.2. Атрибуты и свойства

Все элементы в DOM-дереве могут обладать атрибутами или свойствами, для работы со всеми ними разработаны методы, которые позволяют получить к ним доступ.

5.2.1. Star attribute

Мы получаем наш элемент формы с помощью `getElementById`, и впоследствии, когда вы будете видеть переменную `form` — это всегда наш элемент формы. И вызываем у нее метод `getAttribute`. В качестве параметра он получает имя атрибута, который мы хотим, к которому мы хотим обратиться, а в качестве возвращаемого значения в виде строки — содержимое этого атрибута. И таким образом мы получаем значение атрибута `action`.

```
var form = document.getElementById('auth');

form.getAttribute('action'); // '/login/'
```

Также есть метод `hasAttribute`, который проверяет наличие атрибута, и метод `setAttribute`, который устанавливает значение атрибута. При повторном вызове метод has уже вернет нам true, так как новый атрибут уже был добавлен в DOM-дерево. Есть также метод `remove` для удаления атрибута по его имени.

```
form.hasAttribute('method'); // false
form.setAttribute('method', 'POST');
form.hasAttribute('method'); // true
form.getAttribute('method'); // 'POST'
form.removeAttribute('method');
form.hasAttribute('method'); // false
```

5.2.2. data-атрибуты

> **data-атрибуты – атрибуты, которые начинаются с префикса data-, а далее следует название атрибута в виде латинских букв, цифр и знаков дефиса. И с помощью них можно записывать совершенно произвольные значения под совершенно произвольным именем.**

```
form.getAttribute('data-form-value'); // '123'

form.dataset.formValue; // '123'
form.dataset.hasOwnProperty('formValue'); // true

form.dataset.fooBarBazBaf = 'boo'; // data-foo-bar-baz-baf="boo"
form.hasAttribute('data-foo-bar-baz-baf'); // true
```

Существует специальный объект `dataset`, который есть внутри любого DOM-элемента, который содержит значения всех data-атрибутов, которые есть внутри этого элемента. Он преобразовывает имя атрибута, убирает от него префикс

data-, а все дефисы и последующие буквы преобразуют к CamelCase. Таким образом имя атрибута становится валидным идентификатором для переменной в JavaScript, и мы можем обращаться к нему через так называемую дотнотацию. И мы можем с помощью этого как обратиться к атрибуту на чтение, так и установить новое значение атрибута, которые будет прописано в DOM-дерево уже в развернутом data- виде.

5.2.3. Свойства

У всех элементов есть как атрибуты, так и свойства. И порой это совершенно разные вещи.

```
form.getAttribute('action'); // "/"
form.getAttribute('method'); // null
form.getAttribute('id'); // "auth"


form.action; // "https://yandex.ru/login/"
form.method; // "get"
form.id; // "auth"
```

Порой атрибуты и свойства ведут себя очень непредсказуемо, и значения у них могут сильно отличаться. Поэтому лучше использовать атрибуты в большинстве случаев вместо свойств, а если вы хотите пользоваться свойствами, то использовать самые простые и общие свойства: `id` и `className`.

5.2.4. Классы

Атрибут `class` — это, пожалуй, один из самых широко используемых атрибутов в html. С помощью него вы можете помечать элементы для того, чтобы наложить на них css-свойства и настроить их визуальное отображение, для того чтобы выбрать их с помощью селектора, и вообще объединить в какие-то логически связанные группы.
Для работы с атрибутом `class` есть свойство `className`, которое в строковом представлении содержит значение атрибута `class` в html и представляет весь набор классов, который указан у этого html-элемента в виде строки.

```
form.className; // 'auth form'
form.className += ' login-form'; // 'auth form login-form'
```

Но пользоваться им не очень удобно. Для того чтобы отредактировать набор классов: убрать класс, добавить класс, вам необходимо устанавливать это свойство и заменять строчку уже существующую на новую с помощью конкатенации, с помощью регулярных выражений или с помощью замены внутри строки. Это усложняет и поиск, и выборку, и проверку наличия класса.
Поэтому разработчики создали новый, достаточно современный метод `classList`, который содержит набор методов для работы с классами:

- `add`, который добавляет класс к списку уже существующих;
- `item`, который получает класс по его индексу и по порядковому расположению его в html-элементе;
- `contains`, который позволяет вам по имени класса проверить, есть ли он внутри этого элемента;
- `remove`, который позволяет вам удалить класс из html-элемента.

```
form.classList.add('login-form');
form.classList.item(1); // 'form'
form.classList.item(2); // 'login-form'
form.classList.contains('login-form'); // true
form.classList.remove('login-form');
```

5.3. Создание элементов

Давайте теперь поговорим о создании элемента. Страницы становятся все более интерактивными, и все больше элементов на них появляется динамически в зависимости от каких-то действий пользователя: мы получили ответ от сервера, отрисовали элемент, наполнили его текстом и таким образом сообщили пользователю о том, что произошло.
Для этого существует несколько методов. Прежде всего, это метод `createElement`, который, собственно говоря, отвечает за создание элемента, и метод `appendChild`, который добавляет элемент внутрь нашей страницы.

Сначала мы создаем элемент `span` с помощью `createElement` и конфигурируем наш элемент.

```
var elem = document.createElement('span');

elem.className = 'error';
elem.setAttribute('id', 'auth-error');
elem.setAttribute('status', 'auth-error');
elem.textContent = 'Введен неверный логин или пароль';
```

Важно понимать: созданный элемент находится в памяти и не находится внутри тела страницы (не связан с DOM-деревом).
С помощью `appendChild` мы можем добавить наш элемент к любому другому, как вложенный новый дочерний элемент. В нашем случае мы добавляем созданный и отконфигурированный `span` последним элементом к тегу `body`, то есть добавляем в самый конец страницы.

```
document.body.appendChild(elem);
```

При создании нового DOM-элемента нам не всегда нужно создавать его с нуля: можно взять за основу уже существующий, который есть на странице, скопировать его и поменять в нем какие-то свойства.
Для этого существует метод `cloneNode`, который принимает в качестве необязательного параметра true или false. False — это значение по умолчанию, а если мы передадим true, то все дочерние элементы внутри нашего элемента, который мы хотим склонировать, будут скопированы. Все это будет помещено во вновь созданный склонированный объект. И после этого мы можем так же, как у любого другого элемента, у этой копии поменять атрибуты и свойства, это никоим образом не повлияет на родительский элемент, с которого мы сняли снимок, и с помощью метода `appendChild` также добавить его на страницу.

```
var clone = elem.cloneNode(true);
clone.id = 'mail-error';
clone.textContent = 'Не удалось отправить письмо';

document.body.appendChild(clone);
```

Более подробно вы можете ознакомиться с методами по ссылкам на документацию.

5.4. События в DOM

События — это то, ради чего мы работаем с DOM-элементами вообще: на любых современных страницах множество событий, которые обрабатываются.
Так или иначе практически все DOM-элементы могут реагировать на те или иные события: загрузки, на которое реагирует вся страница, событие загрузки какого-то элемента, событие загрузки вашего JavaScript.
Некоторые DOM-элементы могут реагировать на какие-то события, которых нет у других. Например, событие `submit` есть только у формы. С полным списком событий вы можете ознакомиться в документации.
Самый простой способ добавить обработчик события на какой-то элемент — это явно прописать его в качестве атрибута у этого элемента. Для каждого события есть атрибут с префиксом on, который будет ему соответствовать: onsubmit у формы, onclick, например, у баттона. А далее вы указываете в качестве значения атрибута js-функцию, которая будет вызвана в качестве обработчика этого события.

```
1  <form class="auth form" id="auth" data-form-value="123"
   ↪  action="/login/" onsubmit="submitHandler()">
2
3  <input type="text" value="" name="login">
4
5  <input type="password" value="" name="password">
6
7  <button onclick="clickHandler()">Войти</button>
8
9  </form>
```

Это рабочий способ, и он будет работать даже в самых старых браузерах, но я бы настоятельно не рекомендовал вам им пользоваться:

- вы не можете навешать более одного обработчика на какое-то конкретное событие;
- JS код и ваш HTML код становятся очень тесно связаны;

- меняя HTML код или меняя JavaScript код, вам очень важно всегда помнить, что вы не можете переименовать значение/тип атрибута, иначе это будет другое событие;
- вы не можете навешивать эти обработчики динамически, должны изначально присутствовать в теле HTML-документа.

Для решения всех этих проблем разработчики стандартов W3C создали методaddEventListener, который добавляет слушателя на какое-то событие, которое может произойти на DOM-элементе. В нашем случае мы добавляем для формы на событие `submit`, которое указывается в качестве первого параметра, слушателя, который будет являться js-функцией.

```
var form = document.getElementById('auth');
form.addEventListener('submit', submitHandler);
```

Но обработчиков может быть значительно больше, и вы можете добавить их целый пул, таким образом создать стек обработки событий и разбить вашу обработку на несколько отдельных функций. В каждую из этих функций в качестве первого параметра будет передан объект `event`.

```
var button = document.querySelector('button');

button.addEventListener('click', function(event) {
    console.log(this, event);
});

button.addEventListener('click', clickHandler);

button.addEventListener('click', yetAnotherClickHandler);
```

Все обработчики всегда вызываются в том порядке, в котором они добавляются в качестве слушателей.
Контекстом любой функции обработки события всегда будет являться DOM-элемент, на котором это событие произошло, и на котором был вызван `addEventListener` для того, чтобы его послушать. Как я уже говорил, в любую функцию обработчика событий в качестве первого параметра передается объект события. Обычно эту переменную называют `event` для того, чтобы не путать со всеми

остальными. И у нее есть несколько свойств, которые для нас особенно важны. Прежде всего это свойство `target`, которое указывает на фактический DOM-узел — самый последний листок DOM-дерева, на котором произошло событие. Свойства `altKey`, `ctrlKey`, `shiftKey`, которые говорят о том, была ли нажата соответствующая клавиша во время события и значение type, которое указывает строкой тип события, который фактически сейчас произошел.
Существует два различных, противоположных способа обработки событий: Bubbling и Capturing.

Всплытие события или Bubbling

Это значение, которое принимается по умолчанию в браузерах, и тот вариант обработки событий, который наиболее распространен. При нем событие как бы поднимается вверх, от самого глубокого DOM-узла — к самому верхнему и заканчивается на объекте `document`, который описывает весь наш HTML документ. Для того, чтобы событие всплывало, в качестве последнего параметра в `addEventListener` нужно передать `false`, либо не передавать ничего, так как это значение по умолчанию.

Перехват событий или Capturing

При этом обработчике событий событие погружается от самого верхнего элемента к самому глубокому, самому нижнему, на котором оно могло бы произойти, и сначала срабатывает на объекте `document` и постепенно погружается вниз.
Для того, чтобы событие перехватывалось, в качестве последнего параметра в `addEventListener` вам нужно передать `true`.
Иногда вы хотите остановить событие на каком-то конкретном участке и не хотите, чтобы оно всплывало или погружалось далее. Для того, чтобы остановить всплытие, существует метод `stopPropagation`, который вызывается у объекта «событие». То же самое можно сделать с Capturing вызовом того же метода: событие не будет погружаться дальше.

```
1 var button = document.querySelector('button');
2 button.addEventListener('click', function(event) {
3     event.stopPropagation();
4 }, true); // Capturing
```

Множество событий могут быть назначены на одном элементе, и тогда эти обработчики выстроятся в стек и будут вызываться постепенно, друг за другом, в порядке их добавления. Но если мы этого не хотим, если мы хотим остановить все обработчики в рамках этого элемента, а также не давать им всплывать или погружаться, то мы можем вызвать метод `stopImmediatePropagation` на требуемом элементе. Тогда стек будет опустошен, и событие перестанет всплывать/погружаться. Но пользоваться этим методом нужно очень осторожно, так как вы можете удалить обработчик, который был поставлен другим человеком.
У многих DOM-элементов есть так называемые действия по умолчанию. Например, при клике на ссылки, мы переходим по ссылке, при клике по кнопке мы отправляем форму.Для того чтобы отменить действие по умолчанию, существует метод `preventDefault`, который вызывается у объекта-события.

```
1 var form = document.getElementById('auth');
2
3 form.addEventListener('submit', function(event) {
4     console.log(event.target);
5     event.preventDefault();
6 });
```

5.4.1. Делегировние событий

> **"Делегировние событий" – метод обработки события на родительском элементе относительно того, на котором оно произошло фактически.**

Например, у нас есть список, состоящий из множества ссылок, которые появляются динамически, в зависимости от каких-то действий пользователя.

```
<ul>
<li><a href="#maps">Карты</a></li>
<li><a href="#market">Маркет</a></li>
<li><a href="#news">Новости</a></li>
<li><a href="#translate">Переводчик</a></li>
<li><a href="#images">Картинки</a></li>
<li><a href="#video">Видео</a></li>
<li><a href="#music">Музыка</a></li>
<li><a href="#more" class="more">ещё</a></li>
</ul>
```

Мы бы могли получить все ссылки, пойти по ним в цикле и на каждую назначить свой обработчик событий. Но при добавлении любой новой нам бы приходилось каждый раз делать это заново, а при удалении какого-то элемента нам нужно было бы удалять за ним и события.
Мы можем поступить иначе: назначить обработчик на весь наш список и ловить конкретные события на конкретных элементах. В нашем случае это делегирование можно было бы оформить следующим образом.

```
document.addEventListener('click', function(event) {
    event.preventDefault();

    if (event.target.tagName === 'A') {
        if (event.target.classList.contains('more')) {
            openMorePopup();
        } else {
            console.log(event.target.href);
        }
    }
});
```

Мы начинаем слушать событие клика на объекте `document`, для которого всплывают все события. После этого мы отменяем действия по умолчанию (переход по ссылке). Далее мы обращаемся к свойству `event.target` и получаем фактически элемент, на котором произошло событие, то есть то, куда мы в итоге кликнули, и проверяем, является ли он ссылкой. А далее, с помощью метода `classList` проверяем, есть ли у этой ссылки тот или иной класс. И в зависимости от этого, либо открываем наш попап, либо переходим по ссылке.

Делегирование — достаточно мощный инструмент, который позволяет навешивать вам всего один обработчик на один тип события и получать их с множества различных узлов. Эти узлы могут создаваться динамически, могут удаляться, но не надо переживать за то, добавили ли вы обработку событий на него или нет — любое событие всплывет и может быть обработано на объекте `document` как самой высшей инстанции для обработки события. Единственное, что вам стоит при этом учитывать — это то, что у объекта события нет методов для проверки того, на каком фактически элементе оно произошло и есть ли у этого элемента какой-то класс/ CSS-селектор. И из-за этого приходится городить конструкции в виде нескольких if'ов, проверять явно. Рекомендую поступать следующим образом: получать `event.target`. Так как это может быть глубокий вложенный элемент, всегда помнить об этом, подниматься на определенное количество уровней выше до какого-то элемента с каким-то классом, который вас гарантированно интересует. Например, в нашем случае это мог бы быть тег `le` с элементом списка. И уже на нем проверять события и навешивать обработчики.