JSDG 🦏 | JavaScript in Web Browsers | JavaScript в веб-браузерах | chapter 15

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

Язык JavaScript был создан в 1994 году со специальной целью — сделать возможным динамическое поведение в документах, отображаемых веб-браузерами. С тех пор язык значительно эволюционировал и одновременно произошел бурный рост масштабов и возможностей веб-платформы. В наши дни программисты на JavaScript могут считать веб-сеть полнофункциональной платформой для разработки приложений.

---

async и defer

Применение метода document .write () больше не считается хорошим стилем. Однако сам факт такой возможности означает, что когда синтаксический анализатор HTML встречает элемент <script>, он обязан по умолчанию за пустить сценарий, просто чтобы удостовериться в отсутствии вывода любой HTML-разметки, прежде чем можно будет возобновить анализ и визуализацию документа. В итоге анализ и визуализация веб-страницы могут значительно за медлиться.

К счастью, такой стандартный *синхронный* или *блокирующий* режим вы полнения сценария — не единственный вариант. Дескриптор <script> может иметь атрибуты defer и async, которые заставляют сценарии выполняться по-разному. Они являются булевскими атрибутами, т.е. не имеют значения; им нужно лишь присутствовать в дескрипторе <script>.

‹script defer src="deferred.js"></script>
‹script async src="async.js"></script>

Оба атрибута, defer и async, представляют собой способы сообщения браузеру о том, что в связанном сценарии не применяется метод document. write () для генерации вывода

HTML, вследствие чего браузер может продолжить синтаксический анализ и визуализацию документа во время загрузки сценария.

Атрибут defer заставляет браузер отложить выполнение сценария до тех пор пока документ полностью не загрузится, будет проанализирован и станет готовым к манипуляциям.

Атрибут async заставляет браузер запустить сценарий как можно раньше, но не блокировать анализ документа во время загрузки сценария.

Когда дескриптор <script> содержит оба атрибута, приоритет имеет async.

___

Загрузка сценариев по запросу

Если вы разрабатываете код с применением модулей, то можете загружать модуль по запросу с помощью import (), как было описано в подразделе 10.3.6.

```
// Асинхронно загружает и втолняет сценарий из указанного URL.
// Возвращает объект Promise, который разрешается, когда сценарий загружен
function importScript (url) (
      return new Promise ( (resolve, reject) => {
            let s = document. createElement ("script"); // Создать элемент <script>.
            s.onload = () => | resolve (); }; // Разрешить объект Promise, когда сценарий загружен.
            s.onerror = (e) => { reject (e); }; // Отклонить объект Promise в случае неудачи.
            s.src = url; // Установить URL сценария.
            document. head. append (s); / / Добавить <script> в документ
      });
)
```

___

Веб-воркер — это фоновый поток для выполнения задач с интенсивными вычислениями без замораживания пользовательского интерфейса. Код, который выполняется в потоке веб-воркера, не имеет доступа к со держимому документа, не разделяет никакого состояния с главным потоком или другими веб-воркерами и может взаимодействовать с главным потоком и други ми веб-воркерами только посредством событий асинхронных сообщений. Таким образом, параллелизм не поддается обнаружению главным потоком, а веб-ворке- ры не меняют базовую модель однопоточного выполнения программ JavaScript.

Когда неперехваченное исключение распространится вверх по стеку вызовов до самого конца и сообщение об ошибке готово отобразиться в Консоли инструментов разработчика, функция **window.onerror** будет вызвана с тремя строковыми аргументами. В первом аргументе window.onerror передайся сообщение, описывающее ошибку.

Межсайтовые сценарии

Межсайтовые сценарии (cross-site scripting - XSS) - это термин для обозначения категории проблем, связанных с безопасностью, когда атакующий внедряет HTML-дескрипторы в целевой веб-сайт. Программисты на JavaScript стороны клиента обязаны знать о межсайтовых сценариях и защищаться от них.

События

Категории событий

Зависимые от устройства события ввода

Эти события напрямую привязаны к конкретному устройству ввода, та кому как мышь или клавиатура. Сюда входят типы событий наподобие "mousedown”, "mousemove", "mouseup", "touchstart", "touchmove", "touchend", "keydown" и "keyup".

Независимые от устройства события ввода

- click

- Событие "input" является независимой от устройства альтернативой событию "keydown" и поддерживает клавиатурный ввод, а также варианты вроде вырезания и вставки и методы ввода, применяемые при идеографическом письме.

- Типы событий "pointerdown", "pointermove" и "pointerup" представляют собой независимые от устройства альтернативы событиям мыши и касания.

События пользовательского интерфейса

- focus
- change
- submit

События изменения состояния

- load
- DOMContentLoaded
- Браузеры инициируют события "online”и "offline”в объекте Window при изменении подключаемости к сети.
- Механизм управления хронологией браузера (см. подраздел 15.10.4) инициирует событие "popstate" как ответ на щелчок на кнопке перехода на предыдущую страницу.

События, специфичные для API-интерфейса

- <video> и <audio>

определяют длинный список ассоциированных типов событий, таких как "waiting", "playing", "seeking", "volumechange" и т.д., которые вы можете применять для настрой ки воспроизведения мультимедийного содержимого.

- Скажем, API-интерфейс IndexedDB (см. подраздел 15.12.3) инициирует со бытия "success” и "error", когда запросы к базе данных завершаются успешно или терпят неудачу.
- И хотя новый API-интерфейс fetch () (см. подраздел 15.11.1) для выполнения HTTP-запросов основан на Promise, в API-интерфейсе XMLHttpRequest, который он заменил, определено не сколько типов событий, специфичных для API-интерфейса.

___

Регистрация обработчиков событий

- Первый способ, Доступный с ранней поры существования веб-сети, предусматривает установку свойства объекта или элемента документа, который является целью события.
- Второй способ (более новый и универсальный) заключается в передаче обра ботчика методу addEventListener () объекта или элемента.

Регистрация захватывающего обработчика событий - лишь один из трех вариантов, поддерживаемых методом addEventListener () , и вместо передачи одиночного булевского значения вы также можете передать объект, который
явно указывает желаемые варианты:

```
document. addEventListener ("click", handleClick,
      capture: true,
      once: true,
      passive: true
};
```

Если объект Options имеет свойство capture, установленное в true, тогДа обработчик событий будет зарегистрирован как захватывающий обработчик*
£если свойство capture, установлено в false или опущено, тогда обработчик ле будет захватывающим.

Если объект Options имеет свойство once, установленное в true, тогда про- сдущиватель событий будет автоматически удален после однократного запуска. Если свойство once установлено в false или опущено, тогда обработчик ни когда не удалится автоматически.

Если вы передаете true, то функция обработчика регистрируется как *захватывающий* обработчик событий и вы зывается на другой стадии передачи события.

Если объект Options имеет свойство passive, установленное в true, то он указывает, что обработчик событий никогда не будет вызывать метод preventDefa u lt () для отмены стандартного действия

___

Вызов обработчиков событий

Аргумент обработчика событий

• type. Тип события, которое произошло.
• target. Объект, в котором произошло событие.
• currentTarget. Для распространяемых событий это свойство представляет собой объект, в котором был зарегистрирован текущий обработчик событий.
• timeStamp. Отметка времени (в миллисекундах), которая представляет абсолютное время. Вы можете определить время, прошедшее между двумя событиями, путем вычитания отметки времени первого события из отметки времени второго события.
• isTrusted Это свойство будет равно true, если событие было отправлено самим веб-браузером, и false, если кодом JavaScript.

___

Распространение событий

После того, как обработчики событий, зарегистрированные для элемента цели, вызваны, большинство событий поднимаются

подобно пузырькам вверх по дереву DOM. Вызываются обработчики событий родителя цели. Затем вызываются обработчики, зарегистрированные для прародителя цели. Процесс продолжается вплоть до объекта Document и далее до объекта Window. Пузырьковый подъем предоставляет альтернативу регистрации обработчиков для множества индивидуальных элементов документа: взамен вы можете зарегистрировать единственный обработчик для элемента общего предка и обрабатывать в нем события. Например, вы можете зарегистрировать обработчик событий "change" для элемента <form> вместо того, чтобы регистрировать по одному обработчику событий "change" для каждого элемента формы.

Пузырьковый подъем характерен для большинства событий, возникающих в элементах документа. Заметные исключения - события "focus", "blur" и "scroll". Событие "load", происходящее в элементах документа, поднимается подобно пузырьку, но перестает всплывать в объекте Document и не распространяется до объекта Window. (Обработчик событий "load" объекта Window запускается, только когда документ полностью загружен.)

Пузырьковый подъем представляет собой вторую "стадию" распространения событий. Вызов обработчиков событий на самом объекте цели является второй стадией. Первая стадия, которая происходит еще до вызова обработчиков событий, называется стадией "захвата". Вспомните, что метод addEventListener ( ) принимает необязательный третий аргумент. Если в этом аргументе передается true или (capture:true), тогда обработчик регистрируется как захватывающий обработчик событий для вызова во время первой стадии распространения событий. Стадия захвата распространения событий похожа на стадию пузырькового подъема наоборот. Сначала вызываются захватывающие обработчики объекта Window, затем захватывающие обработчики объекта Document, затем объекта тела и так далее вниз по дереву DOM до тех пор, пока не будут вызваны захватывающие обработчики событий в родительском объекте цели события. Захватывающие обработчики событий, зарегистрированные в самой цели события, не вызываются.

Отмена событий

preventDefault () объекта события. (За исключением случая регистрации обработчика с объектом Options, имеющим установленное свойство passive, что делает вызов preventDefault () безрезультатным.)

Мы можем также отменять распространение со бытий, вызывая метод stopPropagation () объекта события. Если для того же самого объекта определены другие обработчики, то оставшиеся обработчики по- прежнему будут вызваны, но после вызова stopPropagation () никакие про чие обработчики событий для любого другого объекта не вызываются. Метод stopPropagation () работает во время стадии захвата, в самой цели события и в течение стадии пузырькового подъема. Метод stopImmediatePropagation() работает подобно stopPropagation (), но также предотвращает вызов любых последующих обработчиков событий, зарегистрированных для того же объекта.

___

Отправка специальных событий

API-интерфейс для работы с событиями в коде JavaScript стороны клиента об ладает относительно большой мощностью и позволяет вам определять и отправ лять собственные события.

Если объект JavaScript имеет метод **addEventListener** (), тогда он будет “целью события”, а значит иметь также и метод **dispatchEvent** (). Вы можете создать собственный объект события с помощью конструктора **CustomEvent** () и передать его методу **dispatchEvent** ().

// Отправить специальное событие, чтобы уведомить // пользовательский интерфейс о том, что мы заняты. document.

dispatchEvent (new CustomEvent ("busy", { detail: true }));

// После того, как сетевой запрос пройдет успешно или потерпит
// неудачу, отправить еще одно событие, чтобы уведомить
// пользовательский интерфейс о том, что мы больше не заняты.

document. dispatchEvent (new CustomEvent ("busy", { detail:
false }));

// Где-то в другом месте программы можно зарегистрировать обработчик И для событий "busy" и использовать его для показа или сокрытия
// вращателя, уведомляющего пользователя о занятости.

document. addEventListener ("busy", (e) => { if (e.detail)
{ showSpinner (); } else { hideSpinner ();]);

—

Выбор элементов документа

- querySelector () и querySelectorAll () в DOM позволяют находить элемент или элементы внутри документа, которые соответствуют указанному селектору CSS.

Объект NodeList, возвращенный методом querySelectorA ll (), будет иметь свойство length, установленное в 0, если в документе отсутствуют элементы, которые соответствовали бы указанному селектору.

Методы querySelector () и querySelectorAll () реализованы в классах Element и Document. В случае вызова на элементе они возвращают только эле менты, являющиеся потомками данного элемента.

- Есть еще один метод выбора элементов на основе CSS — c l o s e s t (). Он опре делен в классе Element и принимает в своем единственном аргументе селектор, елиселектор соответствует элементу, на котором вызван метод closest (),

// Найти ближайший объемлющий дескриптор <a>, который имеет атрибут href

let hyperlink = event.target.closest ("a (href]");

Вот другой возможный способ использования closest ():

```
// Возвратить true, если элемент e находится внутри
// спискового HTML-элемента.

function insideList (e) {
      return e.closest ("ul,ol,dl") !== null;
}
```

- Связанный метод matches () не возвращает предков или потомков: он просто проверяет, соответствует ли элемент селектору CSS, и в случае соответствия возвращает true, а иначе false:

```
// Возвратить true, если e - заголовочный HTML-элемент.
function isHeading (e) {
      return e.matches ("h1, h2, h3, h4, h5, h6");
}
```

Другие методы выбора элементов

```
/ Искать элемент по идентификатору.
// Аргументом является только идентификатор без префикса #
// селектора CSS. Подобен document.querySelector ("#sect1").
let sect1 = document. getElementById ("sect1");

// Искать все элементы (такие как флажки в форме),
// которые имеют атрибут name="color".
// Подобен document. querySelectorA11 (** [name="color"]').
let colors = document. getElementsByName ("color");

// Искать все элементы <h1> в документе.
// Подобен document. querySelectorAll ("h1").
let headings = document. getElementsByTagName ("h1");

// Метод getElementsByTagName ( ) также определен в элементах.
// Получить все элементы <h2> внутри элемента sect1.
let subheads = sect1.getElementsByTagName ("h2");
```

// Искать все элементы, которые имеют класс "tooltip" •
// Подобен document. querySelectorAll (" tooltip").
let tooltips = document.getElementsByClassName ("tooltip");

// Искать всех потомков элемента sectl, которые имеют класс "sidebar"
// Подобен sect1. querySelectorAll (" sidebar")
let sidebars = sect1.getElementsByClassName ("sidebar");

___

Структура и обход документа

parentNode
children
childElementCount
firstElementChild, lastElementChild.
nextElementSibling, previousElementSibling.

document.children [0].children[1]
document.firstElementChild.firstElementChild.nextElementSibling

Документы как деревья узлов
parentNode
childNodes
firstChild, lastChild.
nextSibling, previousSibling.
nodeType
nodeValue
nodeName

Атрибуты

В классе Element определены универсальные методы getAttribute(), setAttribute (), hasAttribute () n removeAttribute () для запрашивания, установки, проверки и удаления атрибутов элемента. Но значения атрибутов HTML-элементов (для всех стандартных атрибутов стандартных HTML-элементов) доступны в виде свойств объектов HTMLElement, которые представляют эти элементы, и обычно намного легче работать с ними как со свойствами JavaScript, чем вызывать getAttribute () и связанные

методы.

Атрибуты набора данных

Пусть у вас есть HTML-документ, содержащий такой текст:

<h2 id="title" data-section-number="16.1›Attributes</h2>

Тогда для доступа к номеру раздела вы могли бы написать следующий код JavaScript:

let number = document.querySelector
("#title") .dataset.sectionNumber;

——

Веб-компоненты **Теневая модель DOM**

Чтобы превратить специальный элемент в подлинный веб-компонент, должен использоваться мощный механизм инкапсуляции, известный как ***теневая модель DOM*** (shadowDOM).

Теневая модель DOM позволяет прикреплять “корневой элемент теневого дерева” (“shadow root”) к специальному элементу (и также к элементам , **,** <body>, <article>, <main>, <nav>, <header>, <footer>, <section>, <p>, <blockquote>, <aside> или <hl> — <h6>), который становится “веду щим элементом теневого дерева” (“shadow host”). Ведущие элементы теневого дерева, как и все HTML-элементы, уже являются корнем нормального дерева DOM из элементов потомков и текстовых узлов. Корневой элемент теневого дерева — это корень другого, более закрытого дерева из элементов потомков, которое произрастает из ведущего элемента теневого дерева и может считаться отдельным мини-документом.
Слово “теневая” в формулировке “теневая модель DOM” относится к тому факту, что элементы, которые происходят от корневого элемента теневого де рева, “прячутся в тени”: они не являются частью нормального дерева DOM, не присутствуют в массиве children своего ведущего элемента и не посещают ся методами обхода нормального дерева DOM, такими как

querySelector (). Ради контраста дочерние элементы нормального дерева DOM ведущего элемен та теневого дерева иногда называют “световой моделью DOM”.

——

SVG: масштабируемая векторная графика

Масштабируемая векторная графика (scalable vector graphics — SVG) явля ется форматом изображений. Слово “векторная” в названии служит признаком на то, что он фундаментально отличается от форматов растровых изображений таких как GIF, JPEG и PNG, которые указывают матрицу значений пикселей*

Графикав<canvas>

Элемент <canvas> не обладает собственным внешним видом, но создает вНУ^_ ри документа поверхность рисования и предоставляет мощный API-интерФ^ рисования для JavaScript стороны клиента. Главное отличие АР1-интерФеИ <canvas> и SVG связано с тем, что в случае холста рисунки создаются за счет розова методов, а в случае SVG — путем построения дерева элементов XML. Эти два подхода одинаково эффективны: один может эмулироваться посредс твом другого. Тем не менее, на первый взгляд они выглядят совершенно раз ными, и у каждого есть свои сильные и слабые стороны. Скажем, рисунок SVG легко редактировать, удаляя элементы из его описания. Чтобы удалить элемент и3 той же самой графики в <canvas>, часто необходимо очистить рисунок и нарисовать его с нуля. Поскольку API-интерфейс рисования Canvas основан на JavaScript и относительно компактен (в отличие от грамматики SVG), он доку ментируется в книге более подробно.

Трехмерная графика в Canvas

Вытакже можете вызвать getContext () со строкой "webgl", чтобы по лучить объект контекста, который даст возможность рисовать трехмерную графику с применением API-интерфейса WebGL. Крупный, сложный и низ коуровневый API-интерфейс WebGL позволяет программистам на JavaScript обращаться к графическому процессору, писать специальные шейдеры (т.е. затеняющие программы) и выполнять другие очень мощные

графические операции. Однако WebGL в этой книге не рассматривается: разработчики веб-приложений больше предпочитают использовать служебные библи отеки, построенные поверх WebGL, нежели работать с API-интерфейсом WebGL напрямую.

API-интерфейс WebAudio
Местоположение, навигация и хронология

Свойство **location** объектов **Window** и **Document** ссылается на объект **Location,** который представляет URL текущего документа, отображаемого в окне, и также предлагает API-интерфейс для загрузки новых документов в окно.

Хронология просмотр

Свойство history объекта Window ссылается на объект History для окна, который моделирует хронологию просмотра окна в виде списка документов и состояний документов.

Объект History имеет методы back () и forward (), поведение которых по хоже на действие кнопок перехода на предыдущую и на следующую страницу браузера

Третий метод, до (), принимает целочисленный аргумент и может пропускать любое количество страниц вперед (для положительного зна чения аргумента) или назад (для отрицательного значения аргумента) в списке хронологии:

Взаимодействие с сетью

- Метод fetch () определяет API-интерфейс, основанный на Promise, для создания HTTP- и HTTPS-запросов. API-интерфейс fetch () делает базовые запросы GET простыми, но обладает всесторонним набором средств, который также поддерживает практически любой возможный сценарий использования HTTP.

API-интерфейс **f e t c h** () замещает собой причудливый и обманчиво назван ный API-интерфейс **XMLHttpRequest** (который не имеет ничего общего с XML). Вы все еще можете сталкиваться с **XMLHttpRequest** (или его аббре виатурой XHR) в существующем коде, но нет никаких причин использовать его в новом коде и потому в настоящей главе он не документируется. Тем не менее, один пример применения **XMLHttpRequest** в этой книге есть, и вы можете обратиться в подраздел 13.1.3, если хотите увидеть пример взаимо действия с сетью посредством JavaScript в старом стиле.

- API-интерфейс SSE (Server-Sent Events — события, посылаемые сервером) представляет собой удобный, основанный на событиях интерфейс для методик “длинного опроса” HTTP, при которых веб-сервер удерживает сете вое подключение открытым, так что он может посылать данные клиенту всякий раз, когда пожелает.

- Веб-сокеты — это сетевой протокол, который не является HTTP, но пред назначен для взаимодействия с HTTP. Он определяет асинхронный API- интерфейс передачи сообщений, где клиенты и серверы могут посылать и принимать сообщения друг от друга способом, похожим на сетевые соке ты TCP.

События, посылаемые сервером

Фундаментальная особенность протокола HTTP, с учетом которой построена веб-сеть, связана с тем, что клиенты инициируют запросы, а серверы отвечают на эти запросы. Тем не менее, некоторые веб-приложения считают полезным чтобы их сервер посылал уведомления, когда происходят события. Это не является естественным для HTTP, но разработанная методика заключается в том что клиент делает запрос к серверу, и затем ни клиент, ни сервер не закрывают подключение.

Когда серверу есть о чем сообщить клиенту, он записывает дан

ные в подключение, но оставляет его открытым. Результат оказывается таким как будто клиент делает сетевой запрос, а сервер отвечает медленным и пре рывистым образом со значительными паузами между всплесками активности. Сетевые подключения подобного рода обычно не остаются открытыми навсег да, но если клиент обнаруживает, что подключение закрыто, он может просто сделать еще один запрос, чтобы повторно открыть подключение.

Поскольку данная методика представляет собой полезный программный шаблон, в JavaScript стороны клиента она реализована с помощью API-интерфейса EventSource. Чтобы создать такое долговременное подключение к веб-серверу, нужно просто передать URL конструктору EventSource (). Когда сервер записывает (надлежащим образом сформатированные) данные в под ключение, объект EventSource транслирует их в события, которые вы можете прослушивать:

```
let ticker = new EventSource("stockprices.php");
ticker.addEventListener("bid", (event) => {
      displayNewBid(event.data);
}
```

Протокол SSE прямолинеен. Клиент инициирует подключение к серверу (со здавая объект EventSource) и сервер сохраняет это подключение открытым.

Реализация чат-сервера с помощью SSE

Пример 15.12. Реализация чат-сервера с помощью SSE

```
// Это код JavaScript стороны сервера, рассчитанный на запуск с помощью NodeJS.
// Он реализует очень простую, полностью анонимную дискуссионную группу.
// Посредством запросов POST для /chat он отправляет новые сообщения или
// с помощью запросов GET к тому же самому URL получает сообщения в формате
// "text/event-stream".
// Запрос GET к / возвращает простой HTML-файл, который содержит
// пользовательский интерфейс стороны клиента для чата.
const http = require("http");
const fs = require("fs");
const url = require("url");

// HTML-файл для чат-клиента. Используется ниже.
const clientHTML = fs.readFileSync("chatClient.html");

// Массив объектов ServerResponse, которым мы собираемся посылать события.
let clients = [];

// Создать новый сервер и прослушивать порт 8080.
// Для его использования подключитесь к http://localhost:8080/.
let server = new http.Server();
server.listen(8080);

// Когда сервер получает новый запрос, выполнить эту функцию.
server.on("request", (request, response) => {
   // Разобрать запрошенный URL.
   let pathname = url.parse(request.url).pathname;

   // Если запрос был для "/", тогда отправить пользовательский
   // интерфейс клиентской стороны для чата.
   if (pathname === "/") { // A request for the chat UI
     response.writeHead(200, {"Content-Type": "text/html"}).end(clientHTML)
   }
   // В противном случае отправить ошибку 404 для любого пути кроме "/chat"
   // или для любого метода, отличающегося от "GET" и "POST".
   else if (pathname !== "/chat" ||
          (request.method !== "GET" && request.method !== "POST")) {
      response.writeHead(404).end();
   }
   // Если запросом для /chat был GET, тогда клиент подключается.
   else if (request.method === "GET") {
      acceptNewClient(request, response);
   }
```

```
    // Иначе запросом к /chat является POST для нового сообщения.
    else {
        broadcastNewMessage(request, response);
    }
});
// Эта функция обрабатывает запросы GET для конечной точки /chat,
// которая генерируется, когда клиент создает новый объект EventSource
// (или когда EventSource автоматически повторно подключается).
function acceptNewClient(request, response) {
    // Запомнить объект ответа, чтобы ему можно было посылать
    // будущие сообщения.
    clients.push(response);

    // Если клиент закрыл подключение, тогда удалить соответствующий
    // объект ответа из массива активных клиентов.
    request.connection.on("end", () => {
        clients.splice(clients.indexOf(response), 1);
        response.end();
    });

    // Установить заголовки и отправить начальное событие чата
    // для этого одного клиента.
    response.writeHead(200, {
        "Content-Type": "text/event-stream",
        "Connection": "keep-alive",
        "Cache-Control": "no-cache"
    });
    response.write("event: chat\ndata: Connected\n\n");

    // Обратите внимание, что мы намеренно не вызываем здесь response.end().
    // Именно сохранение подключения открытым обеспечивает работу SSE.
}

// Эта функция вызывается в ответ на запросы POST к конечной точке /chat,
// которые клиенты посылают, когда пользователи вводят новые сообщения.
async function broadcastNewMessage(request, response) {
    // Прочитать тело запроса, чтобы получить сообщение пользователя.
    request.setEncoding("utf8");
    let body = "";
    for await (let chunk of request) {
        body += chunk;
    }

    // После того, как тело прочитано, отправить пустой ответ
    // и закрыть подключение.
    response.writeHead(200).end();

    // Сформатировать сообщение в формате text/event-stream, снабжая
    // каждую строку префиксом "data: ".
    let message = "data: " + body.replace("\n", "\ndata: ");

    // Предоставить данным сообщения префикс, который определяет
    // их как событие "chat", и снабдить их суффиксом в виде двух
    // символов новой строки, которые помечают конец события.
    let event = `event: chat\n${message}\n\n`;

    // Теперь отправить это событие всем прослушивающим клиентам.
    clients.forEach(client => client.write(event));
}
```

Веб-сокеты

API-интерфейс WebSocket представляет собой простой интерфейс к сложному и мощному сетевому протоколу.

Веб-сокеты позволяют коду JavaScript в браузере легко обмениваться текстовыми и двоичными сообщениями с

сервером.

Как в случае с событиями, посылаемыми сервером (SSE), клиент обязан установить подключение, но после того, как подключение установлено, сервер может асинхронно посылать сообщения клиенту.

В отличие от SSE поддерживаются двоичные сообщения, и сообщения могут отправляться в обоих направлениях, а не только от сервера до клиента.

—

Хранилище

Веб-хранилище

API-интерфейс веб-хранилища (Web Storage) состоит из объектов localStorage и sessionStorage, которые по существу являются постоянными объектами, отображающими строковые ключи на строковые значения. Веб-хранилище очень легко использовать и оно подходит для хранения крупных (но не гигантских) объемов данных.

Cookie-наборы

Cookie-наборы - это старый механизм хранения на стороне клиента, который был спроектирован для применения сценариями стороны сервера. Неуклюжий API-интерфейс JavaScript позволяет работать с cokie-наборами на стороне клиента, но они трудны в использовании и подходят для хранения только небольших объемов текстовых данных. Кроме того, любые данные, сохраненные как cokie-наборы, всегда передаются серверу с каждым HTTP-запросом, даже если данные интересуют только клиент.

IndexedDB

*IndexedDB представляет собой асинхронный API-интерфейс к объектной
базе данных, которая поддерживает индексацию.*

Потоки воркеров и обмен сообщениями

—

Резюме и рекомендации относительно дальнейшего чтения

Производительность

Главной точкой входа в данном API. интерфейсе является свойство performance объекта окна. Он включает источник времени с высоким разрешением performance. now (), а также методы performance.mark () и performance, measure () для пометки критических точек в вашем коде и измерения прошедшего между ними времени. Вызов указанных методов создает объекты PerformanceEntry, доступ к которым вы можете получить доступ посредством метода performance. getEntries ().

Браузеры добавляют собственные объекты PerformanceEntry каждый раз, когда браузер загружает новую страницу либо извлекает файл через сеть, и такие автоматически создаваемые объекты PerformanceEntry включают подробные детали хронометража сетевой производительности вашего приложения. Связанный класс PerformanceObserver позволяет указывать функцию, которая должна вызываться, когда создаются новые объекты PerformanceEntry.

Безопасность
В главе было дано общее представление о том, как защищать веб-сайты от уязвимостей типа межсайтовых сценариев (XSS), но мы не вдавались в особые детали. Тема веб-безопасности важна и вы можете потратить некоторое время на ее изучение. Помимо XSS полезно знать о HTTP-заголовке Content-Security-Policy и понимать, как он позволяет запрашивать у веббраузера ограничение возможностей, которые он предоставляет коду JavaScript. В той же степени важно понимать CORS.

WebAssembly

WebAssembly (или "wasm") - это низкоуровневый формат байт-кода виртуальной машины, который предназначен для эффективной интеграции с интерпретаторами JavaScript в веб-браузерах. Существуют компиляторы, которые позволяют компилировать программы на C, C++ и Rust в байт-код WebAssembly и выполнять результирующие программы в веб-браузерах со скоростью, близкой к присущей им изначально, не разрушая песочницу браузера или модель безопасности.

WebAssembly может экспортировать функции, которые разрешено вызывать в программах JavaScript. Типичным вариантом применения WebAssembly была бы компиляция стандартной библиотеки сжатия гі на языке C, чтобы код JavaScript имел доступ к высокоскоростным алгоритмам сжатия и распаковки. Дополнительные сведения ищите на веб-сайте https: //webassembly.org/.

15.15.5. Дополнительные средства обьектов

Document и Window

Объекты Window и Document имеют несколько функциональных средств, которые в главе не рассматривались.

• В объекте Window определены методы alert (), confirm() и prompt (), которые отображают для пользователя простые модальные диалоговые окна. Указанные методы блокируют главный поток. Метод confirm () синхронно возвращает булевское значение, а prompt ( ) синхронно возвращает строку пользовательского ввода. Они не подходят для производственной среды, но удобны для создания простых проектов и прототипов.
• Свойства navigator и screen объекта Window вскользь упоминались в начале главы, но объекты Navigator и Screen, на которые они ссылаются, обладают рядом не раскрытых здесь характеристик, представляющих определенный интерес.
• Метод requestFullscreen () любого объекта Element запрашивает отображение этого элемента (<video> или <canvas>, например) в полноэкранном режиме. Метод exitFullscreen () объекта Document обеспечивает возврат к нормальному режиму отображения.

• Метод requestAnimationFrame () объекта Window принимает функцию в качестве своего аргумента и будет выполнять ее, когда браузер готовится визуализировать следующий кадр. В случае внесения визуальных изменений (особенно повторяющихся или анимационных) помещение вашего кода внутрь вызова requestAnimationFrame ( ) может помочь в обеспечении того, что изменения визуализируются плавно и оптимизированы браузером. • Если пользователь выбирает текст внутри вашего документа, то вы можете выяснить детали выбора посредством метода getSelection () объекта Window и получить выбранный текст с помощью getSelection ().toString (). В некоторых браузерах navigator.clipboard является объектом с асинхронным API-интерфейсом для чтения и установки содержимого буфера обмена системы, чтобы сделать возможными взаимодействия с приложениями вне браузера.
• Малоизвестная особенность веб-браузеров связана с тем, что HTMLэлементы с атрибутом contenteditable="true" позволяют редактировать свое содержимое. Метод document. execCommand () включает средства расширенного редактирования текста для редактируемого содержимого.
• Объект MutationObserver позволяет коду JavaScript следить за изменениями в указанном элементе документа или ниже него. Создайте объект MutationObserver посредством конструктора MutationObserver (), передав функцию обратного вызова, которая должна вызываться, когда делаются изменения. Затем вызовите метод observe () объекта MutationObserver, чтобы указать, за какими частями каких элементов нужно следить.
• Объект IntersectionObserver позволяет коду JavaScript выяснять, какие элементы документа находятся на экране, а какие близки к тому, чтобы находиться на экране. Это особенно полезно для приложений, которые хотят динамически загружать содержимое по запросу, когда пользователь выполняет прокрутку.

15.15.6. События

Количество и разнообразие событий, поддерживаемых веб-платформой, может приводить в растерянность. В главе обсуждались различные типы событий, но ниже перечислено еще несколько, которые вы можете счесть полезными. • Браузеры инициируют события "online" и "offline" в объекте Window, когда браузер получает или утрачивает подключение к Интернету.

• Браузеры инициируют событие "visiblitychange" в объеке Document, когда документ становится видимым или невидимым (обычно из-за того, что пользователь переходит с вкладки на вкладку). В коде JavaScript можно проверять document. visibilityState для определения, виден ли ("visible") документ в текущий момент или же он скрыт ("hidden").
• Браузеры поддерживают замысловатый API-интерфейс для пользовательских интерфейсов с перетаскиванием и обмена данными с приложениями за пределами браузера. В этом API-интерфейсе задействовано несколько событий, в частности "dragstart", "dragover", "dragend" и "drop". Данный API-интерфейс сложно использовать корректно, но он полезен, когда нужен.

О нем важно знать, если вы хотите предоставить пользователям возможность перетаскивания файлов со своего рабочего стола в ваше веб-приложение.

• API-интерфейс Pointer Lock позволяет коду JavaScript скрывать указатель мыши и получать низкоуровневые события мыши в виде относительных величин перемещения, а не абсолютных позиций на экране. Обычно поступать так полезно в играх. Вызовите метод request PointerLock () для элемента, которому желаете направлять все события мыши. Затем события "mousemove", доставленные этому элементу, будут иметь свойства movementX movementY.
• PI-интерфейс Gamepad добавляет поддержку для игровых контроллеров. Применяйте navigator.getGamepads () для получения подключенных объектов Gamepad и прослушивайте события "gamepadconnected" в объекте Window, чтобы получать уведомления при подключении нового контроллера. Объект Gamepad определяет API-интерфейс для запрашивания текущего состояния кнопок в контроллере.

15.15.7. Прогрессивные веб-приложения и служебные воркеры

Термин прогрессивные веб-приложения (Progressive Web App - PWA) представляет собой специальный термин, который описывает веб-приложения, построенные с использованием ряда ключевых технологий. Тщательная документация таких ключевых технологий потребовала бы целой книги, и в настоящей главе они не рассматриваются, но вы должны быть

осведомлены обо всех этих API-интерфейсах. Стоит отметить, что мощные современные PI-интерфейсы подобного рода обычно спроектированы для работы только с защищенными подключениями HTTPS. Веб-сайты, которые по-прежнему применяют URL вида http://, не смогут воспользоваться следующими преимуществами.

• ServiceWorker является потоком воркера, способным перехватывать, инспектировать и отвечать на сетевые запросы из веб-приложения, которое он "обслуживает". Когда веб-приложение регистрирует служебный воркер, код такого воркера становится постоянным в локальном хранилище браузера и при повторном посещении пользователем веб-сайта служебный воркер снова активизируется. Служебные воркеры могут кешировать сетевые ответы (включая файл кода JavaScript), а это значит, что веб-приложения, которые задействуют служебные воркеры, способны эффективно устанавливать сами себя на компьютере пользователя для быстрого запуска и автономного применения. На веб-сайте https://serviceworke.rs предлагается книга Service Worker Cookbook -- ценный ресурс, который дает возможность изучить служебные воркеры и связанные с ними технологи.

• API-интерфейс Cache предназначен для использования со служебными воркерами (но также доступен обыкновенному коду JavaScript за рамками воркеров). Он работает с объектами Request и Response, определенными API-интерфейсом fetch (), и реализует кеш пар Request/Response. APIинтерфейс Cache позволяет служебному воркеру кешировать сценарии и другие ресурсы веб-приложения, которое он обслуживает, и также может содействовать автономному применению веб-приложения (что особенно важно для мобильных устройств).

• Файл веб-манифеста (Web Manifest) имеет формат JSON и описывает веб-приложение, включая название, URL и ссылки на значки различных размеров. Если ваше веб-приложение использует служебный воркер и содержит дескриптор <link rel="manifest">, который ссылается на файл .webmanifest, тогда браузеры (особенно браузеры на мобильных устройствах) могут предоставить вам возможность добавить значок для вебприложения на рабочий стол или домашний экран.

• API-интерфейс Notifications позволяет веб-приложениям отображать уведомления с применением собственной системы уведомлений операционной системы на мобильных и настольных устройствах. Уведомления могут включать изображение и текст, а ваш код может получать событие, когда пользователь щелкает на уведомлении. Использование этого API. интерфейса осложняется тем фактом, что вы обязаны сначала запросить у пользователя разрешение на отображение уведомлений.

• API-интерфейс Push позволяет веб-приложениям, которые имеют служебный воркер (и разрешение от пользователя), подписываться на уведомле. ния от сервера и отображать их, даже когда само веб-приложение не запущено. Push-уведомления распространены на мобильных устройствах, и API-интерфейс Push приближает функциональность веб-приложений к функциональности собственных приложений мобильного устройства.

15.15.8. API-интерфейсы мобильных устройств

Существует несколько API-интерфейсов, которые в первую очередь полезны для веб-приложений, выполняющихся на мобильных устройствах. (К сожалению, ряд таких API-интерфейсов работает только на устройствах Android, но не устройствах iOS.)

• API-интерфейс Geolocation позволяет коду JavaScript (при наличии разрешения от пользователя) определять физическое местоположение пользователя. Он хорошо поддерживается на настольных и мобильных устройствах, в том числе на устройствах iOS. Применяйте navigator.geolocation.getCurrentPosition() для запроса текущего местоположения пользователя и navigator.geolocation. watchPosition () для регистрации обратного вызова, который должен вызываться, когда местоположение пользователя изменяется.

• Метод navigator.vibrate () заставляет мобильное устройство (но не iOS) вибрировать. Зачастую это разрешено только в ответ на пользовательский жест, но вызов метода navigator.vibrate () позволит вашему приложению беззвучно сообщать о том, что

жест был распознан.

• API-интерфейс ScreenOrientation предоставляет веб-приложению возможность запрашивать текущую ориентацию экрана мобильного устройства и также блокировать себя в альбомной или портретной ориентации.

• События "devicemotion" n "deviceorientation" в объекте окна сообщают данные акселерометра и магнитометра для устройства, позволяя вам определять, насколько устройство ускоряется и как пользователь ориентирует его в пространстве. (События работают под управлением iOS.) • API-интерфейс Sensor пока еще широко не поддерживается помимо Chrome на устройствах Android, но он предоставляет коду JavaScript возможность доступа к полному комплекту датчиков мобильного устройства, включая акселерометр, магнитометр и датчик окружающего света. Указанные датчики позволяют коду JavaScript определять, например, в каком направлении смотрит пользователь, или обнаруживать, когда пользователь встряхивает свой телефон.

15.15.10. API-интерфейсы для работы с медиаданными

Функция navigator.mediaDevices.getUserMedia () позволяет коду JavaScript запрашивать доступ к микрофону и/или видеокамере устройства пользователя. Видеопотоки можно отображать в дескрипторе <video> (устанавливая свойство srcObject в поток). Неподвижные кадры видеопотока можно захватывать в закадровом дескрипторе <canvas> с помощью функции drawImage () холста, получая в результате фотографию с относительно низким разрешением. Аудио- и видеопотоки, возвращаемые getUserMedia (), могут быть записаны и закодированы в Blob посредством объекта MediaRecorder.

Более сложный API-интерфейс WebRTC делает возможной передачу обьекТОВ MediaStream по сети, позволяя организовывать, например, одноранговую видеоконференцию.

15.15.11. API-интерфейсы для работы с криптографией и связанные с ними API-интерфейсь

Свойство crypto объекта Window предоставляет метод

getRandomValues () для получения криптостойких псевдослучайных чисел. Другие методы для шифрования, расшифровки, генерации ключей, цифровых подписей и т.д. доступны через crypto.subtle. Имя свойства (subtle - хитроумное) является предупреждением для всех, кто использует эти методы, что надлежащее применение