

Основы API, итоги курса

История появления API устройств

За более чем 25 лет с момента создания интернета мы стали свидетелями его распространения на рабочие станции, настольные ПК, ноутбуки, мобильные телефоны, планшеты, а теперь даже на модные аксессуары, такие как часы. Нет никаких сомнений, что интернет никуда не денется.

Действительно, мобильная революция полностью изменила способ доступа людей в интернет. Появление смартфонов с полноценными версиями браузеров привело к экспоненциальному росту аудитории интернета, повысило её мобильность. Увеличилась продолжительность нашего пребывания в сети. Имея смартфон, любой из нас может быть онлайн круглосуточно.

Вполне естественно, что интернет-технологии должны развиваться, чтобы лучше соответствовать возможностям современных устройств, на которых они (технологии) теперь доступны.

На этом уроке мы познакомимся с некоторыми популярными JavaScript API устройств, способными расширять функциональность браузера и обладающими хорошим уровнем поддержки в современных браузерах.

Геолокация

API геолокации позволяет определять местоположение пользователя за счёт позиционирования его устройства. В большинстве случаев позиционирование будет выполняться с использованием GPS, но также могут использоваться менее точные методы, такие как определение местоположения на основе Wi-Fi.

Для соблюдения конфиденциальности у пользователя будет запрошено разрешение на сбор данных о местоположении.

API геолокации доступен через объект `navigator.geolocation`.

Если объект существует, функции определения местоположения доступны. Проверить это можно следующим образом:

```
if ('geolocation' in navigator) {  
  /* местоположение доступно */  
} else {  
  /* местоположение недоступно */  
}
```

```
}
```

Получение текущего местоположения

Чтобы получить текущее местоположения пользователя, надо вызвать метод **getCurrentPosition()**. Это инициирует асинхронный запрос для обнаружения местоположения пользователя и задействует аппаратные средства позиционирования, чтобы получить последнюю актуальную информацию. Когда местоположение определено, выполняется функция обратного вызова (callback). По желанию вы можете указать вторую функцию для обработки ошибки. Третий, опциональный, параметр — объект с полями, где можно настроить максимальное значение возвращаемых данных, время ожидания ответа на запрос и точность возвращаемых данных.

```
navigator.geolocation.getCurrentPosition((position) => {  
  const {latitude, longitude} = position.coords  
  console.log('Географические координаты устройства', latitude, longitude)  
})
```

Важно! По умолчанию **getCurrentPosition()** пытается вернуть результат как можно скорее. Это полезно, когда критична скорость ответа и не так важна точность. Устройства с GPS, например, могут пытаться скорректировать данные около минуты и даже дольше, поэтому в самом начале **getCurrentPosition()** может вернуть менее точные данные, полученные на основе [AGPS](#) мобильного провайдера или сети Wi-Fi.

Наблюдение за текущим местоположением

Если мы хотим следить за изменениями местоположения устройства, это делается с помощью функции **watchPosition()**, которая имеет три входных параметра, похожих на **getCurrentPosition()**. Переданные ей в параметрах callback-функции вызываются много раз, позволяя браузеру обновлять данные о текущей локации либо во время движения, либо после получения более точной информации о местоположении после применения более точных методов позиционирования.

Важно! Использовать **watchPosition()** можно и без вызова **getCurrentPosition()**.

```
let watchId = navigator.geolocation.watchPosition(({coords}) => {  
  console.log('Устройство обновило местоположение', coords.latitude,  
    coords.longitude)  
})
```

Метод **watchPosition()** возвращает числовой идентификатор для использования вместе с методом **clearWatch()**, чтобы отписаться от получения новых данных о местоположении.

```
navigator.geolocation.clearWatch(watchId)
```

Точная настройка позиционирования

Методы **getCurrentPosition()** и **watchPosition()** принимают третьим параметром необязательный объект **PositionOptions** со следующими полями:

1. **enableHighAccuracy** (Boolean) — точность определения позиции поле.
2. **maximumAge** (Number) — максимальное время кеширования в миллисекундах. При повторных запросах, пока время не вышло, будет возвращаться кешированное значение, а после браузер будет запрашивать актуальные данные.
3. **timeout** (Number) — максимальное время ожидания ответа при определении позиции. Число миллисекунд, спустя которое будет вызван обработчик ошибки.

Вызов **watchPosition** может выглядеть следующим образом:

```
const handlePositionSuccess = ({coords}) => {
  console.log('Устройство обновило местоположение', coords.latitude,
    coords.longitude)
}

const handlePositionError = (error) => {
  console.log('Извините, местоположение недоступно', error)
}

const positionOptions = {
  enableHighAccuracy: true,
  maximumAge: 30000,
  timeout: 1000 * 30,
}

const watchId = navigator.geolocation.watchPosition(
  handleSuccessLocation,
  handleLocationError,
  positionOptions
)
```

Обработка ошибок

Callback-функция обработки ошибок, если она была передана в **getCurrentPosition()** или **watchPosition()**, ожидает экземпляр объекта **GeolocationPositionError** в качестве первого аргумента. Он будет содержать два свойства:

- **code** — условное обозначение ошибки, которая произошла;

- **message** — понятное для человека описание ошибки, названной в поле **code**.

Функция выглядит примерно так:

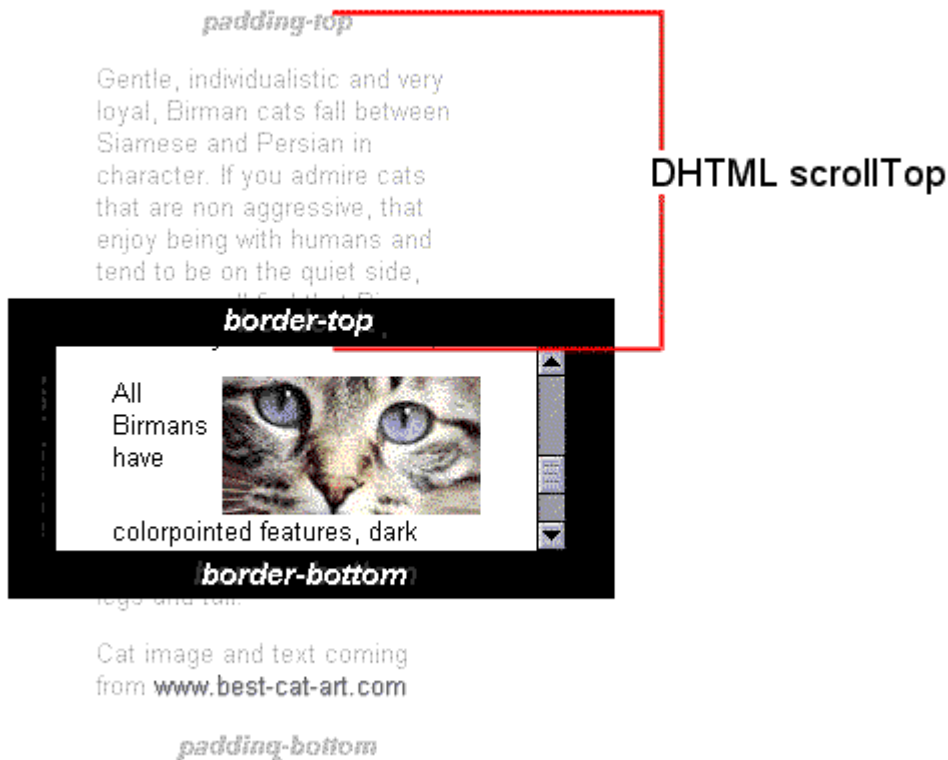
```
const handlePositionError = (error) => {
  switch (error.code) {
    case 1:
      console.log('Пользователь ограничил доступ к местоположению')
      break
    case 2:
      console.log('Ошибка устройства определения местоположения')
      break
    case 3:
      console.log('Достигнут тайм-аут')
      break
    default:
      console.log('Извините, местоположение недоступно', error)
      break
  }
}
```

Работа со скроллом

DOM-интерфейсы **Window** и **Element** содержат несколько полей и методов для работы с полосой прокрутки.

Свойства **Element.scrollTop** и **Element.scrollLeft**

Свойства **Element.scrollTop** и **Element.scrollLeft** возвращают или устанавливают расстояние от начальной точки содержимого элемента (padding-box элемента) до начальной точки его видимого контента. Когда контент элемента не создаёт полосу прокрутки, его **scrollTop** или **scrollLeft** равен нулю.



Значения свойств **Element.scrollTop** и **Element.scrollLeft** могут быть любым целым числом, но с определёнными оговорками:

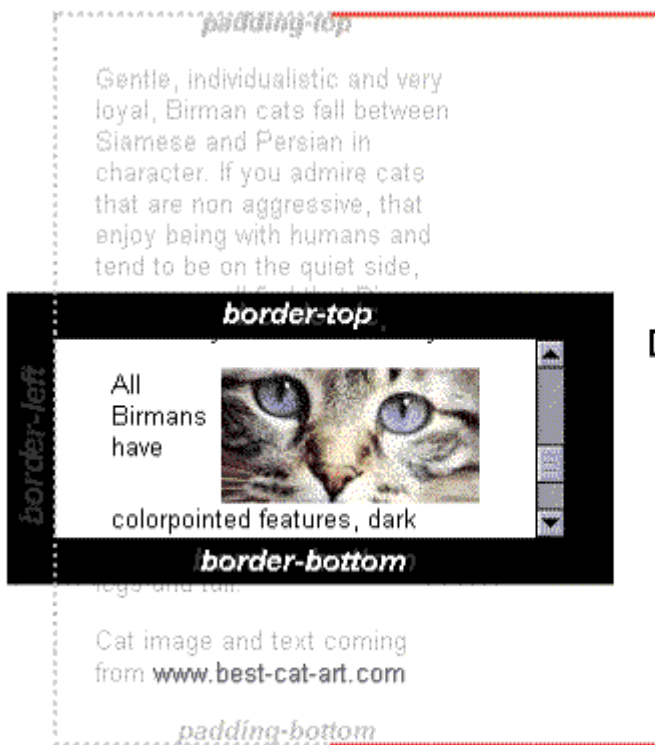
1. Если элемент не прокручивается, у него нет переполнения или мы не прокручиваем элемент, **scrollTop** (**scrollLeft**) устанавливается в 0.
2. Если значение меньше нуля, **scrollTop** (**scrollLeft**) устанавливается в 0.
3. Если установленное значение больше максимума прокручиваемого контента, **scrollTop** (**scrollLeft**) устанавливается в максимум.

```
const box = document.querySelector('#box')
console.log(box.scrollTop, box.scrollLeft)

// Устанавливаем количество прокрученных пикселей
box.scrollTop = 500
```

Свойства Element.scrollHeight и Element.scrollWidth

Свойства **Element.scrollHeight** и **Element.scrollWidth** (только чтение) содержат высоту и ширину содержимого элемента соответственно, включая содержимое, невидимое из-за прокрутки. Значение **scrollHeight** (**scrollWidth**) равно минимальному **clientHeight** (**clientWidth**), необходимому, чтобы поместить всё содержимое элемента в видимую область, не используя полосу прокрутки. Это значение включает в себя padding элемента, но не его margin.



DHTML scrollHeight

Как пример использования свойства **Element.scrollHeight** рассмотрим код, который определяет, прочитал пользователь текст или нет.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
#notice {
  display: inline-block;
  margin-bottom: 12px;
  border-radius: 5px;
  width: 600px;
  padding: 5px;
  border: 2px #7FDF55 solid;
}
#rules {
  width: 600px;
  height: 130px;
  padding: 5px;
  border: #2A9F00 solid 2px;
  border-radius: 5px;
}
    </style>
    <title>Регистрация на сайте</title>
  </head>
  <form name="registration">
```

```

<p id="notice">Пожалуйста, прокрутите и прочитайте следующий текст.</p>
<p>
  <textarea id="rules">
Регистрируясь на сайте, я соглашаюсь со следующими условиями:
Условие 1
Условие 2
...
Условие 99
Условие 100
  </textarea>
</p>
<p>
  <input type="checkbox" id="agree" />
  <label for="agree">Я согласен</label>
  <input type="submit" id="nextstep" value="Далее" />
</p>
</form>
<script>
const rulesElement = document.getElementById('rules')
const agreeCheckbox = document.getElementById('agree')
const nextStepButton = document.getElementById('nextstep')
const {scrollHeight, scrollTop, clientHeight} = rulesElement
let isRead = false

const handleCheckReading = () => {
  if (isRead) {
    return
  }
  isRead = scrollHeight - scrollTop === clientHeight
  agreeCheckbox.disabled = nextStepButton.disabled = !isRead
}
rulesElement.addEventListener('scroll', handleCheckReading, false)
</script>
</body>
</html>

```

Методы scroll, scrollTo, scrollBy

Есть три основных метода, которые позволяют программно управлять полосой прокрутки. При использовании с объектом **Window** они позволяют управлять прокруткой всего документа, а с DOM-элементами — регулировать прокрутку содержимого элемента.

Методы поддерживают два типа синтаксиса: **scrollTo(x-coord, y-coord)** и **scrollTo(options)**:

1. **x-coord** — координаты пикселя по горизонтальной оси документа или элемента, который надо отобразить вверху слева.
2. **y-coord** — координаты пикселя по вертикальной оси документа или элемента, который надо отобразить вверху слева.

3. **options** — объект с тремя возможными параметрами:
 - a. **top** — то же, что и **y-coord**;
 - b. **left** — то же, что и **x-coord**;
 - c. **behavior** — строка, содержащая либо **smooth**, **instant**, либо **auto** (по умолчанию — **auto**).

```
window.scrollTo(0, 1000)

// Этот код меняет поведение прокрутки на smooth
window.scrollTo({
  top: 1000,
  behavior: 'smooth',
})
```

Методы **scroll** и **scrollTo** абсолютно идентичны — мы можем использовать любой из них. Отличие **scrollBy** в том, что он использует относительные координаты, в то время как **scroll** и **scrollTo** используют абсолютные. Последовательно вызывая **scrollBy** с одними и теми же параметрами, мы будем менять положение полосы прокрутки на значение, переданное в параметрах. В случае **scroll** и **scrollTo** изменения произойдут лишь при первом вызове.

Чтобы переместить полосу прокрутки в контексте DOM-элемента, нужно присвоить значения полям **Element.scrollTop** и **Element.scrollLeft**.

Метод **Element.scrollToView**

Метод **Element.scrollToView()** интерфейса **Element** прокручивает текущий контейнер родителя элемента так, чтобы элемент, на котором вызван **scrollIntoView()**, был видим пользователю. Этот метод принимает два типа параметров:

1. **alignToTop** — необязательный аргумент типа Boolean со следующими возможными значениями:
 - a. **true** — верхняя граница элемента выравнивается по верхней границе видимой части окна прокручиваемой области. Соответствует **scrollIntoViewOptions**: `{block: "start", inline: "nearest"}`. Значение по умолчанию.
 - b. **false** — нижняя граница элемента выравнивается по нижней границе видимой части окна прокручиваемой области. Соответствует конфигурации **scrollIntoViewOptions**: `{block: "end", inline: "nearest"}`
2. **scrollIntoViewOptions** — необязательный аргумент типа Object со следующим набором полей:

- a. **behavior** — определяет анимацию скролла. Необязательный параметр. Принимает значение **auto** или **smooth**. По умолчанию — **auto**.
- b. **block** — вертикальное выравнивание. Необязательный параметр. Варианты значений: **start**, **center**, **end** или **nearest**. По умолчанию — **start**.
- c. **inline** — горизонтальное выравнивание. Необязательный параметр. Варианты значений: **start**, **center**, **end** или **nearest**. По умолчанию — **nearest**.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .big {
        background: #ccc;
        height: 300px;
      }
      .box {
        background: lightgreen;
        height: 40px;
      }
    </style>
    <title>Пример - Element.scrollIntoView()</title>
  </head>
  <body>
    <button type="button">Нажми на меня</button>
    <div class="big"></div>
    <div id="box" class="box">Скрытый элемент</div>
    <script>
const hiddenElement = document.getElementById('box')
const button = document.querySelector('button')

const handleButtonClick = () => {
  hiddenElement.scrollIntoView({block: 'center', behavior: 'smooth'})
}
button.addEventListener('click', handleButtonClick)
    </script>
  </body>
</html>
```

Использование Drag and Drop API

API перетаскивания HTML5 DnD API позволяет сделать практически любой элемент на нашей странице перетаскиваемым. В большинстве браузеров выделенный текст, изображения и ссылки по умолчанию перетаскиваются.

Например, если мы начнём перетаскивать логотип Google в [поиске Google](#), то увидим полупрозрачное изображение. Затем это изображение перетаскивается в адресную строку, на

элемент `<input type="file" />` или даже на рабочий стол. Чтобы сделать другие типы контента перетаскиваемыми, нам понадобятся API-интерфейсы HTML5 DnD.

Для превращения элемента в перетаскиваемый ему устанавливают атрибут **draggable=true**. Практически всё может быть со включённым перетаскиванием: изображения, файлы, ссылки и любая разметка на странице.

В нашем примере мы создадим интерфейс для перестановки некоторых столбцов, размещённых посредством CSS Grid. Базовая разметка для столбцов выглядит так: для каждого столбца установлен **draggable**-атрибут **true**.

```
<div class="container">
  <div draggable="true" class="box">A</div>
  <div draggable="true" class="box">B</div>
  <div draggable="true" class="box">C</div>
</div>
```

Это CSS для элементов контейнера и боксов. Обратите внимание, что единственное CSS-правило, связанное с функциональностью DnD, это свойство **cursor: move**. Остальное просто управляет компоновкой и стилями контейнера и боксов.

```
.container {
  display: grid;
  grid-template-columns: repeat(5, 1fr);
  gap: 10px;
}

.box {
  border: 3px solid #666;
  background-color: #ddd;
  border-radius: .5em;
  padding: 10px;
  cursor: move;
}
```

На этом этапе мы можем перетаскивать элементы, но больше ничего не происходит. Чтобы добавить функциональность DnD, нам надо использовать JavaScript API.

Прослушивание событий перетаскивания

Есть ряд событий, к которым можно привязать мониторинг всего процесса перетаскивания.

Когда срабатывают эти события:

1. **dragstart** — когда пользователь начал перетаскивать элемент.

2. **drag** — каждые несколько сотен миллисекунд, пока длится перетаскивание элемента или выделение текста.
3. **dragenter** — когда перетаскиваемый элемент попадает в допустимую цель сброса.
4. **dragleave** — когда перетаскиваемый элемент покидает допустимую цель сброса.
5. **dragover** — когда элемент перетаскивается над допустимой целью сброса каждые несколько сотен миллисекунд.
6. **drop** — когда элемент сбрасывается в допустимую зону сброса.
7. **dragend** — в момент завершения перетаскивания, например, при отпускании кнопки мыши или при нажатии Escape.

Чтобы корректно обрабатывать DnD-процесс, нам потребуется:

- исходный элемент (source element) — откуда начинается перетаскивание;
- объект с данными (data payload) — структура, которую мы перетаскиваем;
- целевой элемент (target) — область, в которой мы «ловим» отпускаемый объект.

Исходный элемент может быть изображением, списком, ссылкой, файловым объектом, блоком HTML и т. д. Целевой элемент — зона перетаскивания (drop zone) или набор таких зон для приёма данных, которые пользователь пытается «бросить». Не все элементы могут быть целями: например, изображение не может.

Обработка начала и завершения перетаскивания

После того как мы указали для нашего контента атрибуты **draggable="true"**, надо подписаться на событие **dragstart**, чтобы запустить последовательность DnD для каждого столбца.

Этот код установит прозрачность столбца на 40%, когда пользователь начнёт его перетаскивать, а затем вернёт её на 100% по окончании события перетаскивания:

```
const items = document.querySelectorAll('.container .box')

const handleDragStart = ({target}) => {
  target.style.opacity = '0.4'
}

const handleDragEnd = ({target}) => {
  target.style.opacity = '1'
}

items.forEach((item) => {
```

```
item.addEventListener('dragstart', handleDragStart)
item.addEventListener('dragend', handleDragEnd)
})
```

Чтобы помочь пользователю понять, как взаимодействовать с интерфейсом, используют дополнительные стили, которые мы установим в обработчиках событий **dragenter**, **dragover** и **dragleave**. В примере ниже столбцы не только перетаскиваются, но и служат целями для перетаскивания. Мы можем помочь пользователю понять это, сделав границу пунктирной, когда он удерживает перетаскиваемый элемент над столбцом. Например, добавим в наш CSS класс **over** для представления элементов, которые считаются целями перетаскивания:

```
.box.over {
  border: 3px dotted #666;
}
```

Затем в нашем JavaScript настроим обработчики событий:

- для добавления класса **over**, когда столбец перетаскивается над областью, занимаемой элементом;
- для удаления класса при покидании области.

В обработчике **dragend** мы также убираем классы в конце перетаскивания:

```
const items = document.querySelectorAll('.container .box')

const handleDragStart = ({target}) => {
  target.style.opacity = '0.4'
}

const handleDragEnd = ({target}) => {
  target.style.opacity = '1'

  items.forEach((item) => {
    item.classList.remove('over')
  })
}

const handleDragOver = (event) => {
  if (event.cancelable) {
    event.preventDefault()
  }

  return false
}

const handleDragEnter = ({target}) => {
  target.classList.add('over')
}
```

```
const handleDragLeave = ({target}) => {
  target.classList.remove('over')
}

items.forEach((item) => {
  item.addEventListener('dragstart', handleDragStart)
  item.addEventListener('dragover', handleDragOver)
  item.addEventListener('dragenter', handleDragEnter)
  item.addEventListener('dragleave', handleDragLeave)
  item.addEventListener('dragend', handleDragEnd)
})
```

В коде стоит обратить внимание на несколько моментов:

1. При перетаскивании, например, ссылки нам надо предотвратить поведение браузера по умолчанию, которое заключается в переходе по этой ссылке. Для этого вызовем `event.preventDefault()` в обработчике **dragover**.
2. Обработчик события **dragenter** используется вместо **dragover** для переключения класса **over**. Если использовать **dragover**, CSS-класс будет переключаться много раз, поскольку событие **dragover** продолжает срабатывать при перемещении курсора внутри области столбца. А если по какой-то причине потребуется использовать именно событие **dragover**, стоит подумать об ограничении количества срабатываний посредством паттерна `throttling` или об отключении слушателя события после первого срабатывания.

Обработка завершения перетаскивания

Чтобы обработать момент, когда пользователь отпускает перетаскиваемый объект над целевым элементом, надо подписаться на событие **drop**. В обработчике события **drop** нам потребуется предотвратить поведение браузера по умолчанию, которое обычно представляет собой перенаправление.

Предотвратим это поведение, вызвав `event.stopPropagation()`.

```
...

const handleDrop = (event) => {
  event.stopPropagation()

  return false
}

items.forEach((item) => {
  item.addEventListener('dragstart', handleDragStart)
```

```
item.addEventListener('dragover', handleDragOver)
item.addEventListener('dragenter', handleDragEnter)
item.addEventListener('dragleave', handleDragLeave)
item.addEventListener('dragend', handleDragEnd)
item.addEventListener('drop', handleDrop)
})
```

Если запустить код на этом этапе, элемент не займёт новое место. Для этого нам надо использовать объект **DataTransfer**.

Именно здесь, в **dataTransfer**, происходит вся магия DnD. Он содержит фрагмент данных, отправленный при перетаскивании. Объект **dataTransfer** устанавливается в событии **dragstart** и читается (обрабатывается) в других событиях перетаскивания. Вызов `e.dataTransfer.setData(mimeType, dataPayload)` позволяет установить MIME-тип объекта и добавить необходимые данные.

В этом примере мы позволим пользователям изменять порядок столбцов. Для этого сначала сохраним HTML-код исходного элемента при старте перетаскивания:

```
const handleDragStart = ({target}) => {
  target.style.opacity = '0.4'

  e.dataTransfer.effectAllowed = 'move'
  e.dataTransfer.setData('text/html', target.innerHTML)
}
```

В обработчике **drop**, где мы обрабатываем отпускание столбца, заменим HTML-код целевого столбца на код исходного столбца, перетаскиваемый пользователем. Но перед этим проверим, не происходит ли отпускание на тот же столбец, из которого оно началось.

```
const handleDrop = (event) => {
  event.stopPropagation()
  const sourceElementHtml = e.dataTransfer.getData('text/html')

  if (event.target.innerHTML === sourceElementHtml) {
    return
  }
  event.target.innerHTML = sourceElementHtml

  return false
}
```

Дополнительные свойства объекта перетаскивания

Объект **dataTransfer** предоставляет свойства для обеспечения визуальной обратной связи с пользователем в процессе перетаскивания. Эти свойства также используются для управления тем, как каждая цель перетаскивания реагирует на определённый тип данных.

Свойство **dataTransfer.effectAllowed** устанавливает, какой «тип перетаскивания» пользователь может выполнять с элементом. Он используется в модели обработки перетаскивания и падения для инициализации **dropEffect** во время событий **dragenter** и **dragover**. Свойству могут быть присвоены значения **none**, **copy**, **copyLink**, **copyMove**, **link**, **linkMove**, **move**, **all** и **uninitialized**.

Свойство **dataTransfer.dropEffect** контролирует данные о том, что пользователь видит во время событий **dragenter** и **dragover**. Когда пользователь наводит курсор на целевой элемент, курсор браузера указывает, какой тип операции будет выполняться: копирование, перемещение или ещё что-то. «Эффект» принимает одно из следующих значений: **none**, **copy**, **link**, **move**.

Метод **dataTransfer.setDragImage(imgElement, x, y)** применяют для настройки обратной связи браузера по умолчанию в виде полупрозрачного изображения. Мы можем дополнительно установить значок перетаскивания.

Заключение

Мы рассмотрели функциональность, предоставляемую Drag and Drop API. Помимо перетаскивания элементов на странице, он позволяет также переносить файлы с рабочего стола пользователя прямо в веб-приложение и получать к доступ к этим файлам для их дальнейшей обработки или отправки на сервер. Подробнее рассмотрим эти возможности на следующих уроках.