Web API
Сборник рецептов

Повысьте уровень JavaScript-приложений

Джо Аттарди

Web API Cookbook

Level Up Your JavaScript Applications

Joe Attardi

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Джо Аттарди

Web API
Сборник рецептов

Повысьте уровень
JavaScript-приложений

Астана
«АЛИСТ»
2025

УДК 004.438
ББК 32.988.02-018.2
А92

Аттарди Д.

А92 Web API. Сборник рецептов: пер. с англ. — Астана: АЛИСТ, 2025. — 304 с.: ил.

ISBN 978-601-12-3681-2

Книга посвящена разработке современных веб-приложений с использованием JavaScript и встроенных возможностей браузера на основе Web API. Приведены практические примеры реализации функций, которые ранее требовали сторонних плагинов: диалоговые окна, работа с геолокацией и другими возможностями, предоставляемыми браузером. Освещаются как устоявшиеся, так и находящиеся в стадии разработки API-интерфейсы, позволяющие создавать мощные и интерактивные веб-приложения, глубоко интегрированные с устройствами пользователей. Рассматривается, как браузеры реализуют модель разрешений для предоставления доступа к таким функциям, как геолокация и push-уведомления. Каждая задача в книге сопровождается готовыми рецептами, содержащими описание проблемы, решение с пояснениями и расширенное обсуждение. Книга ориентирована на программистов с опытом работы на JavaScript, знакомых с программным интерфейсом DOM (Document Object Model).

Для программистов

УДК 004.438
ББК 32.988.02-018.2

© 2025 ALIST LLP

Authorized Russian translation of the English edition of ***Web API Cookbook*** ISBN 9781098150693
© 2024 Joseph Attardi.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Авторизованный перевод с английского языка на русский издания ***Web API Cookbook*** ISBN 9781098150693
© 2024 Joseph Attardi.
Перевод опубликован и продается с разрешения компании-правообладателя O'Reilly Media, Inc.

ISBN 978-1-098-15069-3 (англ.)
ISBN 978-601-12-3681-2 (каз.)

© Joseph Attardi, 2024
© Издание на русском языке. ТОО "АЛИСТ", 2025

Содержание

ПРЕДИСЛОВИЕ ... **13**

Возможности современных браузеров ... 14
Недостатки сторонних библиотек ... 14
Для кого предназначена эта книга ... 14
Что содержится в этой книге ... 15
Дополнительные ресурсы ... 15
Соглашения об используемых обозначениях ... 16
Использование примеров кода ... 17
Платформа онлайн-обучения O'Reilly ... 18
Как с нами связаться ... 18
Благодарности ... 18

ГЛАВА 1. АСИНХРОННЫЕ API ... **21**

1.0. Введение ... 21
1.1. Работа с промисами ... 22
1.2. Загрузка изображения с помощью использования резервного изображения (fallback) ... 24
1.3. Связывание промисов ... 26
1.4. Использование ключевых слов *async* и *await* ... 28
1.5. Параллельное использование промисов ... 29
1.6. Анимация элемента с помощью *requestAnimationFrame* ... 30
1.7. Обертывание событий API в промис ... 33

ГЛАВА 2. ПРОСТОЕ СОХРАНЕНИЕ ДАННЫХ С ПОМОЩЬЮ WEB STORAGE API ... **36**

2.0. Введение ... 36
2.1. Проверка поддержки Web Storage ... 38

2.2. Сохранение строковых данных ... 39
2.3. Сохранение простых объектов ... 40
2.4. Сохранение сложных объектов ... 41
2.5. Отслеживание изменений в хранилище ... 46
2.6. Поиск всех известных ключей ... 47
2.7. Удаление данных ... 49

ГЛАВА 3. URL-АДРЕСА И МАРШРУТИЗАЦИЯ ... 51

3.0. Введение ... 51
3.1. Определение относительного URL-адреса ... 52
3.2. Удаление параметров запроса из URL-адреса ... 54
3.3. Добавление параметров запроса к URL-адресу ... 56
3.4. Чтение параметров запроса ... 58
3.5. Создание простого маршрутизатора на стороне клиента ... 60
3.6. Сопоставление URL-адресов с шаблонами ... 62

ГЛАВА 4. СЕТЕВЫЕ ЗАПРОСЫ ... 65

4.0. Введение ... 65
4.1. Отправка запроса с помощью XMLHttpRequest ... 66
4.2. Отправка GET-запроса с помощью Fetch API ... 67
4.3. Отправка POST-запроса с помощью Fetch API ... 69
4.4. Загрузка файла с помощью Fetch API ... 70
4.5. Отправка Beacon ... 71
4.6. Прослушивание удаленных событий с помощью server-sent events ... 72
4.7. Обмен данными с WebSockets в режиме реального времени ... 74

ГЛАВА 5. БАЗА ДАННЫХ INDEXEDDB ... 77

5.0. Введение ... 77
5.1. Создание, чтение и удаление объектов в базе данных ... 79
5.2. Обновление существующей базы данных ... 85
5.3. Выполнение запросов с использованием индексов ... 87
5.4. Поиск строковых значений с помощью курсора ... 90
5.5. Разбивка большого набора данных на страницы ... 92
5.6. Использование промисов с IndexedDB API ... 94

ГЛАВА 6. НАБЛЮДЕНИЕ ЗА ЭЛЕМЕНТАМИ DOM 97

6.0. Введение 97
6.1. Отложенная загрузка изображения при прокрутке 99
6.2. Обертывание *IntersectionObserver* промисом 101
6.3. Автоматическая пауза при воспроизведении видео 102
6.4. Анимация изменений высоты 103
6.5. Изменение содержимого элемента в зависимости от размера 105
6.6. Применение перехода в момент появления элемента в поле зрения 107
6.7. Использование режима бесконечной прокрутки 108

ГЛАВА 7. ФОРМЫ 110

7.0. Введение 110
7.1. Заполнение поля формы из локального хранилища 111
7.2. Отправка формы с помощью Fetch и FormData API 112
7.3. Отправка формы в формате JSON 114
7.4. Создание обязательного поля формы 116
7.5. Ограничения при вводе числа 117
7.6. Определение шаблона валидации 118
7.7. Валидация формы 119
7.8. Применение пользовательской логики валидации 122
7.9. Проверка группы флажков 124
7.10. Асинхронная проверка поля формы 127

ГЛАВА 8. API ВЕБ-АНИМАЦИИ 130

8.0. Введение 130
8.1. Применение эффекта "пульсации" при нажатии кнопки 132
8.2. Запуск и остановка анимации 135
8.3. Анимация вставки и удаления элементов DOM 136
8.4. Реверсирование анимации 137
8.5. Отображение индикатора прокрутки 141
8.6. Создание подпрыгивающего элемента 143
8.7. Одновременный запуск нескольких анимаций 144
8.8. Отображение анимации загрузки 146
8.9. Соблюдение в анимации предпочтений пользователя 148

ГЛАВА 9. WEB SPEACH API 150

9.0. Введение 150
9.1. Добавление продиктованного текста в текстовое поле 152
9.2. Создание Promise-помощника для распознавания речи 155
9.3. Получение доступных голосов 156
9.4. Синтез речи 157
9.5. Настройка параметров синтеза речи 159
9.6. Автоматическая приостановка речи 160

ГЛАВА 10. РАБОТА С ФАЙЛАМИ 161

10.0. Введение 161
10.1. Загрузка текста из файла 161
10.2. Загрузка изображения из URL-адреса данных 164
10.3. Загрузка видео в качестве URL-адреса объекта 166
10.4. Загрузка изображения с помощью перетаскивания 167
10.5. Проверка и запрос разрешений 170
10.6. Экспорт данных API в файл 171
10.7. Экспорт данных API со ссылкой для скачивания 173
10.8. Загрузка файла с помощью перетаскивания 175

ГЛАВА 11. ИНТЕРНАЦИОНАЛИЗАЦИЯ 177

11.0. Введение 177
11.1. Форматирование даты 178
11.2. Получение частей отформатированной даты 178
11.3. Форматирование относительной даты 179
11.4. Форматирование чисел 181
11.5. Округление знаков после точки 182
11.6. Форматирование ценового диапазона 183
11.7. Форматирование единиц измерения 184
11.8. Применение правил плюрализации 184
11.9. Подсчет символов, слов и предложений 186
11.10. Форматирование списков 187
11.11. Сортировка массива имен 188

ГЛАВА 12. ВЕБ-КОМПОНЕНТЫ ... 190

12.0. Введение ... 190
12.1. Создание компонента для отображения сегодняшней даты ... 193
12.2. Создание компонента для форматирования пользовательской даты ... 194
12.3. Создание компонента обратной связи ... 196
12.4. Создание компонента профильной карточки ... 200
12.5. Создание компонента изображения с отложенной загрузкой ... 202
12.6. Создание компонента раскрытия информации ... 204
12.7. Создание стилизованного компонента кнопки ... 207

ГЛАВА 13. ЭЛЕМЕНТЫ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ ... 211

13.0. Введение ... 211
13.1. Создание диалогового окна предупреждения ... 212
13.2. Создание диалогового окна подтверждения ... 215
13.3. Создание веб-компонента диалогового окна подтверждения ... 218
13.4. Использование элемента раскрытия информации ... 220
13.5. Отображение всплывающего окна ... 222
13.6. Ручное управление всплывающим окном ... 223
13.7. Позиционирование всплывающего окна относительно элемента ... 224
13.8. Отображение всплывающей подсказки ... 227
13.9. Отображение уведомления ... 229

ГЛАВА 14. ИНТЕГРАЦИЯ УСТРОЙСТВ ... 231

14.0. Введение ... 231
14.1. Считывание состояния батареи ... 231
14.2. Считывание состояния сети ... 234
14.3. Определение местоположения устройства ... 235
14.4. Отображение местоположения устройства на карте ... 237
14.5. Копирование и вставка текста ... 238
14.6. Совместное использование контента с помощью Web Share API ... 241
14.7. Создание вибрации устройства ... 242
14.8. Настройка ориентации устройства ... 243

ГЛАВА 15. ИЗМЕРЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ 245

15.0. Введение 245
15.1. Измерение производительности загрузки страниц 246
15.2. Измерение эффективности использования ресурсов 247
15.3. Поиск самых медленных ресурсов 247
15.4. Определение тайминга для конкретного ресурса 248
15.5. Профилирование производительности рендеринга 249
15.6. Профилирование многоэтапных задач 251
15.7. Прослушивание показателей производительности 253

ГЛАВА 16. РАБОТА С КОНСОЛЬЮ 255

16.0. Введение 255
16.1. Стилизации вывода консоли 255
16.2. Использование уровней в журналах сообщений 256
16.3. Создание именованных регистраторов 257
16.4. Отображение массива объектов в таблице 258
16.5. Использование консольных таймеров 260
16.6. Использование консольных групп 261
16.7. Использование счетчиков 263
16.8. Регистрация переменной и ее значения 264
16.9. Протоколирование трассировки стека 265
16.10. Проверка ожидаемых значений 266
16.11. Изучение свойств объекта 267

ГЛАВА 17. CSS 269

17.0. Введение 269
17.1. Выделение текстовых областей 269
17.2. Предотвращение появления текста без стилизации 272
17.3. Анимация переходов DOM 274
17.4. Изменение таблиц стилей во время выполнения 276
17.5. Условная установка CSS-класса 277
17.6. Соответствие медиазапросам 277
17.7. Получение вычисленного стиля элемента 278

ГЛАВА 18. МЕДИА **281**

18.0. Введение 281
18.1. Запись экрана 281
18.2. Захват изображения с камеры пользователя 284
18.3. Захват видео с камеры пользователя 286
18.4. Определение возможностей системной поддержки медиа 288
18.5. Применение видеофильтров 290

ГЛАВА 19. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ **292**

19.0. Введение 292
19.1. В защиту сторонних библиотек 292
19.2. Определяйте функции, а не версии браузера 292
19.3. Полифилы 293
19.4. Заглядывая в будущее 293

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ **296**

ОБ АВТОРЕ **302**

ОБ ИЗОБРАЖЕНИИ НА ОБЛОЖКЕ **303**

Предисловие

JavaScript прошел долгий путь с момента своего появления в конце 1995 года. В первые дни возможности основных API, встроенных в веб-браузеры, были ограничены. Для более расширенной функциональности обычно требовались сторонние библиотеки JavaScript, а в некоторых случаях даже плагины для браузера.

Веб API — это набор глобальных объектов и функций, предоставляемых браузером. Ваш код JavaScript может использовать их для взаимодействия с объектной моделью документа (document object model, DOM), осуществления сетевого взаимодействия, интеграции с собственными функциями устройства и многого другого.

Полифилы

Старые браузеры могут применять некоторые из этих API, используя полифилы[1]. Полифил — это библиотека JavaScript, которая реализует недостающие функциональные возможности. Полифилы обычно используются для функций веб API, которые не реализованы в старых браузерах.

Несмотря на преимущества, у полифилов есть некоторые недостатки.

- Они загружаются как сторонние библиотеки, что увеличивает размер вашего пакета.
- Как правило, разработчики браузеров не поддерживают их, поэтому в них могут быть ошибки или несоответствия.
- Некоторые расширенные функциональные возможности не могут быть эффективно полифилированы или не могут быть полифилированы вообще.

[1] Полифил (полифилл; от англ. *polyfill*) — код, реализующий какую-либо функциональность, которая не поддерживается в некоторых версиях веб-браузеров. Обычно реализуется как библиотека JavaScript, обеспечивающая поддержку веб-стандарта HTML5 в версиях браузеров, где поддержка этих функций частично или полностью отсутствует. Применение полифилов ("полифинирование") обеспечивает более или менее единообразное отображение веб-страниц в разных веб-браузерах. — *Прим. пер.*

Возможности современных браузеров

Современные веб API обладают двумя большими преимуществами для веб-платформы.

Больше не требуются подключаемые модули.

В прошлом большая часть этой функциональности была доступна только для собственных приложений или громоздких подключаемых модулей для браузера. (Помните ActiveX и Flash?)

Меньше зависимостей от программного обеспечения сторонних разработчиков.

Современные браузеры предоставляют значительную функциональность, для которой раньше требовались сторонние библиотеки JavaScript. Популярные библиотеки, такие как jQuery, Lodash и Moment, теперь больше не нужны.

Недостатки сторонних библиотек

Сторонние библиотеки могут быть полезны при работе со старыми браузерами или с более новыми функциями, но их применение сопряжено с некоторыми затратами.

Требуется загружать больше кода.

Использование библиотек увеличивает объем JavaScript-кода, который должен загружаться браузером. Независимо от того, поставляется ли он в комплекте с вашим приложением или загружается отдельно из сети доставки контента (content delivery network, CDN), вашему браузеру все равно придется его загружать. Это может привести к увеличению как времени загрузки, так и расхода заряда батареи на мобильных устройствах.

Повышенный риск.

Библиотеки с открытым исходным кодом, даже популярные, могут не поддерживаться. При обнаружении ошибок и уязвимостей обновление не гарантируется. Браузеры, как правило, поддерживаются крупными компаниями (основные браузеры — от Google, Mozilla, Apple и Microsoft), и вероятнее всего, что подобные проблемы будут устранены.

Это не значит, что сторонние библиотеки плохи. В них также есть много преимуществ, особенно если вам нужна поддержка старых браузеров. Как и во всем, что касается разработки программного обеспечения, использование библиотек требует соблюдения баланса.

Для кого предназначена эта книга

Эта книга предназначена для разработчиков программного обеспечения, имеющих хотя бы небольшой опыт работы с JavaScript, которые желают получить максимальную отдачу от веб-платформы.

Предполагается, что вы хорошо разбираетесь в самом языке JavaScript: синтаксисе, особенностях языка и стандартных библиотечных функциях. Вы также должны иметь практические знания о DOM API, используемом для создания интерактивных браузерных приложений на JavaScript.

В этой книге содержится множество рецептов, которые подойдут разработчикам с любым уровнем квалификации и опыта.

Что содержится в этой книге

Каждая глава содержит набор *рецептов* — примеров кода для выполнения конкретной задачи. Каждый рецепт состоит из трех разделов.

Задача

Описывает проблему, которую решает этот рецепт.

Решение

Содержит код и пояснения, которые реализуют рецептурное решение.

Обсуждение

В этом разделе могут содержаться дополнительные примеры кода и сравнения с другими методами.

Примеры кода и демонстрационные версии доступны на сопутствующем веб-сайте, **https://WebAPIs.info**.

Дополнительные ресурсы

По своей природе Интернет постоянно меняется. В Интернете доступно множество отличных ресурсов, которые помогут прояснить любые возникающие вопросы.

CanIUse.com

На момент написания этой книги некоторые API-интерфейсы все еще находились в стадии разработки или "экспериментального использования". Следите за примечаниями о совместимости в рецептах, в которых используются эти API. Для большинства функций вы можете ознакомиться с последними данными о совместимости по адресу **https://CanIUse.com**. Можете выполнить поиск по названию функции и просмотреть последнюю информацию о том, какие версии браузеров поддерживают API, а также о любых ограничениях или предостережениях для конкретных версий браузера.

MDN Web Docs

MDN Web Docs (**https://oreil.ly/rLxi7**) — это документация по API для всевозможных веб-приложений. Она в подробностях охватывает все описанные в этой книге API-интерфейсы, а также другие темы, такие как CSS и HTML. В ней содержатся детальные статьи и руководства, а также спецификации API.

Спецификации

В случае сомнений, спецификация функции или API являются определяющим ресурсом. Это не самое интересное чтение, но это хорошее место для поиска подробной информации о крайних случаях использования или ожидаемом поведении.

Разные API имеют различные стандарты, но большинство из них можно найти в Web Hypertext Application Technology Working Group (WHATWG; **https://oreil.ly/PR0x7**) или World Wide Web Consortium (W3C; **https://oreil.ly/dFokl**).

Стандарты для ECMAScript (которые определяют функции языка JavaScript) поддерживаются и разрабатываются Международным техническим комитетом Ecma 39, более известным как TC39 (**https://tc39.es/**).

Соглашения об используемых обозначениях

В книге используются следующие общепринятые типографские обозначения.

Курсив

Обозначает новые термины.

Жирный шрифт

Обозначает URL-адреса, адреса электронной почты, элементы интерфейса программ.

`Моноширинный шрифт`

Используется в листингах программ, а также внутри абзацев для обозначения таких элементов программы, как имена переменных и функций, базы данных, типы данных, переменные среды, операторы, методы.

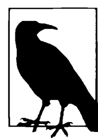
`Моноширинный полужирный`

Для обозначения команд или другого текста, который должен ввести пользователь, а также для обозначения ключевых слов в листингах программ.

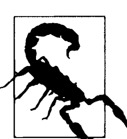
`Моноширинный курсив`

Для обозначения текста, который должен быть заменен значениями, введенными пользователем или значениями, определяемыми контекстом, а также для обозначения комментариев в листингах программ.

Данный элемент обозначает подсказку или совет.



Данный элемент обозначает общее замечание.



Данный элемент обозначает предупреждение или предостережение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресу **https://github.com/joeattardi/web-api-cookbook**. Также посетите сопутствующий книге веб-сайт (**https://www.webapis.info/**), где многие фрагменты кода и рецепты из этой книги представлены в виде полных живых работающих примеров. Если у вас возникнут технические вопросы или проблемы с использованием примеров кода, пожалуйста, отправьте электронное письмо по адресу **bookquestions@oreilly.com**.

Эта книга предназначена для того, чтобы помочь вам в выполнении вашей работы. В целом, если в этой книге предлагается пример кода, вы можете использовать его в своих программах и документации. Вам не нужно обращаться к нам за разрешением, если только вы не воспроизводите значительную часть кода. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешение не требуется. Для продажи или распространения примеров из книг O'Reilly требуется разрешение. Для ответа на вопрос со ссылкой на эту книгу и примеры кода не требуется разрешение. Включение значительного количества для использования примера кода из этой книги в документацию к вашему продукту требует разрешения.

Мы ценим указание авторства, но, как правило, не требуем этого. Указание авторства обычно включает название, автора, издателя и ISBN. Например: "Web API Cookbook by Joseph Attardi (O'Reilly). Copyright 2024 Joe Attardi, 978-1-098-15069-3".

Если вы считаете, что применение вами примеров кода выходит за рамки добросовестного использования или вышеуказанного разрешения, не стесняйтесь обращаться к нам по адресу **permissions@oreilly.com**.

Платформа онлайн-обучения O'Reilly

O'REILLY®

Более 40 лет компания O'Reilly Media организует технические и бизнес-тренинги, передает знания и опыт, чтобы помочь компаниям достичь успеха.

Уникальная сеть экспертов и новаторов делится своими знаниями и опытом через книги, статьи и платформу онлайн-обучения. Эта платформа предоставляет доступ по требованию к интерактивным учебным курсам, углубленным программам обучения, интерактивным средам кодирования, а также к обширной коллекции текстовых и видеоматериалов от O'Reilly и более 200 других издательств. Дополнительная информация представлена на сайте **https://oreilly.com**.

Как с нами связаться

Просим направлять комментарии и вопросы относительно данной книги в издательство по адресу:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA, 95472

800-998-9938 (в США или Канаде)

707-829-0515 (международный или местный)

707-829-0104 (факс)

support@oreily.com

https://www.oreily.com/about/contact.html

На веб-странице книги можно ознакомиться с редакциями, примерами и другой дополнительной информацией. Она доступна по адресу **https://oreil.ly/web-api-cookbook**.

Новости и информацию о наших книгах и курсах можно найти на сайте **https://oreilly.com**.

Ищите нас на LinkedIn: **https://linkedin.com/company/oreilly-media**.

Следите за нами в Twitter: **https://twitter.com/oreillymedia**.

Смотрите нас на YouTube: **https://www.youtube.com/oreillymedia**.

Благодарности

Прежде всего, я хотел бы от всего сердца поблагодарить мою семью и друзей за поддержку, особенно мою жену Лиз и сына Бенджамина, за то, что они терпели мое бесконечное клацанье клавишами. Когда я работаю, я, как правило, печатаю очень быстро и громко.

Спасибо Аманде Куинн, старшему редактору по подбору контента, за то, что она пригласила меня в качестве автора O'Reilly. За эти годы я прочитал бесчисленное множество книг O'Reilly и никогда не думал, что однажды напишу одну из них сам. Также спасибо Луизе Корриган за то, что познакомила меня с Амандой и положила начало процессу (и тем, кто работал со мной несколько лет назад, когда я публиковал свою самую первую книгу!).

Особая благодарность Вирджинии Уилсон, старшему редактору отдела разработки, за то, что помогала мне на протяжении всего процесса написания книги и регулярно встречалась со мной, чтобы не дать мне сбиться с пути.

Я также хотел бы поблагодарить замечательных технических рецензентов этой книги: Мартину Дауден, Шалка Нитлинга, Сару Шук и Адама Скотта. Благодаря их полезным отзывам книга стала намного лучше.

Наконец, я хотел бы поблагодарить команды, разрабатывающие все эти современные веб-интерфейсы. Без них этой книги не было бы!

ГЛАВА 1
Асинхронные API

1.0. Введение

Многие API, описанные в этой книге, являются асинхронными. При вызове одной из этих функций или методов результат может быть получен не сразу. Разные API имеют различные механизмы возврата результата, когда он будет готов.

Функция обратного вызова (callback)

Самый простой асинхронный шаблон — это функция обратного вызова (callback). Это функция, которую вы передаете асинхронному API. Когда работа завершена, API вызывает вашу callback-функцию с результатом. Функции обратного вызова могут использоваться как сами по себе, так и как часть других асинхронных шаблонов.

Событие (event)

Многие API-интерфейсы браузеров основаны на *событиях*. Событие — это некое действие, которое происходит асинхронно. Вот несколько примеров событий:

- нажата кнопка;
- перемещена мышь;
- выполнен сетевой запрос;
- произошла ошибка.

У события есть имя, например `click` или `mouseover`, и объект с данными о произошедшем событии. Последний может содержать информацию, например, о том, на каком элементе был сделан щелчок, или код состояния HTTP. Когда вы прослушиваете событие, вы предоставляете функцию обратного вызова, которая получает объект события в качестве аргумента.

Объекты, генерирующие события, реализуют интерфейс `EventTarget`, который предоставляет методы `addEventListener` и `removeEventListener`. Для того чтобы прослушать событие для элемента или другого объекта, вы можете вызвать для него метод

addEventListener, передав имя события и функцию-обработчик. Функция обратного вызова вызывается каждый раз, когда запускается событие, пока оно не будет удалено. Прослушиватель[1] можно удалить вручную, вызвав метод removeEventListener, или во многих случаях прослушиватели автоматически удаляются браузером, когда объекты уничтожаются или удаляются из DOM.

Промис (promise)

Многие новые API-интерфейсы используют промисы. Promise — это объект, возвращаемый из функции, который является плейсхолдером для конечного результата асинхронного действия. Вместо того чтобы прослушивать событие, вы вызываете then для объекта Promise. Вы передаете callback-функцию в then, которая в конечном итоге вызывается с результатом в качестве аргумента. Для обработки ошибок вы передаете другую функцию обратного вызова методу catch промиса.

Промис *выполняется*, когда операция завершается успешно, и отклоняется, когда возникает ошибка. Значение, полученное в ходе успешного выполнения, передается в качестве аргумента для последующего вызова callback-функции в then, либо передается отклоненное значение в качестве аргумента callback-функции в catch.

Существует несколько ключевых различий между событиями и промисами.

- Обработчики событий запускаются несколько раз, в то время как обратный вызов в then выполняется только один раз. Вы можете рассматривать Promise как одноразовую операцию.
- Если вы вызываете then в Promise, вы всегда будете получать результат (если он есть). В этом заключается отличие от событий, в которых, в случае если событие происходит до добавления прослушивателя, оно теряется.
- В Promise встроен механизм обработки ошибок. В случае событий обычно требуется прослушивать события ошибки (error) для обработки условий ошибки.

1.1. Работа с промисами

Задача

Вы хотите вызвать API, который использует промисы, и получить результат.

Решение

Вызовите then объекта Promise для обработки результата в функции обратного вызова. Для того чтобы обработать возможные ошибки, добавьте вызов catch.

[1] Или слушатель события (listener). — *Прим. ред.*

Представьте, что у вас есть функция `getUsers`, которая отправляет сетевой запрос на загрузку списка пользователей. Эта функция возвращает `Promise`, который в конечном итоге преобразуется в список пользователей (пример 1.1).

Пример 1.1. Использование API, основанного на Promise

```
getUsers()
  .then(
    // Эта функция вызывается, когда загружен список пользователей.
    userList => {
      console.log('User List:');
      userList.forEach(user => {
        console.log(user.name);
      });
    }
  ).catch(error => {
    console.error('Failed to load the user list:', error);
  });
```

Обсуждение

`Promise`, возвращаемый из `getUsers`, является объектом с методом `then`. Когда загружается список пользователей, выполняется обратный вызов, передаваемый в `then`, с использованием списка пользователей в качестве аргумента.

В этом промисе также есть метод `catch` для обработки ошибок. Если при загрузке списка пользователей возникает ошибка, вызывается callback-функция, переданная в `catch`, с объектом `error`. Выполняется только один из этих обратных вызовов — в зависимости от результата.

Всегда обрабатывайте ошибки

Важно всегда обрабатывать ошибки, связанные с `Promise`. Если вы этого не сделаете и промис будет отклонен, браузер выдаст исключение для необработанного отклонения, что может привести к сбою вашего приложения.

Для того чтобы предотвратить аварийное завершение работы вашего приложения из-за необработанного отклонения, добавьте прослушиватель (listener) к объекту `window` для события `unhandledrejection`. Если какой-либо промис отклонен и вы не обработали его с помощью `catch`, сработает это событие. Здесь вы можете предпринять такое действие, как регистрация ошибки в журнале.

1.2. Загрузка изображения с помощью использования резервного изображения (fallback)

Задача

Вы хотите загрузить изображение для показа на странице. Если при загрузке изображения произошла ошибка, вы хотите использовать известный вам URL-адрес изображения в качестве запасного варианта.

Решение

Создайте элемент `Image` программно и отслеживайте его события `load` (загрузка) и `error` (ошибка). Если срабатывает событие `error`, замените желаемое изображение резервным. Как только загрузится запрошенное изображение или изображение-заменитель, при желании добавьте его в DOM.

Для более аккуратного API вы можете обернуть всё в `Promise`. `Promise` либо завершается добавлением `Image`, либо отклоняется с ошибкой, если ни изображение, ни резервный вариант (fallback) не могут быть загружены (пример 1.2).

Пример 1.2. Загрузка изображения с помощью резервной копии

```
/**
 * Загружает изображение. Если происходит ошибка, использует
 * резервную копию (fallback).
 *
 * @param url - URL-адрес изображения
 * @param fallbackUrl - URL-адрес резервной копии на случай ошибки
 * @returns Promise, который разрешается в элемент изображения для включения в DOM
 */
function loadImage(url, fallbackUrl) {
  return new Promise((resolve, reject) => {
    const image = new Image();

    // Попытка загрузки изображения из URL.
    image.src = url;

    // Изображение генерирует событие 'load' в случае успешной загрузки.
    image.addEventListener('load', () => {
      // Загруженное изображение используется для разрешения промиса.
      resolve(image);
```

```
    // В случае неудачи генерируется событие 'error'.
    image.addEventListener('error', error => {
      // Отклонение промиса по одному из двух сценариев:
      // (1) отсутствует URL-адрес резервной копии;
      // (2) загрузка из URL-адреса резервной копии терпит неудачу.
      if (!fallbackUrl || image.src === fallbackUrl) {
        reject(error);
      } else {
        // Если это выполняется, значит, загрузка оригинального изображения неудачная.
        // Пробуем загрузку резервной копии.
        image.src = fallbackUrl;
      }
    });
  });
}
```

Обсуждение

Функция `LoadImage` принимает основной и резервный URL-адреса и возвращает промис. Затем она создает новое изображение и присваивает его атрибуту `src` указанный URL-адрес. Браузер пытается загрузить изображение.

Возможны три исхода.

Успешная загрузка.

Если изображение загружается успешно, запускается событие `load`. Обработчик события разрешает промис изображением, которое затем может быть вставлено в DOM.

Резервный вариант.

Если изображение не загружается, запускается событие `error`. Обработчик ошибок присваивает атрибуту `src` значение резервного URL-адреса, и браузер пытается загрузить резервное изображение. В случае успеха запускается событие `load`, которое разрешает промис с использованием резервного изображения.

Сбой.

Если ни основное изображение, ни резервное изображение не могут быть загружены, обработчик ошибок отклоняет промис с событием `error`.

Событие `error` происходит каждый раз при возникновении ошибки загрузки. Обработчик сначала проверяет, не связан ли сбой с резервным URL-адресом. Если это так, значит, не удалось загрузить ни с исходного, ни с резервного варианта URL-адреса. Это случай сбоя, поэтому промис отклоняется.

Если это не резервный URL-адрес, это означает, что запрошенный URL-адрес не удалось загрузить. Теперь устанавливается резервный URL-адрес и делается попытка загрузить изображение с него.

Здесь важен порядок проверок. Без первой проверки, если резервный вариант не будет загружен, обработчик ошибок запустит бесконечный цикл установки (недопустимого) резервного варианта URL, запрашивая его и снова запуская событие `error`.

В примере 1.3 показано, как использовать эту функцию `loadImage`.

Пример 1.3. Использование функции `loadImage`

```
loadImage('https://example.com/profile.jpg', 'https://example.com/fallback.jpg')
  .then(image => {
  // container - это элемент DOM, в который будет вставлено изображение.
  container.appendChild(image);
}).catch(error => {
  console.error('Image load failed');
 });
```

1.3. Связывание промисов

Задача

Вы хотите последовательно вызвать несколько API-интерфейсов на основе *промисов*. Каждая операция зависит от результата предыдущей.

Решение

Используйте цепочку промисов для последовательного выполнения асинхронных задач. Представьте себе приложение для ведения блога с двумя API, и они оба возвращают промисы.

`getUser(id)`

Загружает пользователя с заданным идентификатором пользователя.

`getPosts(user)`

Загружает все записи блога для данного пользователя.

Если вы хотите загрузить записи пользователя, вам сначала нужно загрузить объект `user` — вы не можете вызвать `getPosts`, пока не будут загружены данные пользователя. Вы можете сделать это, связав два промиса вместе, как показано в примере 1.4.

Пример 1.4. Использование последовательности промисов

```
/**
 * Загружает заголовки постов пользователя.
 * @param userId - идентификатор пользователя, посты которого вы желаете загрузить
 * @returns Promise, который разрешается в массив заголовков постов
 */
function getPostTitles(userId) {
  return getUser(userId)
    // Функция обратного вызова с загруженным объектом пользователя.
    .then(user => {
      console.log(`Getting posts for ${user.name}`);
      // Этот Promise также возвращается из .then
      return getPosts(user);
    })
    // Вызов then из метода getPosts промиса.
    .then(posts => {
      // Возвращает другой Promise, который разрешится в массив заголовков постов
      return posts.map(post => post.title);
    })
    // Вызывается, если отклоняется как getUser, так и getPosts
    .catch(error => {
      console.error('Error loading data:', error);
    });
}
```

Обсуждение

Значение, возвращаемое обработчиком then промиса, переносится в новый промис. Этот промис возвращается из самого метода then. Это означает, что возвращаемое значение then также является промисом, поэтому вы можете привязать к нему другое значение then. Вот так вы и создаете цепочку обещаний.

getUser возвращает промис, который преобразуется в объект user. Обработчик then вызывает getPosts и возвращает результирующий промис, который возвращается снова из then, так что вы можете вызвать then еще раз, чтобы получить конечный результат — массив сообщений пользователя.

В конце цепочки находится вызов catch для обработки любых ошибок. Конструкция работает как блок try/catch. Если в какой-либо точке цепочки возникает ошибка, вызывается обработчик catch с этой ошибкой, а остальная часть цепочки не выполняется.

1.4. Использование ключевых слов *async* и *await*

Задача

Вы работаете с API, который возвращает промис, но хотите, чтобы код работал более линейным или синхронным способом.

Решение

Используйте ключевое слово `await` вместе с промисом вместо вызова `then` (пример 1.5). Рассмотрим еще раз функцию `getUsers` из рецепта 1.1. Эта функция возвращает промис, который преобразуется в список пользователей.

Пример 1.5. Использование ключевого слова await

```
// Для того чтобы в своем теле использовать await, функция должна быть
// объявлена с ключевым словом async.
async function listUsers() {
  try {
    // Эквивалентно getUsers().then(...)
    const userList = await getUsers();
    console.log('User List:');
    userList.forEach(user => {
      console.log(user.name);
    });
  } catch (error) { // Эквивалентно .catch(...)
    console.error('Failed to load the user list:', error);
  }
}
```

Обсуждение

`await` — это альтернативный синтаксис для работы с промисами. Вместо вызова `then` с callback-функцией, которая принимает результат в качестве аргумента, `await` эффективно "приостанавливает" выполнение остальной части функции и возвращает результат, когда промис выполнен.

Если промис отклонен, `await` выдает альтернативное значение. Алгоритм выполняется с помощью стандартного блока `try/catch`.

1.5. Параллельное использование промисов

Задача

Вы хотите выполнить серию асинхронных задач параллельно, используя промисы.

Решение

Соберите все промисы и передайте их в функцию `Promise.all`. Она принимает массив промисов и ожидает завершения их выполнения, а возвращает новый промис, который выводится после выполнения всех промисов или отклоняется, если какой-либо из данных промисов отклонен (пример 1.6).

Пример 1.6. Загрузка нескольких пользователей с помощью `Promise.all`

```
// Загрузка сразу трех пользователей.
Promise.all([
  getUser(1),
  getUser(2),
  getUser(3)
]).then(users => {
  // users - это массив объектов, значения возвращаются
  // с помощью параллельных вызовов getUsers.
}).catch(error => {
  // Если какой-либо из промисов отвергается.
  console.error('One of the users failed to load:', error);
});
```

Обсуждение

Если у вас несколько задач, которые не зависят друг от друга, использовать `Promise.all` — хороший выбор. В примере 1.6 трижды вызывается `getUser`, каждый раз с передачей нового идентификатора пользователя. Эти промисы собираются в массив, который передается в `Promise.all`. Все три запроса выполняются параллельно.

`Promise.all` возвращает еще один промис. Как только все три пользователя успешно загрузятся, этот новый промис будет выполнен с массивом, содержащим загруженных пользователей. Индекс каждого результата соответствует индексу промиса во входном массиве. В нашем случае он возвращает массив с пользователями `1`, `2` и `3` в указанном порядке.

Что делать, если одному или нескольким из этих пользователей не удалось загрузиться? Возможно, один из идентификаторов пользователя не существует или произошла временная сетевая ошибка. Если *какой-либо* из промисов, передаваемых в `Promise.all`, был отвергнут, новый промис также немедленно отвергается. Значение отказа такое же, как и у отклоненного промиса.

Если одного из пользователей не удается загрузить, промис, возвращаемый `Promise.all`, отклоняется с ошибкой, которая произошла. Результаты других промисов будут потеряны.

Если же вы хотите получить результаты любых отработанных промисов (или ошибок из других отклоненных промисов), можете использовать функцию `Promise.allSettled`. С ее помощью новый промис возвращается точно так же, как и посредством `Promise.all`. Однако этот промис всегда выполняется, как только работа со всеми промисами закончится (вне зависимости от того, выполнены они или отклонены).

Как показано в примере 1.7, выходное значение представляет собой массив, каждый элемент которого имеет свойство `status`. Оно принимает значение либо `fulfilled` (выполнено), либо `rejected` (отклонено), в зависимости от результата выполнения этого промиса. Если `status` имеет значение `fulfilled`, объект также имеет свойство `value`, которое является выходным значением. С другой стороны, если `status` имеет значение `rejected`, у него теперь есть свойство `reason`, которое указывает на причину отклонения.

Пример 1.7. Использование `Promise.allSettled`

```
Promise.allSettled([
  getUser(1),
  getUser(2),
  getUser(3)
]).then(results => {
  results.forEach(result => {
    if (result.status === 'fulfilled') {
      console.log('- User:', result.value.name);
    } else {
      console.log('- Error:', result.reason);
    }
  });
});
// Здесь не надо ловить ошибку, так как allSettled выполняется всегда.
```

1.6. Анимация элемента с помощью *requestAnimationFrame*

Задача

Вы хотите эффективно анимировать элемент с помощью JavaScript.

Решение

Используйте функцию `requestAnimationFrame`, чтобы запланировать запуск обновлений анимации с регулярными интервалами.

Представьте, что у вас есть элемент div, который вы хотите скрыть с помощью анимации затухания (fade). Это делается путем интервального изменения настройки прозрачности с помощью функции обратного вызова, передаваемой в requestAnimationFrame (пример 1.8). Продолжительность каждого интервала зависит от желаемого количества кадров в секунду (frames per second, FPS) в анимации.

Пример 1.8. Анимация постепенного исчезновения с использованием requestAnimationFrame

```
const animationSeconds = 2; // Анимировать на 2 секунды
const fps = 60; // Приятная глазу скорость анимации

// Интервал между кадрами
const frameInterval = 1000 / fps;

// Общее число кадров анимации
const frameCount = animationSeconds * fps;

// Величина изменения прозрачности между кадрами
const opacityIncrement = 1 / frameCount;

// ВременнАя метка последнего кадра
let lastTimestamp;

// Начальное значение прозрачности
let opacity = 1;

function fade(timestamp) {
  // Установить последнюю отметку времени, если она не существует.
  if (!lastTimestamp) {
    lastTimestamp = timestamp;
  }

  // Вычислить время, прошедшее с последней смены кадра.
  // Если прошло еще недостаточно времени, запланируйте еще один
  // вызов этой функции и возврат.
  const elapsed = timestamp - lastTimestamp;
  if (elapsed < frameInterval) {
    requestAnimationFrame(animate);
    return;
  }
```

```
    // Время для следующего кадра. Запомнить значение.
    lastTimestamp = timestamp;

    // Настроить прозрачность и убедиться, что это значение больше 0.
    opacity = Math.max(0, opacity - opacityIncrement)
    box.style.opacity = opacity;

    // Если значение прозрачности не достигло 0, запланировать
    // еще один вызов этой функции.
    if (opacity > 0) {
      requestAnimationFrame(animate);
    }
}

// Запланировать первый вызов функции анимации.
requestAnimationFrame(fade);
```

Обсуждение

Это красивый и производительный способ анимации элементов с помощью JavaScript, который хорошо поддерживается браузером. Поскольку анимация выполняется асинхронно, она не блокирует основной поток браузера. Если пользователь переключается на другую вкладку, анимация приостанавливается, т. е. `requestAnimationFrame` не вызывается без необходимости.

Когда вы планируете запуск функции с помощью `requestAnimationFrame`, функция вызывается перед следующей операцией перерисовки. Периодичность зависит от браузера и частоты обновления экрана.

Перед анимацией в примере 1.8 выполняются некоторые вычисления на основе заданной длительности анимации (2 секунды) и частоты кадров (60 кадров в секунду). Программа вычисляет общее количество кадров и использует длительность анимации для расчета продолжительности каждого кадра. Если вам нужна другая частота кадров, которая не соответствует частоте обновления экрана системой, программа отслеживает, когда было выполнено последнее обновление анимации, чтобы поддерживать заданную частоту кадров.

Затем, на основе количества кадров, рассчитывается величина прозрачности, действующая в каждом кадре.

Функция `fade` передается в вызов `requestAnimationFrame` через определенные интервалы времени. Каждый раз, когда браузер вызывает эту функцию, он передает временну́ю метку. Функция `fade` вычисляет, сколько времени прошло с момента последнего кадра. Если прошло еще недостаточно времени, программа ничего не делает и просит браузер повторить вызов в следующий раз.

По прошествии достаточного времени выполняется следующий этап анимации. Берется вычисленное скорректированное значение прозрачности и применяется к стилю элемента. В зависимости от тайминга, это может привести к тому, что непрозрачность будет меньше 0, а это недопустимо. Проблема устраняется с помощью функции `Math.max`, которая устанавливает минимальное значение 0.

Если прозрачность еще не достигла 0, необходимо вывести больше кадров анимации. Для этого снова вызывается `requestAnimationFrame`, чтобы запланировать следующее выполнение.

В качестве альтернативы этому методу современные браузеры поддерживают API веб-анимации, о котором вы узнаете в *главе 8*. Этот API позволяет вам указывать ключевые кадры с помощью свойств CSS, а браузер сам обрабатывает обновление промежуточных значений для вас.

1.7. Обертывание событий API в промис

Задача

Вы хотите обернуть API, основанный на событиях, для возврата промиса.

Решение

Создайте новый объект `Promise` и зарегистрируйте слушателей событий в его конструкторе. Когда вы получите ожидаемое событие, преобразуйте `Promise` в значение. Аналогичным образом отклоните промис, если произойдет событие ошибки.

Иногда эти действия называются преобразованием функции в промис. В примере 1.9 демонстрируется такое преобразование API `XMLHttpRequest`.

Пример 1.9. Преобразование API XMLHttpRequest в промис

```
/**
 * Отправляет GET-запрос на указанный URL. Возвращает промис, который
 * разрешается в объект JSON, или отказ, если возникла ошибка либо
 * это не объект JSON.
 *
 * @param url - URL-адрес запроса
 * @returns Promis, который разрешается в тело ответа
 */
function loadJSON(url) {
  // Создает новый промис-объект, выполняя асинхронную работу внутри
  // функции конструктора.
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
```

```
    // Если запрос успешный, анализирует JSON-ответ и
    // разрешает промис в результирующий объект.
    request.addEventListener('load', event => {
      // Оборачивает вызов JSON.parse в блок try/catch на случай
      // ответа не в формате JSON.
      try {
        resolve(JSON.parse(event.target.responseText));
      } catch (error) {
        // Произошла ошибка при анализе тела ответа.
        // Отказ от промиса с указанием этой ошибки.
        reject(error);
      }
    });

    // Если запрос неудачен, отказ от промиса с указанием ошибки.
    request.addEventListener('error', error => {
      reject(error);
    });

    // Указать URL-адрес и отправить запрос.
    request.open('GET', url);
    request.send();
  });
}
```

В примере 1.10 показано, как использовать преобразованную в промис функцию `loadJSON`.

Пример 1.10. Использование функции-помощника loadJSON

```
// Используем .then
loadJSON('/api/users/1').then(user => {
  console.log('Got user:', user);
})

// Используем await
const user = await loadJSON('/api/users/1');
console.log('Got user:', user);
```

Обсуждение

Вы создаете промис, вызывая *функцию конструктора* `Promise` с помощью оператора `new`. Эта функция получает два аргумента — функции `resolve` и `reject`.

Функции разрешения (`resolve`) и отклонения (`reject`) предоставляются движком JavaScript. В конструкторе `Promise` вы выполняете асинхронную работу и отслеживаете события. Когда вызывается функция `resolve`, промис немедленно преобразуется в это значение. Вызов функции `reject` работает таким же образом — она отклоняет промис с ошибкой.

Создание собственного промиса может помочь в подобных ситуациях, но в целом обычно не нужно создавать их вручную. Если API уже возвращает промис, вам не нужно включать конструкции создания и возврата в собственный промис — просто используйте автоматический возврат напрямую.

ГЛАВА 2

Простое сохранение данных с помощью Web Storage API

2.0. Введение

Web Storage API сохраняет простые данные локально, в браузере пользователя. Вы можете получить их позже, даже после закрытия и повторного открытия браузера.

Этот API имеет интерфейс хранилища `Storage`, который обеспечивает сохранность данных и доступ к ним. Вы не создаете экземпляры `Storage` напрямую; есть два глобальных экземпляра: `window.localStorage` и `window.sessionStorage`. Единственное различие между ними заключается в том, как долго они сохраняют данные.

Данные `sessionStorage` связаны с определенной сессией браузера. Этот экземпляр сохраняет данные, если страница перезагружается, но при закрытии браузера данные полностью теряются. Разные вкладки из одного и того же источника не передают друг другу сохраненные данные.

С другой стороны, `localStorage` использует единое пространство для хранения данных всех вкладок и сеансов для одного и того же источника. Браузер сохраняет эти данные даже после того, как вы его закроете. В целом хранилище сеансов — хороший выбор, если вы хотите сохранить что-то недолговечное или конфиденциальное, что вы бы хотели уничтожить после закрытия браузера.

В обоих случаях пространство для хранения зависит от конкретного источника данных.

Что такое источник (origin)?

Источник страницы — это строка, объединяющая протокол (http или https), хост и порт URL-адреса. Например, URL-адреса **https://example.com/path/to/index.html** и **https://example.com/profile/index.html** имеют один и тот же источник — **https://example.com**.

Получение и установка элементов

Web Storage может хранить только строковые значения. У каждого значения есть ключ, который вы можете использовать для его поиска. API прост.

`GetItem(key)`

Возвращает строку, привязанную к ключу *key*, или `null`, если ключ не существует.

`setItem(key, value)`

Сохраняет строковое значение *value* в соответствии с заданным ключом *key*. Если ключ уже существует, значение переписывается.

`clear()`

Удаляет все сохраненные данные текущего источника.

Недостатки

Web Storage может быть действительно полезным, но у него есть несколько недостатков.

Ограничения по типу данных.

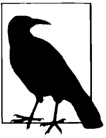
Web Storage может хранить только строковые данные. Вы можете хранить простые объекты, но не напрямую — вам нужно будет преобразовать их в объектную нотацию JavaScript (JSON).

Ограничения по размеру.

Каждый источник имеет ограниченный объем доступного пространства для хранения. В большинстве браузеров он составляет 5 Мбайт. Если хранилище источника заполнится, браузер выдаст исключение при попытке добавить дополнительные данные.

Соображения безопасности.

Несмотря на то что браузер хранит данные каждого источника отдельно, он по-прежнему уязвим для атак с использованием кроссайтовых сценариев (cross-site scripting, XSS). С помощью XSS-атаки злоумышленник может внедрить код, который украдет локально сохраненные данные. Поэтому подумайте о том, стоит ли так хранить конфиденциальные данные.



Все примеры в этой главе используют *локальное* хранилище, то же относится и к сохранению *сессий*. Оба объекта сохраняются при помощи одного `Storage`-интерфейса.

2.1. Проверка поддержки Web Storage

Задача

Прежде чем использовать локальное хранилище, вы хотите проверить его доступность, чтобы избежать сбоя в работе вашего приложения. Вы также хотите разобраться с ситуацией, когда локальное хранилище доступно, но заблокировано пользовательскими настройками.

Решение

Проверьте глобальный объект `window` на наличие свойства `localStorage` с целью убедиться, что браузер поддерживает локальное хранилище. Если проверка прошла успешно, локальное хранилище доступно (пример 2.1).

Пример 2.1. Проверка доступности локального хранилища

```
/**
 * Определяет доступность локального хранилища.
 * @returns true, если браузер может использовать локальное хранилище,
 *          и false в противном случае.
 */
function isLocalStorageAvailable() {
  try {
    // Локальное хранилище доступно, если свойство существует.
    return typeof window.localStorage !== 'undefined';
  } catch (error) {
    // Если window.localStorage существует, но пользователь блокирует
    // локальное хранилище, попытка прочитать свойство выбросит исключение.
    // Если это произойдет, считаем локальное хранилище недоступным.
  return false;
  }
}
```

Обсуждение

Функция в примере 2.1 обрабатывает оба случая: поддерживается ли локальное хранилище вообще и не заблокировано ли оно пользовательскими настройками.

Она проверяет, не является ли значение `window.localStorage` неопределенным. Если проверка пройдена, это означает, что браузер поддерживает локальное хранилище. Если пользователь заблокировал локальное хранилище, простое обращение к `window.localStorage` выдает исключение с сообщением о том, что доступ запрещен.

Поместив проверку свойства в блок try/catch, вы также можете справиться с этим сценарием. При перехвате исключения можно считать, что локальное хранилище недоступно, т. к. возвращается значение false.

2.2. Сохранение строковых данных

Задача

Вы хотите сохранить строковое значение в локальном хранилище и прочитать его позже.

Решение

Используйте localStorage.GetItem и localStorage.setItem для чтения и записи данных. В примере 2.2 показано, как мы можем использовать локальное хранилище для запоминания значения при определении цвета.

Пример 2.2. Сохранение данных в локальном хранилище

```
// Ссылка на входной элемент выбора цвета.
const colorPicker = document.querySelector('#colorPicker');

// Загрузить сохраненный цвет, если он существует, и установить его в colorPicker.
const storedValue = localStorage.getItem('savedColor');
if (storedValue) {
  console.log('Found saved color:', storedValue);
  colorPicker.value = storedValue;
}

// Обновить сохраненный цвет, если он изменился.
colorPicker.addEventListener('change', event => {
  localStorage.setItem('savedColor', event.target.value);
  console.log('Saving new color:', colorPicker.value);
});
```

Обсуждение

При первой загрузке страницы локальное хранилище проверяется на наличие ранее сохраненного цвета. Если вы вызываете getItem с несуществующим ключом, возвращается значение null. Возвращаемое значение задается в определителе цвета только в том случае, если оно не является нулевым или пустым.

Когда значение цвета меняется, обработчик событий сохраняет новое значение в локальном хранилище. Если цвет уже имеется в хранилище, он перезаписывается.

2.3. Сохранение простых объектов

Задача

У вас есть объект JavaScript, например, профиль пользователя, который вы хотите сохранить в локальном хранилище. Вы не можете сделать это напрямую, поскольку локальное хранилище поддерживает только строковые значения.

Решение

Используйте `JSON.stringify` для преобразования объекта в строку JSON перед сохранением. При последующей загрузке значения используйте `JSON.parse`, чтобы преобразовать его обратно в объект, как показано в примере 2.3.

Пример 2.3. Использование JSON.parse и JSON.stringify

```
/**
 * Сериализует объект профиля пользователя в JSON-строку и сохраняет
 * в локальном хранилище.
 * @param userProfile - объект профиля для сохранения
 */
function saveProfile(userProfile) {
  localStorage.setItem('userProfile', JSON.stringify(userProfile));
}

/**
 * Загружает профиль пользователя из локального хранилища и
 * десериализует JSON обратно в объект.
 * Если профиль не существует, возвращается пустой объект.
 * @returns сохраненный профиль пользователя или пустой объект.
 */
function loadProfile() {
  // Если профиль не существует, возвращает null. В этом случае используется
  // пустой объект как значение по умолчанию.
  return JSON.parse(localStorage.getItem('userProfile')) || {};
}
```

Обсуждение

Передача объекта профиля непосредственно в `localStorage.setItem` не даст желаемого эффекта, как показано в примере 2.4.

Пример 2.4. Попытка сохранить массив

```
const userProfile = {
  firstName: 'Ava',
  lastName: 'Johnson'
};

localStorage.setItem('userProfile', userProfile);

// Выводит [object Object]
console.log(localStorage.getItem('userProfile'));
```

Сохраненное значение равно `[object Object]`. Это результат вызова функции `toString` для объекта профиля.

`JSON.stringify` принимает объект и возвращает JSON-строку, представляющую объект. Передача объекта профиля пользователя в `JSON.stringify` приводит к получению нижеприведенной JSON-строки:

```
{
  "firstName": "Ava",
  "lastName": "Johnson"
}
```

Этот подход работает для таких объектов, как профиль пользователя, но спецификация JSON ограничивает разновидность структур, которые могут быть преобразованы в строку. Вообще, это объекты, массивы, строки, числа, логические значения и `null`. Другие значения, такие как экземпляры классов или функции, не могут быть сериализованы таким образом.

2.4. Сохранение сложных объектов

Задача

Вы хотите сохранить в локальном хранилище объект, который нельзя напрямую преобразовать в строку JSON. Например, объект профиля пользователя может содержать объект даты, указывающий, когда он был обновлен в последний раз.

Решение

Используйте функции replacer и reviver совместно с JSON.stringify и JSON.parse для обеспечения сериализации сложных данных.

Рассмотрим следующий объект профиля:

```
const userProfile = {
  firstName: 'Ava',
  lastName: 'Johnson',

  // Это дата представляет собой June 2, 2025.
  // Месяцы начинаются с 0, но дни начинаются с 1.
  lastUpdated: new Date(2025, 5, 2);
}
```

Если вы сериализуете этот объект с помощью JSON.stringify, результирующая строка будет содержать последнюю полученную дату в виде строки даты в формате ISO (пример 2.5).

Пример 2.5. Попытка сериализовать объект с помощью объекта Date

```
const json = JSON.stringify(userProfile);
```

Результирующая строка JSON выглядит следующим образом:

```
{
  "firstName": "Ava",
  "lastName": "Johnson",
  "lastUpdated": '2025-06-02T04:00:00.000Z'
}
```

Теперь у вас есть строка JSON, которую вы можете сохранить в локальном хранилище. Однако, если вы вызовете JSON.parse с этой строкой JSON, результирующий объект будет немного отличаться от исходного. Свойство lastUpdated по-прежнему является строкой, а не датой, потому что JSON.parse не знает, что это должен быть объект Date.

Для того чтобы справиться с такими ситуациями, JSON.stringify и JSON.parse поддерживают специальные функции, называемые replacer и reviver соответственно. Эти функции предоставляют пользовательскую логику для преобразования непримитивных значений в JSON и из него.

Сериализация с помощью функции *replacer*

Аргумент `replacer` в `JSON.stringify` может работать несколькими различными способами. В MDN есть подробная документация по функции `replacer` (**https://oreil.ly/H56TM**).

Функция `replacer` принимает два аргумента: ключ и значение (пример 2.6). `JSON.stringify` сначала вызывает эту функцию с пустой строкой в качестве ключа и преобразуемой строкой в качестве значения. Вы можете преобразовать поле `lastUpdated` здесь в сериализуемое представление объекта `Date`, вызвав функцию `getTime()`, которая выдает дату в виде количества миллисекунд, прошедших с момента начала эпохи (полночь по Гринвичу 1 января, 1970).

Пример 2.6. Функция replacer

```
function replacer(key, value) {
  if (key === '') {
    // Первый вызов replacer, "value" - это сам объект.
    // Возвращает все свойства объекта и преобразует lastUpdated.
    // При этом используется синтаксис распространения объекта,
    // чтобы создать копию "value" перед добавлением свойства lastUpdated.
    return {
      ...value,
      lastUpdated: value.lastUpdated.getTime()
    };
  }

  // После первоначального преобразования replacer вызывается один раз
  // для каждой пары key/value. Больше никаких замен не требуется.
  return value;
}
```

Вы можете передать эту функцию `replacer` в `JSON.stringify`, чтобы преобразовать объект в формат JSON, как показано в примере 2.7.

Пример 2.7. Преобразование в строку с помощью replacer

```
const json = JSON.stringify(userProfile, replacer);
```

Это генерирует следующую строку JSON:

```
{
  "firstName": "Ava",
```

```
  "lastName": "Johnson",
  "lastUpdated": 1748836800000
}
```

Число в свойстве `lastUpdated` — это временна́я метка для 2 июня 2025 года.

Десериализация с помощью функции *reviver*

Позже, когда вы передадите эту строку JSON в `JSON.parse`, свойство `lastUpdated` останется в виде числа (временно́й метки). Вы можете использовать функцию `reviver` для преобразования этого сериализованного числового значения обратно в объект `Date`.

`JSON.parse` вызывает функцию `reviver` для каждого свойства JSON-строки. Для каждого ключа значение, возвращаемое функцией, является значением, заданным в конечном объекте (пример 2.8).

Пример 2.8. Функция reviver

```
function reviver(key, value) {
  // JSON.parse вызывает reviver один раз для каждой пары key/value.
  // Отслеживаем ключ lastUpdated.
  // Продолжаем, если это действительно значение lastUpdated.
  if (key === 'lastUpdated' && value) {
    // Здесь значением является временна́я метка. Вы можете передать это значение
    // конструктору Date, чтобы создать объект данных,
    // ссылающийся на соответствующее время.
    return new Date(value);
  }

  // Остальные значения остаются без изменений.
  return value;
}
```

Для того чтобы использовать `reviver`, передайте его в качестве второго аргумента в `JSON.parse`, как показано в примере 2.9.

Пример 2.9. Парсинг с помощью reviver

```
const object = JSON.parse(userProfile, reviver);
```

Этот код возвращает объект, который эквивалентен объекту профиля пользователя, с которого мы начинали:

```
{
  firstName: 'Ava',
  lastName: 'Johnson',
  lastUpdated: [Date object representing June 2, 2025]
}
```

Обсуждение

С помощью этого надежного метода преобразования объекта в формат JSON и обратно, который сохраняет свойство `Date` неизменным, вы можете сохранить значения в локальном хранилище.

Приведенный здесь подход — это всего лишь один из способов работы с функцией `replacer`. Вместо функции `replacer` вы также можете определить функцию `toJSON` для преобразуемой строки. В сочетании с фабричной функцией функция `replacer` не потребуется.

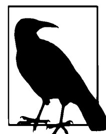
> **Фабричные функции**
>
> В примере 2.10 используется *фабричная функция* для создания объектов профиля пользователя. Она принимает несколько аргументов и возвращает новый объект, содержащий данные, основанные на этих аргументах. Фабричная функция аналогична функции конструктора класса. Основное отличие заключается в том, что вы используете функцию конструктора с оператором `new`, а фабричная функция вызывается напрямую, как и любая другая функция.

Пример 2.10. Использование фабричной функции, добавляющей функцию toJSON

```
/**
 * Фабричная функция для создания объекта профиля пользователя,
 * со свойством lastUpdated, равным today, и методом toJSON
 *
 * @param firstName - имя пользователя
 * @param lastName - фамилия пользователя
 */
function createUser(firstName, lastName) {
  return {
    firstName,
    lastName,
    lastUpdated: new Date(),
    toJSON() {
      return {
```

```
        firstName: this.firstName,
        lastName: this.lastName,
        lastUpdated: this.lastUpdated.getTime();
      }
    }
  }
}

const userProfile = createUser('Ava', 'Johnson');
```

Вызов `JSON.stringify` с объектом в примере 2.10 возвращает ту же строку JSON, что и раньше, с соответствующим преобразованием `lastUpdated` во временну́ю метку.



Не существует подобного механизма для преобразования строки обратно в объект с помощью `JSON.parse`. Если вы используете подход `toJSON`, показанный здесь, вам все равно потребуется написать функцию `reviver`, чтобы правильно десериализовать строку профиля пользователя.

Поскольку функции не могут быть сериализованы, результирующая строка JSON не будет содержать свойство `toJSON`. Какой бы метод вы ни выбрали, результирующий JSON будет одним и тем же.

2.5. Отслеживание изменений в хранилище

Задача

Вы хотите получать уведомление, когда другая вкладка из того же источника вносит изменения в локальное хранилище.

Решение

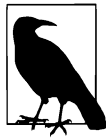
Прослушайте событие `storage` в объекте `window`. Оно срабатывает, когда другие вкладки или сеансы, открытые в том же браузере, вносят изменения в какие-либо данные в локальном хранилище (пример 2.11).

Пример 2.11. Прослушивание изменений в хранилище с другой вкладки

```
// Прослушивание события 'storage'. Если другая вкладка меняет
// элемент 'savedColor', обновите colorPicker этой страницы новым значением.
window.addEventListener('storage', event => {
  if (event.key === 'savedColor') {
```

```
    console.log('New color was chosen in another tab:', event.newValue);
    colorPicker.value = event.newValue;
  }
});
```

Вспомните выбор цвета из рецепта 2.2. Если пользователь открывает несколько вкладок и изменяет цвет на другой вкладке, вы можете получить уведомление и обновить локальную копию данных в памяти, чтобы все было синхронизировано.

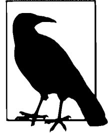


Событие `storage` *не* запускается на вкладке или странице, на которой было произведено изменение хранилища. Оно предназначено для отслеживания изменений, внесенных *другими* страницами в локальное хранилище.

Событие `storage` указывает, какой ключ был изменен и каково его новое значение. Оно также содержит старое значение на случай, если оно понадобится вам для сравнения.

Обсуждение

Основным вариантом использования события `storage` является синхронизация нескольких сеансов друг с другом в режиме реального времени.



Событие `storage` запускается только для других вкладок и сеансов в том же браузере на том же устройстве.

Даже если вы не отслеживаете событие `storage`, все сессии одного источника по-прежнему используют одни и те же данные локального хранилища. Если вы вызовете `localStorage.getItem` в любой момент, вы все равно получите последнее значение. Событие `storage` просто предоставляет уведомление в режиме реального времени о том, когда происходит такое изменение, чтобы приложение могло обновить локальные данные.

2.6. Поиск всех известных ключей

Задача

Вы хотите знать все ключи, которые в данный момент находятся в локальном хранилище для текущего источника.

Решение

Используйте свойство length с функцией key для создания списка всех известных ключей. Объекты хранилища не имеют функции прямого возврата списка ключей, но вы можете создать такой список, используя следующее:

- свойство length возвращает количество ключей;
- функция key, которой задан индекс, возвращает ключ с этим индексом.

Вы можете объединить их циклом for, чтобы создать массив всех ключей, как показано в примере 2.12.

Пример 2.12. Составление списка ключей

```
/**
 * Создает массив всех ключей, найденных в локальном хранилище.
 * @returns массив ключей
 */
function getAllKeys() {
  const keys = [];

  for (let i = 0; i < localStorage.length; i++) {
    keys.push(localStorage.key(i));
  }

  return keys;
}
```

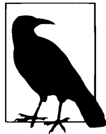
Обсуждение

Вы можете комбинировать свойство length и функцию key для выполнения других типов запросов. Это может быть, например, функция, которая принимает массив ключей и возвращает объект, содержащий только эти пары "ключ/значение" (пример 2.13).

Пример 2.13. Запрос подмножества пар "ключ/значение"

```
function getAll(keys) {
  const results = {};

  // Проверить каждый ключ в локальном хранилище.
  for (let i = 0; i < localStorage.length; i++) {

    // Получить i-й ключ. Если в массиве ключей есть этот ключ,
    // добавить сам ключ и его значение в результирующий объект.
```

```
    const key = localStorage.key(i);
    if (keys.includes(key)) {
      results[key] = localStorage.getItem(key);
    }
  }

  // Результат теперь содержит все пары key/value,
  // которые существуют в локальном хранилище.
  return results;
}
```



Порядок расположения ключей, указанный в функции key, может отличаться в разных браузерах.

2.7. Удаление данных

Задача

Вы хотите удалить некоторые или все данные из локального хранилища.

Решение

Используйте методы removeItem и clear. Для того чтобы удалить определенную пару "ключ/значение" из локального хранилища, вызовите localStorage.removeItem со значением ключа (пример 2.14).

Пример 2.14. Удаление элемента из локального хранилища

```
// Это безопасная операция. Если ключ не существует,
// исключение не генерируется.
localStorage.removeItem('my-key');
```

Вызовите localStorage.clear, чтобы удалить все данные из локального хранилища для текущего источника, как показано в примере 2.15.

Пример 2.15. Удаление всех элементов из локального хранилища

```
localStorage.clear();
```

Обсуждение

Браузеры ограничивают объем данных, которые вы можете хранить в Web Storage. Обычно лимит составляет около 5 Мбайт. Для того чтобы избежать нехватки места и возникновения ошибки, вам следует удалять ненужные элементы. В зависимости от того, для чего вы используете Web Storage, вы также можете предоставить пользователям возможность самим удалять сохраненные данные. Представьте средство выбора эмодзи, которое сохраняет недавно выбранные эмодзи в локальном хранилище. Вы можете добавить кнопку **Clear Recents** (Очистить последние), которая удаляет эти элементы.

ГЛАВА 3

URL-адреса и маршрутизация

3.0. Введение

Большинство веб-страниц и приложений так или иначе работают с URL-адресами. Это может быть такое действие, как создание ссылки с определенными параметрами запроса или маршрутизация на основе URL-адреса в одностраничном приложении (single-page application, SPA).

URL-адрес — это просто строка, соответствующая некоторым правилам синтаксиса, определенным в RFC 3986 "Унифицированный идентификатор ресурса (URI): общий синтаксис" (**https://oreil.ly/SUziR**). Существует несколько составных частей URL-адреса, которые вам, возможно, потребуется проанализировать или обработать. Выполнение этих операций с помощью таких методов, как регулярные выражения или конкатенация строк, не всегда надежно.

Сегодня браузеры поддерживают URL API. Этот API предоставляет конструктор URL, который может создавать, выводить и манипулировать URL-адресами. Сначала этот API был несколько ограничен, но в более поздних обновлениях были добавлены утилиты, такие как интерфейс `URLSearchParams`, которые упростили создание и чтение строк запроса.

Части URL-адреса

Когда вы вызываете конструктор URL-адреса со строкой, представляющей допустимый URL-адрес, результирующий объект содержит свойства, представляющие различные составные компоненты URL-адреса. На рис. 3.1 показаны наиболее часто используемые компоненты.

Протокол (1).

Для веб-адресов обычно используется `http:` или `https:` (обратите внимание, что двоеточие включено, а слеши отсутствуют). Возможны и другие протоколы, такие как `file:` (для локального файла, не размещенного на сервере) или `ftp:` (для ресурса на FTP-сервере).

Имя хоста (2).

Имя домена или хоста (`example.com`).

Путь (3).

Путь к ресурсу относительно корневого каталога со слешем в начале (`/admin/login`).

Поисковые данные (4).

Любые параметры запроса. Включают символ ? (`?username=sysadmin`).

Рис. 3.1. Пример URL-адреса с выделенными составными частями

Имеются и другие части URL-адреса.

Хеш (hash).

URL-адрес может содержать хеш (включая символ хеша #). Иногда это используется для внутренней навигации в старых одностраничных приложениях. Для URL-адреса **https://example.com/app#prfile** значением хеша будет `#profile`.

Хост (host).

Аналогично `hostname`, но также включает номер порта (если указан), например `localhost:8443`.

Источник (origin).

Источник URL-адреса. Обычно включает протокол, имя хоста и порт (если они указаны).

Вы можете получить строку URL целиком, вызвав для нее метод `toString` или обратившись к ее свойству `href`.

Если конструктору URL передается недопустимая строка URL-адреса, он генерирует исключение.

3.1. Определение относительного URL-адреса

Задача

У вас есть частичный или относительный URL, например `/api/users`, который вы хотите преобразовать в полный абсолютный URL, например **https://example.com/api/users**.

Решение

Создайте объект URL, передав относительный URL и желаемый базовый URL, как показано в примере 3.1.

Пример 3.1. Создание относительного URL-адреса

```
/**
 * Заданы относительный путь и базовый URL-адрес.
 * Результат - полный абсолютный URL-адрес.
 * @param relativePath - относительный путь для URL
 * @param baseUrl - допустимый URL-адрес в качестве базового
 */
function resolveUrl(relativePath, baseUrl) {
  return new URL(relativePath, baseUrl).href;
}

// https://example.com/api/users
console.log(resolveUrl('/api/users', 'https://example.com'));
```

Без второго аргумента конструктор URL выдал бы ошибку, поскольку `/api/users` не является допустимым URL-адресом. Второй аргумент является базовым для построения нового URL-адреса. При построении URL-адреса предполагается, что указанный путь относится к базовому URL-адресу.

Обсуждение

Вторым аргументом должен быть допустимый URL. Для создания окончательного URL-адреса применяются типичные правила для допустимого в зависимости от первого аргумента относительного URL-адреса.

Если первый аргумент начинается с косой черты, то путь к базовому URL-адресу игнорируется, и новый URL-адрес указывается относительно корневого каталога базового URL-адреса:

```
// https://example.com/api/v1/users
console.log(resolveUrl('/api/v1/users', 'https://example.com'));

// https://example.com/api/v1/users
// Заметьте, что /api/v2 отбрасывается из-за слеша в /api/v1/users
console.log(resolveUrl('/api/v1/users', 'https://example.com/api/v2'));
```

В противном случае URL-адрес вычисляется относительно базового URL-адреса:

```
// https://example.com/api/v1/users
console.log(resolveUrl('../v1/users/', 'https://example.com/api/v2'));

// https://example.com/api/v1/users
console.log(resolveUrl('users', 'https://example.com/api/v1/groups'));
```

Если первый аргумент сам по себе является допустимым URL-адресом, базовый URL-адрес игнорируется.

Если второй аргумент конструктора не является строкой, для него вызывается метод `toString` и используется результирующая строка. Это означает, что вы можете передавать другие объекты URL или даже другие объекты, похожие на URL. Вы даже можете передавать `window.location` (объект `Location`, свойства которого аналогичны свойствам URL) для создания нового URL в текущем источнике (пример 3.2).

Пример 3.2. Создание относительного URL-адреса в том же источнике

```
const usersApiUrl = new URL('/api/users', window.location);
```

3.2. Удаление параметров запроса из URL-адреса

Задача

Вы хотите удалить все параметры запроса из URL-адреса.

Решение

Создайте объект URL и присвойте его свойству `search` значение пустой строки, как показано в примере 3.3.

Пример 3.3. Удаление параметров запроса URL

```
/**
 * Удаляет все параметры из URL-адреса.
 *
 * @param inputUrl - строка URL-адреса, содержащая параметры запроса
 * @returns новую строку URL-адреса без параметров запроса
 */
function removeAllQueryParameters(inputUrl) {
  const url = new URL(inputUrl);
  url.search = '';
```

```
  return url.toString();
}

// Результат 'https://example.com/api/users'
removeAllQueryParams('https://example.com/api/users?user=sysadmin&q=user');
```

Обсуждение

Параметры запроса в URL-адресе представлены двумя способами: с помощью свойства search и свойства searchParams.

Значение свойства search — это единая строка, содержащая все параметры запроса вместе с начальным символом ?. Если вы хотите удалить всю строку запроса, можете задать для нее значение пустой строки.

Обратите внимание, что для свойства search задано значение пустой строки. Если вы зададите для него значение null, вы получите буквальную строку null в строке запроса (пример 3.4).

Пример 3.4. Некорректная попытка удалить все параметры запроса

```
const url = new URL('https://example.com/api/users?user=sysadmin&q=user');

url.search = null;
console.log(url.toString()); // https://example.com/api/users?null
```

Свойство searchParams является объектом URLSearchParams. Оно содержит методы для просмотра, добавления и удаления параметров запроса. При добавлении параметров запроса оно автоматически обрабатывает символы кодировки. Если вы хотите удалить только один параметр запроса, можете вызвать метод delete для этого объекта, как показано в примере 3.5.

Пример 3.5. Удаление одного параметра запроса

```
/**
 * Удаляет один параметр из URL
 *
 * @param inputUrl - URL-строка, содержащая параметры запроса
 * @param paramName - имя удаляемого параметра
 * @returns новую URL-строку с удаленным параметром
 */
function removeQueryParameter(inputUrl, paramName) {
  const url = new URL(inputUrl);
  url.searchParams.delete(paramName);
  return url.toString();
}
```

```
console.log(
  removeQueryParameter(
    'https://example.com/api/users?user=sysadmin&q=user',
    'q'
  )
); // https://example.com/api/users?user=sysadmin
```

3.3. Добавление параметров запроса к URL-адресу

Задача

У вас есть существующий URL-адрес, в котором, возможно, уже присутствуют некоторые параметры запроса, и вы хотите добавить дополнительные параметры запроса.

Решение

Используйте объект URLSearchParams, доступный через свойство searchParams, чтобы добавить дополнительные параметры (пример 3.6).

Пример 3.6. Добавление дополнительных параметров запроса

```
const url = new URL('https://example.com/api/search?objectType=user');

url.searchParams.append('userRole', 'admin');
url.searchParams.append('userRole', 'user');
url.searchParams.append('name', 'luke');

// Печатает
"https://example.com/api/search?objectType=user&userRole=admin&userRole=user
&name=luke"
console.log(url.toString());
```

Обсуждение

Изначально этот URL-адрес уже содержит параметр запроса (objectType=user). В коде используется свойство searchParams проанализированного URL-адреса для добавления еще нескольких параметров запроса. Добавлены два параметра UserRole. Когда вы используете метод append, он добавляет новые значения и сохраняет существующие. Для того чтобы заменить все параметры с заданным именем новыми значениями, вы можете вместо append использовать метод set.

С новыми параметрами полный URL-адрес теперь будет выглядеть следующим образом:

```
https://example.com/api/search?objectType=user&userRole=admin&userRole=user&name=luke
```

Если вы вызываете append с именем параметра, но без значения, вы получите исключение, как показано в примере 3.7.

Пример 3.7. Попытка вызова append без значения

```
const url = new URL('https://example.com/api/search?objectType=user');

// TypeError: Failed to execute 'append' on 'URLSearchParams':
// требуются два аргумента, а представлен один.
url.searchParams.append('name');
```

Этот метод корректно обрабатывает аргументы других типов. Если он получает значение не строкового типа, то преобразует его в строку (пример 3.8).

Пример 3.8. Добавление не строковых параметров

```
const url = new URL('https://example.com/api/search?objectType=user');

// Результирующий URL имеет следующую строку запроса:
// ?objectType=user&name=null&role=undefined
url.searchParams.append('name', null);
url.searchParams.append('role', undefined);
```

Использование URLSearchParams для добавления параметров запроса автоматически устраняет любые потенциальные проблемы с кодировкой. Если вы добавляете параметр с зарезервированным символом (определенным в RFC 3986), например & или ?, URLSearchParams автоматически кодирует их, чтобы обеспечить допустимый URL. В нем используется *процентное кодирование (percent encoding)*, когда добавляется знак процента, за которым следуют шестнадцатеричные цифры, представляющие соответствующий символ. Например, & становится %26, потому что 0x26 — это шестнадцатеричный код для амперсанда.

Вы можете увидеть эту кодировку в действии, добавив параметр запроса, содержащий несколько зарезервированных символов, как показано в примере 3.9.

Пример 3.9. Кодировка зарезервированных символов в параметре запроса

```
const url = new URL('https://example.com/api/search');

// Придуманный пример строки, демонстрирующей несколько зарезервированных символов
url.searchParams.append('q', 'admin&user?luke');
```

Результирующий URL-адрес приобретает вид:

```
https://example.com/api/search?q=admin%26user%3Fluke
```

URL-адрес содержит %26 вместо & и %3F вместо ?. Эти символы имеют особое значение в URL-адресе. ? указывает на начало строки запроса, а & является разделителем между параметрами.

Как показано в примере 3.6, многократный вызов метода append с одним и тем же ключом добавляет новый параметр запроса с заданным ключом. Когда вы вызываете .append('UserRole', ' user '), он добавляет параметр UserRole=user и сохраняет предыдущий UserRole=admin. У объекта URLSearchParams также есть метод set, который также добавляет параметры запроса, но по-другому. set заменяет все существующие параметры в соответствии с заданным ключом новыми параметрами (пример 3.10). Если вы попытаетесь создать URL-адрес с теми же параметрами, используя set, то получите другой результат.

Пример 3.10. Добавление параметров запроса с помощью set

```
const url = new URL('https://example.com/api/search?objectType=user');

url.searchParams.set('userRole', 'admin');
url.searchParams.set('userRole', 'user');
url.searchParams.set('name', 'luke');
```

Когда вы используете метод set вместо append, второй параметр userRole перезаписывает первый, и в результате получается URL-адрес:

```
https://example.com/api/search?objectType=user&userRole=user&name=luke
```

Обратите внимание, что существует только один параметр userRole — последний, который был добавлен.

3.4. Чтение параметров запроса

Задача

Вы хотите проанализировать параметры запроса и вывести их список в URL-адресе.

Решение

Используйте метод forEach для URLSearchParams, чтобы вывести список ключей и значений (пример 3.11).

Пример 3.11. Чтение параметров запроса

```
/**
 * Принимает URL и возвращает массив параметров запроса
 *
 * @param inputUrl - URL-строка
 * @returns массив объектов - пар "ключ/значение"
 */
function getQueryParameters(inputUrl) {
  // Мы не можем здесь использовать объект, потому что может быть
  // множество параметров с одним и тем же ключом, а мы хотим вернуть все параметры.
  const result = [];

  const url = new URL(inputUrl);

  // Добавляем пару key/value к результирующему массиву.
  url.searchParams.forEach((value, key) => {
    result.push({ key, value });
  });

  // Результат готов!
  return result;
}
```

Обсуждение

При указании параметров запроса в URL-адресе все зарезервированные символы с процентным кодированием декодируются обратно к их исходным значениям (пример 3.12).

Пример 3.12. Использование функции getQueryParameters

```
getQueryParameters('https://example.com/api/search?name=luke%26ben'); ❶
```

❶ Параметр `name` содержит символ амперсанда в процентной нотации (`%26`).

Этот код выводит `name=luke%26ben` с исходным незашифрованным значением:

```
name: luke&ben
```

`forEach` выполняет итерацию по каждой уникальной комбинации пар "ключ/значение". Даже если URL-адрес содержит несколько параметров запроса с одним и тем же ключом, каждая уникальная пара "ключ/значение" выводится отдельно.

3.5. Создание простого маршрутизатора на стороне клиента

Задача

У вас есть одностраничное приложение, и вы хотите добавить маршрутизацию на стороне клиента. Так вы позволите пользователю перемещаться между различными URL-адресами без создания нового сетевого запроса при помощи замены содержимого на стороне клиента.

Решение

Используйте `history.pushState` и событие `popstate` для реализации простого маршрутизатора. Этот простой маршрутизатор отображает содержимое шаблона, когда URL-адрес совпадает с известным маршрутом (пример 3.13).

Пример 3.13. Простой маршрутизатор на стороне клиента

```
// Определения маршрутов. Каждый маршрут имеет путь и содержимое для просмотра.
const routes = [
  { path: '/', content: '<h1>Home</h1>' },
  { path: '/about', content: '<h1>About</h1>' }
];

function navigate(path, pushState = true) {
  // Найти соответствующий маршрут и просмотреть содержимое.
  const route = this.routes.find(route => route.path === path);

  // Будьте осторожны при использовании innerHTML, т. к. это может быть небезопасно.
  document.querySelector('#main').innerHTML = route.content;

  if (pushState) {
    // Изменим URL для соответствия новому маршруту.
    history.pushState({}, '', path);
  }
}
```

Установив этот маршрутизатор, вы можете добавлять ссылки:

```
<a href="/">Home</a>
<a href="/about">About</a>
```

history.pushState и событие popstate

Метод pushState глобального объекта history изменяет текущий URL-адрес без перезагрузки страницы. Он добавляет новый URL-адрес в историю браузера.

Метод принимает три аргумента.

- Первый — объект, содержащий произвольные данные, которые необходимо связать с новым событием в history. Эти данные о состоянии также доступны из события popstate.
- Второй аргумент не используется, но его необходимо указать. Здесь можно использовать пустую строку.
- Наконец, новый URL. Это может быть абсолютный URL-адрес или относительный путь. Если вы используете абсолютный URL-адрес, он должен находиться в том же источнике, что и текущая страница, иначе браузер выдаст исключение.

Каждый вызов pushState создает запись в истории. Всякий раз, когда изменяется текущая запись в истории (обычно с помощью кнопок **Back** и **Forward** в браузере), окно запускает событие popstate.

Когда вы переходите по этим ссылкам, браузер пытается перейти на новую страницу, отправляя запрос на сервер. Скорее всего, это приведет к ошибке 404, а это не то, что вам нужно. Для того чтобы использовать маршрутизацию на стороне клиента, вам необходимо перехватить события щелчков мышью и интегрировать их в маршрутизатор из примера 3.13, как показано в примере 3.14.

Пример 3.14. Добавление обработчиков щелчков мышью для маршрутизации ссылок

```
document.querySelectorAll('a').forEach(link => {
  link.addEventListener('click', event => {
    // Не допустить, чтобы браузер пытался загрузить новый URL с сервера!
    event.preventDefault();
    navigate(link.getAttribute('href'));
  });
});
```

Когда вы нажимаете на одну из этих ссылок, вызов preventDefault останавливает работу браузера по умолчанию (выполнение полностраничной навигации). Вместо этого он использует атрибут href и передает его маршрутизатору на стороне клиента. Если он находит подходящий маршрут, то отображает содержимое для этого маршрута.

Для того чтобы сделать представленный алгоритм полноценным решением, нужен еще один необходимый элемент. Если вы выберете один из этих маршрутов на сто-

роне клиента, а затем нажмете кнопку возврата на предыдущую страницу в браузере, ничего не произойдет, ведь на самом деле страница не перемещается, а просто возвращает предыдущее состояние из маршрутизатора. Для того чтобы справиться с этим сценарием, вам также необходимо прослушать событие popstate браузера и отобразить правильное содержимое, как показано в примере 3.15.

Пример 3.15. Отслеживание события popstate

```
window.addEventListener('popstate', () => {
  navigate(window.location.pathname, false);
});
```

Когда пользователь нажимает кнопку **Back**, браузер запускает событие popstate. Это действие возвращает URL-адрес страницы, и вам просто нужно найти в содержимом маршрут, соответствующий URL-адресу. А вот вызывать метод pushState надобности нет, потому что этот вызов добавляет новое состояние истории, но это, вероятно, вам не нужно, поскольку вы же просто удалили старое состояние истории из стека.

Обсуждение

Этот клиентский маршрутизатор работает, но есть одна проблема. Если вы перейдете по ссылке **About** (О программе), а затем нажмете кнопку **Refresh** (Обновить), браузер отправит новый сетевой запрос, что, вероятно, приведет к ошибке 404. Для того чтобы устранить эту последнюю проблему, сервер должен быть настроен на возврат основного содержимого HTML и JavaScript независимо от пути (path) URL-адреса. При этом загружается код маршрутизатора, который вызывается со значением window.location.pathname. Если все настроено правильно, обработчик маршрута на стороне клиента выполняет и отображает корректное содержимое.

При использовании маршрутизации на стороне клиента переход между страницами может осуществляться быстрее, поскольку нет необходимости обращаться к серверу. Это делает навигацию более плавной и отзывчивой. Но есть и недостатки. Для того чтобы обеспечить быстрое перемещение по странице, вам часто приходится загружать много дополнительного JavaScript-кода, поэтому первоначальная загрузка страницы может быть медленнее.

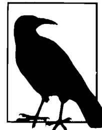
3.6. Сопоставление URL-адресов с шаблонами

Задача

Вы хотите определить набор допустимых URL-адресов, с которыми вы можете сопоставлять свои URL-адреса. Вы также можете извлечь часть пути из URL-адреса. Например, если URL-адрес — **https://example.com/api/users/123/profile**, а вам нужен идентификатор пользователя (123).

Решение

Используйте API URL Pattern, чтобы определить ожидаемый шаблон и извлечь нужную вам часть.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/Eb-k2**).

С помощью этого API вы можете создать объект `URLPattern`, определяющий шаблон, который вы можете использовать для сопоставления URL-адресов (пример 3.16). Он создается с указанием строки, определяющей соответствующий шаблон. Строка может содержать именованные группы, которые извлекаются при сопоставлении со строкой URL. Вы можете получить доступ к извлеченным значениям по их индексу. Эти группы аналогичны группам, получаемым в регулярном выражении.

Пример 3.16. Создание URLPattern

```
const profilePattern = new URLPattern({ pathname: '/api/users/:userId/profile' });
```

В примере 3.16 показан простой шаблон URL-адреса с одной именованной группой `userId`. Перед названием группы ставится двоеточие. Вы можете использовать этот объект шаблона для соответствия URL-адреса и, если они совпадают, извлечения из них идентификатора пользователя. В примере 3.17 рассматриваются некоторые различные URL-адреса и способы их проверки на соответствие шаблону профиля с помощью метода тестирования.

Пример 3.17. Проверка URL-адресов на соответствие шаблону

```
// Шаблон не будет соответствовать только пути pathname;
// это должен быть полноценный URL-адрес.
console.log(profilePattern.test('/api/users/123/profile'));

// Этот URL подходит, потому что pathname соответствует шаблону.
console.log(profilePattern.test('https://example.com/api/users/123/profile'));

// Он также соответствует URL-объектам.
console.log(profilePattern.test(new URL
('https://example.com/api/users/123/profile')));

// Путь pathname должен точно соответствовать, поэтому не подходит.
console.log(profilePattern.test('https://example.com/v1/api/users/123/profile'));
```

profilePattern указывает точное соответствие имени пути, поэтому последний пример 3.17 не сработал. Вы можете определить менее строгую версию, в которой используется подстановочный знак (wildcard — *), поэтому соответствие не обязательно должно быть точным. С помощью этого нового шаблона вы можете сопоставлять имена частичных путей (пример 3.18).

Пример 3.18. Использование подстановочного знака в шаблоне

```
const wildcardProfilePattern = new URLPattern
({ pathname: '/*/api/users/:userId/profile' });

// Здесь совпадение, потому что часть /v1 URL соответствует *.
console.log(wildcardProfilePattern.test
('https://example.com/v1/api/users/123/profile'));
```

Вы можете использовать метод exec шаблона, чтобы получить больше данных о совпадении. Если шаблон совпадает с URL, exec возвращает объект, содержащий все совпадения с частями URL. У каждого вложенного объекта есть свойство input, указывающее, какая часть URL совпадает, и свойство groups, в котором указаны все именованные группы, определенные в шаблоне.

Вы можете использовать exec для извлечения идентификатора пользователя из совпадающих URL-адресов, как показано в примере 3.19.

Пример 3.19. Извлечение идентификатора пользователя

```
const profilePattern = new URLPattern({ pathname: '/api/users/:userId/profile' });

const match = profilePattern.exec('https://example.com/api/users/123/profile');
console.log(match.pathname.input); // '/api/users/123/profile'
console.log(match.pathname.groups.userId); // '123'
```

Обсуждение

Хотя у этого API пока нет полной поддержки браузерами, он очень гибкий. Вы можете определять шаблоны для любых частей URL, сопоставляя свойство input и извлекая группы.

ГЛАВА 4

Сетевые запросы

4.0. Введение

Сегодня непросто найти веб-приложение, которое не отправляло бы никаких сетевых сообщений. С самого зарождения Web 2.0 и нового подхода, известного как Ajax (Asynchronous JavaScript and XML — асинхронный JavaScript и XML), веб-приложения отправляют асинхронные запросы для получения новых данных без перезагрузки всей страницы. XMLHttpRequest API положил начало новой эре интерактивных JavaScript-приложений. Несмотря на название, XMLHttpRequest (или XHR, как его иногда называют) также может работать с JSON и с его помощью формировать полезные данные.

XMLHttpRequest изменил правила игры, но работать с этим API оказалось непросто. Со временем сторонние библиотеки, такие как Axios и jQuery, добавили более оптимизированные API, которые дополняли базовый XHR API.

В 2015 году новый API под названием Fetch, основанный на промисах, стал новым стандартом, и разработчики постепенно начали включать его поддержку в браузеры. На сегодняшний день Fetch является стандартным способом выполнения асинхронных запросов из ваших веб-приложений.

В этой главе рассматриваются XHR и Fetch, а также некоторые другие API для сетевого взаимодействия.

Beacon.

Простой односторонний POST-запрос, идеально подходящий для отправки аналитических данных.

События, отправляемые сервером (server-sent events, SSE).

Одностороннее постоянное соединение с сервером для получения событий в режиме реального времени.

WebSocket.

Двустороннее постоянное соединение для двунаправленной связи.

4.1. Отправка запроса с помощью XMLHttpRequest

Задача

Вы желаете отправить GET-запрос в общедоступный API и хотите поддерживать старые браузеры, которые не реализуют Fetch API.

Решение

Используйте API XMLHttpRequest. XMLHttpRequest — это асинхронный, основанный на событиях API для создания сетевых запросов. Использование в общих чертах XMLHttpRequest заключается в следующем:

1. Создайте новый объект `XMLHttpRequest`.
2. Добавьте прослушиватель для события `load`, который получает данные ответа.
3. Вызовите `open` для формирования запроса, передав HTTP-метод и URL-адрес.
4. Наконец, вызовите `send` для инициализации и отправки HTTP-запроса.

В примере 4.1 показан простой образец работы с данными JSON с использованием XHR.

Пример 4.1. Отправка GET-запроса с помощью XMLHttpRequest

```
/**
 * Загружает данные из URL /api/users и выводит их в консоль.
*/
function getUsers() {
  const request = new XMLHttpRequest();

  request.addEventListener('load', event => {
    // Целью события является сам XHR; он содержит свойство responseText,
    // которое мы можем использовать для создания объекта JavaScript из текста JSON.
    const users = JSON.parse(event.target.responseText);
    console.log('Got users:', users);
  });

  // Обрабатывайте любые возможные ошибки запроса. Здесь обрабатываются
  // только сетевые ошибки. Если запрос возвращает статус ошибки,
  // например 404, все равно срабатывает событие 'load', в котором вы
  // можете проверить код состояния.
  request.addEventListener('error', err => {
    console.log('Error!', err);
  });
```

```
  request.open('GET', '/api/users');
  request.send();
}
```

Обсуждение

XMLHttpRequest API основан на событиях. При получении ответа на запрос запускается событие загрузки `load`. В примере 4.1 обработчик события загрузки передает необработанный текст ответа в `JSON.parse`. Он ожидает, что текст ответа будет в формате JSON, и использует `JSON.parse` для преобразования строки JSON в объект.

Если при загрузке данных возникает ошибка, запускается событие `error`. Это позволяет обрабатывать ошибки подключения или работы сети, но код состояния HTTP, который считается "ошибкой", например 404 или 500, *не* вызывает ошибки. Вместо этого он также запускает событие `load`.

Для защиты от таких ошибок вам необходимо проверить свойство `status` ответа, чтобы определить, не возникла ли такая ситуация с ошибкой. Доступ к нему можно получить, обратившись к `event.target.status`.

Fetch поддерживается уже давно, поэтому, если вам не нужно поддерживать действительно старые браузеры, вам, скорее всего, не понадобится использовать XMLHttpRequest. Бóльшую часть времени, если не всё, вы будете использовать Fetch API.

4.2. Отправка GET-запроса с помощью Fetch API

Задача

Вы хотите отправить GET-запрос к общедоступному API из современного браузера.

Решение

Используйте Fetch API. Fetch — это более новый API запросов, который использует промисы. Он очень гибкий и может отправлять все виды данных, но в примере 4.2 отправляется простой запрос GET в API.

Пример 4.2. Отправка GET-запроса с помощью Fetch API

```
/**
 * Загружает пользователей, вызывая /api/users API, и анализирует JSON-ответ.
 * @returns промис, разрешаемый в массив пользователей,
 *          который возвращается с помощью API
 */
```

```
function loadUsers() {
  // Создаем запрос.
  return fetch('/api/users')
    // Преобразуем тело ответа в объект.
    .then(response => response.json())
    // Обрабатываем ошибки, включая ошибки сети и ошибки в строках JSON.
    .catch(error => console.error('Unable to fetch:', error.message));
}

loadUsers().then(users => {
  console.log('Got users:', users);
});
```

Обсуждение

API Fetch более лаконичен. Он возвращает `Promise`, который преобразуется в объект, представляющий HTTP-ответ. Объект `response` содержит такие данные, как код состояния, заголовки и текст.

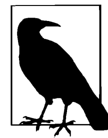
Для того чтобы получить текст ответа в формате JSON, вам нужно вызвать метод `json` ответа. Этот метод считывает текст из потока и возвращает `Promise`, который преобразуется в текст JSON, проанализированный как объект. Если текст ответа содержит недопустимый в JSON формат, `Promise` отклоняется.

В ответе также есть методы для чтения текста в других форматах, таких как `FormData` или обычная текстовая строка.

Поскольку Fetch работает с промисами, вы также можете использовать `await`, как показано в примере 4.3.

Пример 4.3. Использование Fetch с async/await

```
async function loadUsers() {
  try {
    const response = await fetch('/api/users');
    return response.json();
  } catch (error) {
    console.error('Error loading users:', error);
  }
}

async function printUsers() {
  const users = await loadUsers();
  console.log('Got users:', users);
}
```



Помните, чтобы использовать await в функции, эта функция должна содержать ключевое слово async.

4.3. Отправка POST-запроса с помощью Fetch API

Задача

Вы хотите отправить POST-запрос в API, который ожидает запрос в виде текста в формате JSON.

Решение

Используйте Fetch API, указав метод (POST), текст в формате JSON и тип содержимого (пример 4.4).

Пример 4.4. Отправка данных в формате JSON с помощью POST-запроса Fetch API

```
/**
 * Создает нового пользователя, посылая POST-запрос к /api/users.
 * @param firstName - имя пользователя
 * @param lastName - фамилия пользователя
 * @param department - отдел пользователя
 * @returns Promise, который разрешается в тело ответа API
 */
function createUser(firstName, lastName, department) {
  return fetch('/api/users', {
    method: 'POST',
    body: JSON.stringify({ firstName, lastName, department }),
    headers: {
      'Content-Type': 'application/json'
    }
  })
    .then(response => response.json());
}

createUser('John', 'Doe', 'Engineering')
  .then(() => console.log('Created user!'))
  .catch(error => console.error('Error creating user:', error));
```

Обсуждение

В примере 4.4 отправляются некоторые данные в формате JSON в POST-запросе. Вызов `JSON.stringify` для объекта `user` преобразует данные в *строку* JSON, которая требуется для отправки в виде тела запроса с помощью `fetch`. Вам также необходимо задать заголовок `Content-Type`, чтобы сервер знал, как интерпретировать тело запроса.

Fetch также позволяет отправлять другие типы контента в качестве тела запроса. В примере 4.5 показано, как вы могли бы отправить POST-запрос с некоторыми данными формы.

Пример 4.5. Отправка данных формы в POST-запросе

```
fetch('/login', {
  method: 'POST',
  body: 'username=sysadmin&password=password',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded;charset=UTF-8'
  }
})
  .then(response => response.json())
  .then(data => console.log('Logged in!', data))
  .catch(error => console.error('Request failed:', error));
```

4.4. Загрузка файла с помощью Fetch API

Задача

Вы хотите загрузить данные файла с помощью POST-запроса, используя Fetch API.

Решение

Используйте элемент `<input type="file">` и отправьте содержимое файла в качестве текста запроса (пример 4.6).

Пример 4.6. Отправка файловых данных с помощью Fetch API

```
/**
 * Имея на входе форму с типом данных 'file', посылает POST-запрос с данными файла
 * в качестве тела запроса.
 * @param  form - объект формы (должен иметь тип данных 'file' с именем 'file')
 * @returns Promise, который разрешается, когда получен ответ в виде JSON-строки
 */
```

```
function uploadFile(form) {
  const formData = new FormData(form);
  const fileData = formData.get('file');
  return fetch('https://httpbin.org/post', {
    method: 'POST',
    body: fileData
  })
    .then(response => response.json());
}
```

Обсуждение

Для загрузки файла с использованием современных API-интерфейсов браузера требуется выполнить не так много шагов. `<input type="file">` предоставляет данные файла через FormData API и включается в текст запроса POST. Обо всем остальном позаботится браузер.

4.5. Отправка Beacon

Задача

Вы хотите отправить быстрый запрос, не дожидаясь ответа, например, для отправки аналитических данных.

Решение

Используйте Beacon API для отправки данных в POST-запросе. Обычный POST-запрос с помощью Fetch API может не завершиться до момента загрузки страницы. Использование маячка (beacon) с большей вероятностью приведет к успеху (пример 4.7). Браузер не ожидает ответа, и отправленный запрос, скорее всего, будет успешен, когда пользователь уже покинет ваш сайт.

Пример 4.7. Отправка Beacon

```
const currentUser = {
  username: 'sysadmin'
};

// Какие-то аналитические данные, которые мы хотим получить.
const data = {
  user: currentUser.username,
  lastVisited: new Date()
};
```

```
// Отправляет данные перед выгрузкой.
document.addEventListener('visibilitychange', () => {
  // Если свойство visibility имеет значение 'hidden', это означает,
  // что страница только что стала скрытой.
  if (document.visibilityState === 'hidden') {
    navigator.sendBeacon('/api/analytics', data);
  }
});
```

Примечание о надежности маячков

Ранее рекомендовалось использовать события beforeunload или unload для отправки аналитических маячков, но во многих случаях это может быть ненадежным. Многие сайты, такие как MDN (**https://oreil.ly/iBoG-**), теперь рекомендуют вместо этого использовать событие visibilitychange.

Обсуждение

При вызове XMLHttpRequest или fetch браузер ожидает ответа и возвращает его (с событием или промисом). Как правило, вам не нужно ждать ответа для односторонних запросов, таких как запрос аналитических данных.

Вместо промиса navigator.sendBeacon возвращает логическое значение, указывающее, была ли операция отправки запланирована. Дальнейших событий или уведомлений нет.

navigator.sendBeacon всегда отправляет POST-запрос. Если вы хотите отправить несколько наборов аналитических данных, таких как набор взаимодействий с пользовательским интерфейсом, вы можете собирать их в массив по мере того, как пользователь взаимодействует с вашей страницей, а затем отправить массив в теле POST-запроса с маячком.

4.6. Прослушивание удаленных событий с помощью server-sent events

Задача

Вы хотите получать уведомления от вашего сервера без повторяющихся запросов.

Решение

Используйте EventSource API для получения событий, отправляемых сервером (server-sent events, SSE).

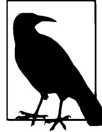
Для того чтобы начать прослушивание SSE, создайте новый экземпляр EventSource, передав URL-адрес в качестве первого аргумента (пример 4.8).

Пример 4.8. Открытие SSE-соединения

```
const events = new EventSource('https://example.com/events');

// Запускается сразу после соединения.
events.addEventListener('open', () => {
  console.log('Connection is open');
});

// Срабатывает, если возникает ошибка при соединении.
events.addEventListener('error', event => {
  console.log('An error occurred:', event);
});

// Срабатывает, если получено событие типа 'heartbeat'.
events.addEventListener('heartbeat', event => {
  console.log('got heartbeat:', event.data);
});

// Срабатывает, если получено событие типа 'notice'.
events.addEventListener('notice', event => {
  console.log('got notice:', event.data);
})

// EventSource оставляет соединение открытым. Если мы хотим закрыть
// соединение, нам нужно вызвать close для объекта EventSource.
function cleanup() {
  events.close();
}
```

Обсуждение

EventSource должен подключаться к специальной конечной точке HTTP, которая оставляет соединение открытым с заголовком Content-Type со значением text/event-stream. Всякий раз, когда происходит событие, сервер может отправлять новое сообщение по открытому соединению.



Как указывает MDN (**https://oreil.ly/MliFN**), настоятельно рекомендуется использовать HTTP/2 с SSE. Иначе браузеры устанавливают строгие ограничения на количество подключений к `EventSource` для каждого домена. В этом случае может быть не более шести подключений.

Это ограничение действует независимо от вкладки; оно распространяется на все вкладки браузера в данном домене.

Когда `EventSource` получает событие посредством постоянного соединения, это событие представляет собой обычный текст. Вы можете получить доступ к тексту события из свойства `data` объекта `event`. Вот пример события типа `notice`:

```
event: notice
data: Connection established at 10:51 PM, 2023-04-22
id: 3
```

Для того чтобы прослушать это событие, вызовите `addEventListener('notice')` для объекта `EventSource`. Объект `event` имеет свойство `data`, значение которого равно любому строковому значению с префиксом `data:`.

Если у события нет типа события, вы можете прослушать общее событие `message`, чтобы получить его.

4.7. Обмен данными с WebSockets в режиме реального времени

Задача

Вы хотите отправлять и получать данные в режиме реального времени без необходимости повторно опрашивать сервер с помощью Fetch-запросов.

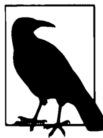
Решение

Используйте WebSocket API, чтобы открыть постоянное соединение с вашим сервером (пример 4.9).

Пример 4.9. Создание соединения посредством веб-сокетов

```
// Открывает соединение Websocket (URL должен начинаться с ws: или //wss:).
const socket = new WebSocket(url);

socket.addEventListener('open', onSocketOpened);
socket.addEventListener('message', handleMessage);
socket.addEventListener('error', handleError);
socket.addEventListener('close', onSocketClosed);
```

```
function onSocketOpened() {
  console.log('Socket ready for messages');
}

function handleMessage(event) {
  console.log('Received message:', event.data);
}

function handleError(event) {
  console.log('Socket error:', event);
}

function onSocketClosed() {
  console.log('Connection was closed');
}
```



Для того чтобы использовать веб-сокеты, на вашем сервере должна быть конечная точка с поддержкой WebSocket, к которой вы можете подключиться. В MDN есть хороший обзор создания WebSocket-сервера (**https://oreil.ly/fzX67**).

Как только сокет инициирует событие `open`, вы можете начать отправку сообщений, как показано в примере 4.10.

Пример 4.10. Отправка сообщений через веб-сокет

```
// Сообщения - это простые строки.
socket.send('Hello');

// Сокет требует данных в виде строки, вы можете сериализовать
// объекты с помощью JSON.stringify.
socket.send(JSON.stringify({
  username: 'sysadmin',
  password: 'password'
}));
```

Подключение через WebSocket является двунаправленным. Полученные данные с сервера запускают событие `message`. Вы можете обработать их по мере необходимости или даже отправить ответ (пример 4.11).

Пример 4.11. Ответ на сообщение WebSocket

```
socket.addEventListener('message', event => {
  socket.send('ACKNOWLEDGED');
});
```

Наконец, чтобы навести порядок по окончании сеанса, вы можете закрыть соединение, вызвав для объекта WebSocket метод `close`.

Обсуждение

WebSockets хорошо подходят для приложений, требующих работы в режиме реального времени, таких как система чата или мониторинг событий. Конечные точки WebSocket имеют схему `ws://` или `wss://`. Они аналогичны `http://` и `https://`. Во втором случае используется шифрование.

Для того чтобы инициировать подключение к веб-сокету, браузер сначала отправляет веб-сокету GET-запрос. Полезная нагрузка запроса для URL-адреса `wss://example.com/websocket` выглядит примерно так:

```
GET /websocket HTTP/1.1
Host: example.com
Sec-WebSocket-Key: aSBjYW4gaGFzIHdzIHBsej8/
Sec-WebSocket-Version: 13
Connection: Upgrade
Upgrade: websocket
```

Это инициирует так называемое WebSocket-рукопожатие (handshake). В случае успеха сервер отправляет в ответ стандартный код 101 (переключение протоколов):

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: bm8gcGVla2luZywgcGxlYXNlIQ==
```

WebSocket-протокол определяет алгоритм генерации заголовка `Sec-Websocket-Accept` на основе `Sec-WebSocket-Key` запроса. Клиент проверяет это значение, и в этот момент двустороннее сокет-соединение становится активным, затем сокет запускает событие `open`.

Как только соединение будет открыто, вы сможете прослушивать сообщения с помощью события `message` и отправлять сообщения, вызывая `send` для объекта `socket`. Позже вы сможете завершить работу с веб-сокетом путем вызова метода `close` для объекта `socket`.

ГЛАВА 5
База данных IndexedDB

5.0. Введение

В *главе 2* описывалось локальное сохранение данных в браузере пользователя. Оно хорошо работает для строковых значений и допускающих сериализацию объектов, но выполнение запроса не является идеальным, и к тому же объектам необходима JSON-сериализация. *IndexedDB* — это новый, более мощный механизм сохранения данных, представленный во всех современных браузерах. База данных IndexedDB содержит *хранилища объектов* (что-то вроде таблиц в реляционной базе данных). Каждое хранилище объектов может иметь индексы определенных свойств для более эффективного запроса. IndexedDB также поддерживает более продвинутые концепции, такие как управление версиями и транзакции.

Хранилища объектов и индексы

База данных IndexedDB содержит одно или несколько хранилищ объектов. Все операции по добавлению, удалению или запросу данных выполняются в хранилище объектов. Хранилище объектов — это коллекция объектов JavaScript, которые сохраняются в базе данных. Вы можете определить *индексы* для хранилища объектов. Индекс хранит дополнительную информацию в базе данных, которая позволяет запрашивать объекты по свойству `indexed`. Например, предположим, что вы создаете базу данных для хранения информации о продукте. У каждого продукта есть ключ, скорее всего, это идентификатор продукта (ID) или артикул (stock keeping unit, SKU). Они позволяют вам быстро выполнить поиск по базе данных и найти нужный продукт.

Если вам нужно запрашивать данные по цене, можете создать индекс для свойства `price`. С помощью индекса вы можете указать конкретную цену или диапазон цен, и индекс обеспечит вам быстрый поиск нужных записей.

Ключи

Объекты хранилища имеют *ключ*, который однозначно идентифицирует тот или иной объект в хранилище. Ключ объекта хранилища аналогичен первичному ключу

в таблице реляционной базы данных. В хранилище объектов IndexedDB есть два типа ключей.

Встроенные (in-line) ключи определяются в самом объекте. Например, вот список намеченных к решению задач со встроенным ключом:

```
{
  // Здесь ключом является id.
  id: 100,
  name: 'Take out the trash',
  completed: false
}
```

Здесь ключом является свойство id. При добавлении элементов задач в такое хранилище объектов для них должно быть определено свойство id. Кроме того, при создании хранилища объектов необходимо указать *путь к ключу* id. Путь к ключу сообщает IndexedDB имя свойства, содержащего встроенный ключ:

```
const todosStore = db.createObjectStore('todos', { keyPath: 'id' });
```

Если вы хотите использовать встроенные ключи и не беспокоиться о сохранении их уникальности, можете указать IndexedDB использовать автоматический инкремент значения ключей:

```
const todosStore = db.createObjectStore('todos',
  { keyPath: 'id', autoIncrement: true });
```

Внешние (out-of-line) ключи не сохраняются в объекте. Внешний ключ указывается в качестве отдельного аргумента с помощью add или put при сохранении объекта. Следуя предыдущему примеру, вы также можете использовать внешние ключи для списка задач. Это означает, что ключ или свойство id не будут храниться как часть объекта:

```
const todo = {
  name: 'Take out the trash',
  completed: false
};
// позже при добавлении нового элемента в список
todoStore.add(todo, 100);
```

Транзакции

Операции с базой данных IndexedDB используют *транзакции*. Транзакция — это логическая группировка задач базы данных, реализуемых совместно для выполнения некоторой работы. Они предназначены для защиты целостности данных в базе данных. Если одна из операций в рамках транзакции завершается неудачей, вся транзакция считается неудачной, и любая завершенная работа возвращается к состоянию, существовавшему до начала транзакции.

Транзакция может быть доступна только для чтения или для чтения-записи, в зависимости от типа операции, которую вы хотите выполнить. Вы можете создать тран-

закцию, вызвав метод `transaction` базы данных IndexedDB. При этом следует передать имена всех хранилищ объектов, которые должны быть задействованы в этой транзакции, и тип транзакции (только для чтения (`readonly`) или чтения-записи (`readwrite`)).

Как только у вас будет транзакция, вы сможете получить ссылку на нужные вам хранилища объектов и приступить к выполнению операции с базой данных. Эти операции возвращают *запрос* IndexedDB. Все операции чтения и записи данных в базе данных IndexedDB требуют проведения транзакции.

Запросы

Когда вы выполняете операцию с хранилищем объектов в рамках транзакции, вы получаете обратно объект `request`, который реализует интерфейс `IDBRequest`. Запрошенная работа начинается асинхронно.

Когда работа выполнена, объект `request` запускает событие `success`, содержащее результаты. Например, событие успешного выполнения операции запроса включает в себя объекты, найденные с помощью этого запроса.

На рис. 5.1 показан общий процесс функционирования IndexedDB: формирование транзакции, открытие хранилища объектов, создание запроса и отслеживание событий.

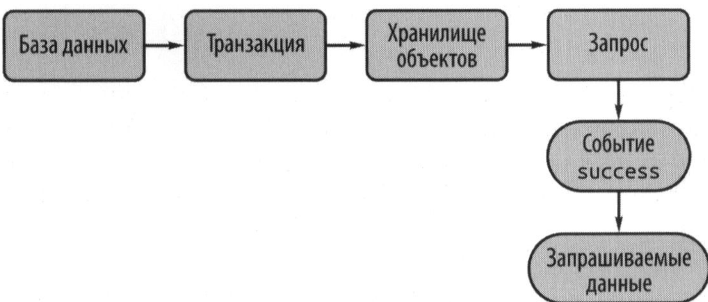


Рис. 5.1. Элементы работы с базой данных IndexedDB

5.1. Создание, чтение и удаление объектов в базе данных

Задача

Вы хотите реализовать базу данных IndexedDB, в которой можно создавать, читать и удалять объекты. Например, это может быть список контактов.

Решение

Создайте базу данных с единственным хранилищем объектов и определите операции создания, чтения и удаления.

Для того чтобы создать или открыть базу данных, вызовите IndexedDB.open (пример 5.1). Если база данных не была создана ранее, запускается событие upgradeneeded, требующее обновления. В обработчике этого события вы можете создать хранилище объектов. Когда база данных открыта и готова к использованию, запускается событие success (успешного завершения).

Пример 5.1. Открытие базы данных

```
/**
 * Открывает базу данных и при необходимости создает объект хранилища.
 * Поскольку это асинхронный процесс, требуется функция обратного вызова onSuccess.
 * Как только база данных будет готова, onSucces будет вызвана с объектом базы данных.
 * @param onSuccess - функция обратного вызова, которая выполняется,
 *                    когда база данных готова
 */
function openDatabase(onSuccess) {
  const request = indexedDB.open('contacts');

  // При необходимости создает хранилище объекта.
  request.addEventListener('upgradeneeded', () => {
    const db = request.result;

    // Объекты contact будут иметь свойство 'id', которое будет
    // использоваться в качестве ключа. Когда вы добавляете новый объект contact,
    // вам не нужно устанавливать свойство 'id'; флаг автоинкремента означает,
    // что база данных будет автоматически устанавливать для вас 'id'.
    db.createObjectStore('contacts', {
      keyPath: 'id',
      autoIncrement: true
    });
  });

  // Когда база данных готова к использованию, она запускает событие success.
  request.addEventListener('success', () => {
    const db = request.result;
    // Сделайте обратный вызов с возвратом базы данных.
    onSuccess(db);
  });

  // Всегда обрабатывайте ошибки!
  request.addEventListener('error', () => {
    console.error('Error opening database:', request.error);
  });
}
```

Перед отображением контактов вам необходимо загрузить их из базы данных. Для этого используйте транзакцию только для чтения и вызовите метод `getAll` хранилища объектов, который извлекает все объекты из хранилища (пример 5.2).

Пример 5.2. Считывание контактов

```
/**
 * Читает контакты из базы данных и выводит их в таблицу.
 * @param contactsDb - база данных IndexedDB
 * @param onSuccess  - функция обратного вызова,
 *                     выполняющаяся по окончании загрузки контактов
 */
function getContacts(contactsDb, onSuccess) {
  const request = contactsDb
    .transaction(['contacts'], 'readonly')
    .objectStore('contacts')
    .getAll();

  // Когда данные загружены, база данных генерирует событие success.
  request.addEventListener('success', () => {
    console.log('Got contacts:', request.result);
    onSuccess(request.result);
  });

  request.addEventListener('error', () => {
    console.error('Error loading contacts:', request.error);
  });
}
```

Для добавления контакта требуется транзакция типа `readwrite`. Передайте объект `contact` методу `add` хранилища объектов (пример 5.3).

Пример 5.3. Добавление контакта

```
/**
 * Добавляет контакт в базу данных, после чего вновь выводит таблицу контактов.
 * @param contactsDb - база данных IndexedDB
 * @param contact    - новый объект contact для добавления
 * @param onSuccess  - обратный вызов, выполняемый после добавления контакта
 */
function addContact(contactsDb, contact, onSuccess) {
  const request = contactsDb
```

```
    .transaction(['contacts'], 'readwrite')
    .objectStore('contacts')
    .add(contact);

  request.addEventListener('success', () => {
    console.log('Added new contact:', contact);
  onSuccess();
});

  request.addEventListener('error', () => {
    console.error('Error adding contact:', request.error);
  });
}
```

Вам также потребуется транзакция readwrite для удаления контакта (пример 5.4).

Пример 5.4. Удаление контакта

```
/**
 * Удаляет контакт из базы данных, после чего вновь выводит таблицу контактов.
 * @param contactsDb - база данных IndexedDB
 * @param contact    - объект contact для удаления
 * @param onSuccess  - обратный вызов, выполняемый после удаления контакта
 */

function deleteContact(contactsDb, contact, onSuccess) {
  const request = contactsDb
    .transaction(['contacts'], 'readwrite')
    .objectStore('contacts')
    .delete(contact.id);

  request.addEventListener('success', () => {
    console.log('Deleted contact:', contact);
    onSuccess();
  });

  request.addEventListener('error', () => {
    console.error('Error deleting contact:', request.error);
  });
}
```

Обсуждение

При создании базы данных вы вызываете IndexedDB.open, который создает запрос на открытие базы данных. Если в результате вызывается событие upgradeneeded, вы можете создать необходимое хранилище объектов.

> **Версии IndexedDB**
>
> В IndexedDB есть понятие *версионной* базы данных. Каждый раз, когда вы вносите изменения в схему своей базы данных (в случае IndexedDB — изменение набора хранилищ объектов и индексов), вам необходимо учитывать всех пользователей, у которых в браузерах уже сохранена более старая версия базы данных.
>
> Именно здесь появляется событие upgradeneeded. Вызывая indexedDB.open, вы можете указать номер версии для базы данных. Каждый раз, внося изменение в схему базы, вы увеличиваете это число. Если браузер пользователя с более старой версией вашей базы данных обнаруживает этот новый номер версии, IndexedDB запускает событие upgradeneeded. Оно сообщает вам о старой и новой версиях базы данных. Учитывая эти факторы, вы можете определить, какие изменения вам необходимо внести в базу данных.
>
> Так вы можете совершенствовать дизайн своей базы данных, сохраняя при этом данные пользователей нетронутыми.

У каждого объекта в хранилище объектов должен быть уникальный ключ. Если вы попытаетесь добавить объект с дублирующимся ключом, вы получите сообщение об ошибке.

Схема для других операций, как правило, одинакова:

1. Создайте транзакцию.
2. Получите доступ к хранилищу объектов.
3. Вызовите нужный метод в хранилище объектов.
4. Прослушайте событие success.

При этом каждая функция, реализующая ту или иную операцию, принимает аргумент onSuccess, вызываемый в случае успеха. Поскольку база данных IndexedDB является асинхронной, вам нужно дождаться завершения операции, прежде чем продолжить. Функция openDatabase передает базу данных в функцию onSuccess, где вы можете сохранить ее в переменной на будущее (пример 5.5).

Пример 5.5. Использование функции openDatabase

```
let contactsDb;

// Открывает базу данных и выполняет просмотр списка контактов.
// Обработчик события success настраивает contactsDb на прием нового объекта,
// затем загружает и просматривает контакты.
```

```
openDatabase(db => {
  contactsDb = db;
  renderContacts(contactsDb);
});
```

Как только вы настроите переменную contactsDb, можете передать ее другим операциям с базой данных. Если вы хотите отобразить список контактов, вам нужно дождаться, пока они сначала будут загружены, поэтому вы должны передать обработчик успешного выполнения, который получает объекты контактов и отображает их (пример 5.6).

Пример 5.6. Загрузка и просмотр контактов

```
getContacts(contactsDb, contacts => {
  // Контакты загружены, теперь надо их визуализировать.
  renderContacts(contacts);
});
```

Аналогично при внесении нового контакта необходимо дождаться добавления нового объекта, а затем загрузить и отобразить обновленный список контактов (пример 5.7).

Пример 5.7. Добавление и повторное отображение контактов

```
const newContact = { name: 'Connie Myers', email: 'cmyers@example.com' };
addContact(contactsDb, newContact, () => {
  // Контакт был добавлен, теперь загрузите обновленный список и отобразите его.
  getContacts(contactsDb, contacts => {
    renderContacts(contacts);
  })
});
```

Если вы не хотите постоянно передавать ссылку на базу данных, вы можете инкапсулировать внутри нового объекта ссылку на нее и требуемые функции, как показано в примере 5.8.

Пример 5.8. Инкапсулированная база данных

```
const contactsDb = {
  open(onSuccess) {
    const request = indexedDB.open('contacts');
```

```
    request.addEventListener('upgradeneeded', () => {
      const db = request.result;
      db.createObjectStore('contacts', {
        keyPath: 'id',
        autoIncrement: true
      });
    });

    request.addEventListener('success', () => {
      this.db = request.result;
      onSuccess();
    });
  },

  getContacts(onSuccess) {
    const request = this.db
      .transaction(['contacts'], 'readonly')
      .objectStore('contacts')
      .getAll();
    request.addEventListener('success', () => {
      console.log('Got contacts:', request.result);
      onSuccess(request.result);
    });
  },

  // Другие операции добавляются аналогичным образом.
};
```

При таком подходе вам по-прежнему нужны обратные вызовы, уведомляющие вас о выполнении операций, но объект `contactsDb` отслеживает ссылку на базу данных для вас. (Использования глобальной переменной не требуется!)

5.2. Обновление существующей базы данных

Задача

Вы хотите обновить существующую базу данных, чтобы добавить новое хранилище объектов.

Решение

Используйте новую версию базы данных. При обработке события upgradeneeded в зависимости от версии определите, нужно ли добавлять в базу данных текущего пользователя новое хранилище объектов.

Представьте, что у вас есть база данных списка дел с хранилищем объектов todos. Позже, при обновлении вашего приложения, вы захотите добавить новое хранилище объектов people, чтобы можно было назначать задачи пользователям.

Для вызова indexedDB.open теперь требуется новый номер версии. Вы можете увеличить номер версии до 2 (пример 5.9).

Пример 5.9. Обновление базы данных

```
// База данных todoList теперь имеет версию 2
const request = indexedDB.open('todoList', 2);

// Если база данных пользователя имеет версию 1, запускается событие
// upgradeneeded, поэтому может быть добавлен новый объект хранилища.
request.addEventListener('upgradeneeded', event => {
  const db = request.result;

  // Это же событие запускается в случае отсутствия базы данных.
  // Этот случай тоже нуждается в обработке и создании объекта todos базы данных.
  // Свойство oldVersion содержит текущую версию базы данных пользователя.
  // Если база данных только что создана, oldVersion равно 0.
  if (event.oldVersion < 1) {
    db.createObjectStore('todos', {
      keyPath: 'id'
    });
  }

  // Если база данных еще не обновлена до версии 2, создать новый объект базы данных.
  if (event.oldVersion < 2) {
    db.createObjectStore('people', {
      keyPath: 'id'
    });
  }
});

request.addEventListener('success', () => {
  // База данных готова.
});
```

```
// Фиксируйте любую возможную ошибку. Объект error хранится
// в свойстве error запроса.
request.addEventListener('error', () => {
  console.error('Error opening database:', request.error);
});
```

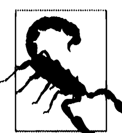
Обсуждение

При вызове `indexedDB.open` вы можете указать версию базы данных. Если вы не сделаете это, по умолчанию версия будет равна 1.

Всякий раз, когда база данных открывается, текущая версия базы данных в браузере (если таковая имеется) сравнивается с номером версии, переданным в `indexedDB.open`. Если база данных еще не существует или версия не обновлена, вы получите событие `upgradeneeded`, требующее обновления.

В обработчике события `upgradeneeded` вы можете проверить свойство `oldVersion`, чтобы определить текущую версию базы данных браузера. Если база данных еще не существует, значение `oldVersion` равно 0.

Основываясь на старой версии, вы можете определить, какие хранилища объектов и индексы уже существуют, а какие необходимо добавить.



Если вы попытаетесь создать хранилище объектов или индекс, который уже существует, браузер выдаст исключение. Перед созданием этих объектов обязательно проверьте свойство `oldVersion` события.

5.3. Выполнение запросов с использованием индексов

Задача

Вы хотите эффективно запрашивать данные на основе значения свойства, отличного от ключа (обычно называемого первичным ключом).

Решение

Создайте индекс для этого свойства, затем выполните запрос по этому индексу.

Рассмотрим пример базы данных сотрудников. У каждого сотрудника есть имя, место работы (отдел) и уникальный идентификатор ID в качестве ключа. Возможно, вы захотите отфильтровать сотрудников по принадлежности к определенному отделу.

Когда запускается событие `upgradeneeded` и вы создаете хранилище объектов, вы также можете определить индексы для этого хранилища объектов (пример 5.10). В примере 5.11 показано, как выполнять запрос по указанному индексу.

Пример 5.10. Определение индекса при создании хранилища объектов

```
/**
 * Открывает базу данных, при необходимости создавая хранилище объектов
 * и индексируя их.
 * Как только база данных будет готова, onSuccess будет вызван
 * с объектом базы данных.
 * @param onSuccess - функция обратного вызова, которая выполняется,
 *                    когда база данных готова.
 */
function openDatabase(onSuccess) {
  const request = indexedDB.open('employees');

  request.addEventListener('upgradeneeded', () => {
    const db = request.result;

    // Новым объектам employee будет присвоено автоматически
    // сгенерированное свойство id, которое будет служить их ключом.
    const employeesStore = db.createObjectStore('employees', {
      keyPath: 'id',
      autoIncrement: true,
    });

    // Создайте индекс для свойства department под названием 'department'.
    employeesStore.createIndex('department', 'department');
  });

  request.addEventListener('success', () => {
    onSuccess(request.result);
  });
}
```

Пример 5.11. Запрос сотрудников по индексу отдела

```
/**
 * Получает данные о сотрудниках данного отдела или обо всех сотрудниках,
 * если не указан ни один отдел.
 * @param department - название отдела
 * @param onSuccess  - функция обратного вызова, которая выполняется
 *                     при загрузке сотрудников
 */
```

```
function getEmployees(department, onSuccess) {
  const request = employeeDb
    .transaction(['employees'], 'readonly')
    .objectStore('employees')
    .index('department')
    .getAll(department);

  request.addEventListener('success', () => {
    console.log('Got employees:', request.result);
    onSuccess(request.result);
  });

  request.addEventListener('error', () => {
    console.log('Error loading employees:', request.error);
  });
}
```

Обсуждение

В зависимости от ваших потребностей хранилище объектов IndexedDB может содержать более одного индекса.

В этом примере для запроса индекса используются определенные значения, но индекс также может быть запрошен для *диапазона* ключей. Эти диапазоны определяются с помощью интерфейса `IDBKeyRange`. Диапазон определяется с точки зрения его *границ* — он указывает начальную и конечную точки, и возвращаются все ключи в пределах этого диапазона.

Интерфейс `IDBKeyRange` поддерживает четыре типа границ.

`IDBKeyRange.lowerBound`

Определяет нижнюю границу ключей.

`IDBKeyRange.upperBound`

Определяет верхнюю границу ключей.

`IDBKeyRange.bound`

Определяет нижнюю *и* верхнюю границы.

`IDBKeyRange.only`

Указывает только на один ключ.

Диапазоны ключей `lowerBound`, `upperBound` и `bound` также принимают второй логический параметр, указывающий, является ли диапазон открытым или закрытым. Если указано значение `true`, то он считается *открытым* диапазоном и исключает сами границы. `IDBKeyRange.upperBound(10)` соответствует всем ключам, которые *меньше или равны* 10, но `IDBKeyRange.upperBound(10, true)` соответствует всем ключам, кото-

рые *меньше* 10, потому что само значение 10 исключено. Границы диапазона ключей не обязательно должны быть числами. В качестве ключей могут использоваться объекты других типов, такие как строки и объекты-даты.

5.4. Поиск строковых значений с помощью курсора

Задача

Вы хотите сделать запрос в хранилище объектов IndexedDB, чтобы найти строковые объекты со свойством, соответствующим шаблону.

Решение

Используйте курсор, проверяя свойство каждого объекта, чтобы увидеть, содержит ли он заданную строку.

Представьте себе приложение со списком сотрудников. Вы хотите выполнить поиск по всем контактам, имя которых содержит введенный текст. Для этого примера предположим, что база данных уже открыта, а хранилище объектов называется `employees`.

Курсор перемещается по каждому объекту в хранилище объектов. Он останавливается на каждом объекте, и вы можете получить доступ к текущему элементу и/или перейти к следующему элементу. Вы можете проверить, содержит ли имя контакта текст запроса, и собрать результаты в массив (пример 5.12).

Пример 5.12. Поиск строковых значений с помощью курсора

```
/**
 * Ищем сотрудников по имени.
 *
 * @param name - строка запроса
 * @param onSuccess - обратный вызов, который получит имена найденных сотрудников
 */
function searchEmployees(name, onSuccess) {
  // Массив из всех контактов с именем, содержащим текст запроса
  const results = [];

  const query = name.toLowerCase();

  const request = employeeDb
    .transaction(['employees'], 'readonly')
    .objectStore('employees')
    .openCursor();
```

```
  // Запрос курсора выдаст событие success для каждого найденного объекта.
  request.addEventListener('success', () => {
    const cursor = request.result;
    if (cursor) {
      const name = `${cursor.value.firstName} ${cursor.value.lastName}`
      .toLowerCase();
      // Добавит контакт в результирующий массив, если он соответствует запросу.
      if (name.includes(query)) {
        results.push(cursor.value);
      }

      // Перейдите к следующей записи.
      cursor.continue();
    } else {
      onSuccess(results);
    }
  });

  request.addEventListener('error', () => {
    console.error('Error searching employees:', request.error);
  });
}
```

Обсуждение

Когда вы вызываете openCursor в хранилище объектов, он возвращает объект запроса `IDBRequest`. Далее запускается событие `success` для первого объекта в хранилище. Для каждого события `success` запрос имеет свойство `result`, которым является сам объект `cursor`. Вы можете получить доступ к текущему значению, на которое указывает курсор, с помощью свойства `value`.

Обработчик результатов проверяет поля имени и фамилии текущего объекта, сначала преобразуя оба в нижний регистр, чтобы при поиске регистр не учитывался. Если есть совпадение, то объект добавляется в результирующий массив.

Когда вы закончите обработку текущего объекта, для курсора вы можете вызвать команду `continue`. Это приведет к переходу к следующему объекту и вызовет еще одно событие `success`. Если вы достигли дна хранилища объектов и в нем не осталось объектов, `request.result` будет равен `null`. Если это происходит, то является сигналом, что поиск завершен и у вас есть искомые контакты.

При каждом перемещении курсора все объекты, соответствующие поисковому запросу, добавляются в массив результатов `results`. Этот массив передается в обратный вызов `success`, когда процесс поиска завершен.

5.5. Разбивка большого набора данных на страницы

Задача

Вы хотите разбить большой набор данных на страницы, каждая из которых имеет свое смещение (offset) и длину (length).

Решение

Используйте курсор, чтобы перейти к первому элементу на запрашиваемой странице и набрать нужное количество элементов (пример 5.13).

Пример 5.13. Использование курсора для получения страницы с записями

```
/**
 * Использует курсор для извлечения одной "страницы" данных
 * из хранилища объектов IndexedDB.
 *
 * @param db - объект базы данных IndexedDB
 * @param storeName - имя хранилища объектов
 * @param offset - начальное смещение (номер первого элемента 0)
 * @param length - число возвращаемых элементов от точки смещения
 */
function getPaginatedRecords(db, storeName, offset, length) {
  const cursor = db
    .transaction([storeName], 'readonly')
    .objectStore(storeName)
    .openCursor();

  const results = [];

  // Этот флаг указывает, переместился ли курсор вперед на величину смещения или нет.
  let skipped = false;

  request.addEventListener('success', event => {
    const cursor = event.target.result;

    if (!skipped) {
      // Установите флаг и перейдите вперед на заданное смещение.
      // В следующий раз курсор будет находиться в стартовом положении,
      // и можно будет начать сбор записей.
      skipped = true;
      cursor.advance(offset);
```

```
    } else if (cursor && result.length < length) {
      // Соберите запись, на которую в данный момент указывает курсор.
      results.push(cursor.value);

      // Переходите к следующей записи.
      cursor.continue();
    } else {
      // Либо записей не осталось, либо заданная длина была достигнута.
      console.log('Got records:', request.result);
    }
  });

  request.addEventListener('error', () => {
    console.error('Error getting records:', request.error);
  });
}
```

Обсуждение

Возможно, вы не захотите начинать с первой записи, поэтому и нужен аргумент `offset`. При первом запуске обработчик событий вызывает метод `advance` с запрошенным смещением. Метод указывает курсору перейти к нужному начальному элементу. С технической точки зрения, `advance` не передвигается в указанное место, а скорее перемещается на заданную позицию *относительно текущего индекса*. Однако в данном примере это фактически одно и то же, поскольку действие всегда начинается с нулевого индекса.

Вы не сможете начать сбор значений до следующей итерации перемещения курсора. Справиться с этим помогает установленный флаг `skipped`, указывающий на то, что курсор переместился вперед. При следующем прохождении этот флаг будет отображаться как значение `true`, и курсор попытается перейти вперед еще раз.

Как только курсор переместится вперед, будет запущено еще одно событие `success`. Теперь курсор указывает на первый элемент, который нужно выбрать (при условии, что элементы остались; объект курсора равен `null`, если объектов больше нет). Текущее значение будет добавлено в результирующий массив. Далее вызывается метод `continue` для перемещения курсора к следующему значению.

Этот процесс продолжается до тех пор, пока результирующий массив не достигнет заданной длины или хранилище объектов не будет исчерпано. Последнее произойдет, если значение `offset` + `length` превысит количество объектов в хранилище объектов.

Как только объектов для сбора больше не останется, будет готова полная страница результатов.

5.6. Использование промисов с IndexedDB API

Задача

Вам нужен основанный на промисах API для работы с базой данных IndexedDB.

Решение

Создайте обертку `Promise` для запросов к IndexedDB. Когда запрос вызовет событие `success`, разрешите `Promise`. Если в результате будет вызвано событие `error`, отклоните его.

В примере 5.14 создается обертка для функции `indexedDB.open`. Она открывает или создает базу данных и возвращает промис, который выполняется, когда база данных готова.

Пример 5.14. Создание базы данных с промисом

```
/**
 * Открывает базу данных, при необходимости создавая хранилище объектов.
 * @returns Promise, который разрешается базой данных или отклоняется с ошибкой
 */
function openDatabase() {
  return new Promise((resolve, reject) => {
    const request = indexedDB.open('contacts-promise');

    // При необходимости создайте хранилище объектов.
    request.addEventListener('upgradeneeded', () => {
      const db = request.result;
      db.createObjectStore('contacts', {
        keyPath: 'id',
        autoIncrement: true
      });
    });

    request.addEventListener('success', () => resolve(request.result));
    request.addEventListener('error', () => reject(request.error));
  });
}
```

Для того чтобы загрузить некоторые данные из базы данных, в примере 5.15 используется оболочка вокруг метода getAll. Она запрашивает данные, затем возвращает Promise, который преобразуется в массив объектов после их загрузки.

Пример 5.15. Получение объектов из хранилища с использованием промиса

```
/**
 * Считывает контакты из базы данных.
 * @returns Promise, который разрешается в виде контактов или отклоняется с ошибкой
 */
function getContacts() {
  return new Promise((resolve, reject) => {
    const request = contactsDb
      .transaction(['contacts'], 'readonly')
      .objectStore('contacts')
      .getAll();

    request.addEventListener('success', () => {
      console.log('Got contacts:', request.result);
      resolve(request.result);
    });

    request.addEventListener('error', () => {
      console.error('Error loading contacts:', request.error);
      reject(request.error);
    });
  });
}
```

Теперь, когда у вас есть API, который возвращает промисы, вы можете использовать then или async/await при работе с вашей базой данных (пример 5.16).

Пример 5.16. Использование базы данных с промисами

```
async function loadAndPrintContacts() {
  try {
    const db = await openDatabase();
    const contacts = await getContacts();
    console.log('Got contacts:', contacts);
```

```
  } catch (error) {
    console.error('Error:', error);
  }
}
```

Обсуждение

Использование Promise API с функцией async/await устраняет необходимость в повторной передаче обработчиков успешных вызовов. Как показано в примере 5.16, вы также можете воспользоваться преимуществами цепочки промисов, чтобы избежать вложенных обратных вызовов и обработчиков событий.

ГЛАВА 6

Наблюдение за элементами DOM

6.0. Введение

В этой главе рассматриваются три типа *наблюдателей* (observers), которые браузер предоставляет вам для просмотра элементов DOM: `MutationObserver`, `IntersectionObserver` и `ResizeObserver`. Эти объекты-наблюдатели могут отслеживать элементы DOM и уведомлять вас об определенных изменениях или событиях.

Наблюдатели создаются с помощью функции обратного вызова. Эта функция вызывается всякий раз, когда на странице происходят соответствующие события. Она вызывается с одним или несколькими параметрами, содержащими информацию о том, что именно произошло. Однако такое действие всего лишь создает наблюдателя. Для того чтобы начать наблюдение за элементом, вам нужно вызвать функцию `observe` наблюдателя, передав элемент, за которым вы хотите наблюдать, и дополнительный набор параметров.

MutationObserver

`MutationObserver` отслеживает изменения, происходящие в элементе DOM. Вы можете отслеживать изменения:

- в дочерних элементах;
- в атрибутах;
- в текстовом содержимом.

Что именно отслеживает браузер, определяется в объекте `options`, передаваемом функции `observe`. При наблюдении за элементом вы также можете указать необязательный параметр `subtree`. Он расширяет возможности мониторинга дочерних элементов, атрибутов и/или текстового содержимого на все узлы-потомки (а не только на элемент и его прямые дочерние элементы).

Когда происходит интересующая вас мутация, выполняется обратный вызов с массивом объектов `MutationEntry`, которые описывают только что произошедшие изменения.

ResizeObserver

Как следует из названия, ResizeObserver уведомляет вас об изменении размера элемента. Как только размер изменяется, запускается функция обратного вызова с информацией о том, какой размер был модифицирован. Параметры содержат информацию о новом размере элемента.

IntersectionObserver

IntersectionObserver отслеживает изменения положения элемента относительно области просмотра. Область просмотра может быть прокручиваемым элементом или самим окном браузера. Если какая-либо часть дочернего элемента видна в пределах прокручиваемой области, считается, что она *пересекает* родительский элемент. На рис. 6.1 показаны элементы на прокручиваемой странице.

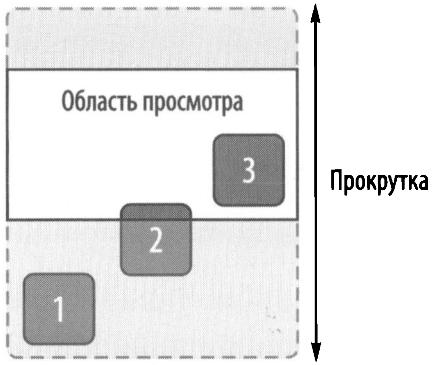


Рис. 6.1. Элемент 1 не пересекается, элемент 2 частично пересекается, а элемент 3 полностью пересекается с областью просмотра

IntersectionObserver использует понятие *коэффициента пересечения* — какая часть элемента на самом деле пересекла границу корневого элемента. Если элемент виден полностью, его коэффициент равен 1. Если он полностью скрыт от глаз пользователя, его коэффициент равен 0. Если он ровно наполовину виден и наполовину невидим, то его коэффициент равен 0,5. Среди параметров, передаваемых в функцию обратного вызова, имеется свойство intersectionRatio, определяющее текущий коэффициент пересечения.

При создании IntersectionObserver вы также можете указать *пороговое значение*. Оно определяет момент срабатывания наблюдателя. По умолчанию пороговое значение равно 0. Это означает, что observer запускается, как только элемент становится частично видимым, даже если это всего лишь один пиксел. Порог, равный 1, срабатывает только тогда, когда элемент становится полностью видимым.

6.1. Отложенная загрузка изображения при прокрутке

Задача

Вы хотите отложить загрузку изображения до тех пор, пока оно полностью не окажется в поле зрения пользователя. Иногда это называется *отложенной (lazy) загрузкой*.

Решение

Используйте `IntersectionObserver` для элемента `<img>` и дождитесь, пока он не пересечется с областью просмотра. Как только он попадет в область просмотра, установите атрибут `src`, чтобы начать загрузку изображения (пример 6.1).

Пример 6.1. Отложенная загрузка изображения с помощью IntersectionObserver

```
/**
 * Организует отложенную загрузку изображения.
 *
 * @param img - ссылка на элемент-изображение в узле DOM
 * @param url - URL-адрес загружаемого изображения
 */
function lazyLoad(img, url) {
  const observer = new IntersectionObserver(entries => {
    // isIntersecting становится равным true, как только изображение
    // появляется в области просмотра.
    // В этот момент установите URL изображения и остановите прослушивание.
    if (entries[0].isIntersecting) {
      img.src = url;
      observer.disconnect();
    }
  });

  // Начните наблюдение за элементом-изображением.
  observer.observe(img);
}
```

Обсуждение

Когда вы создаете `IntersectionObserver`, вы предоставляете ему функцию обратного вызова. Каждый раз, когда элемент входит в область просмотра или выходит из

нее, наблюдатель вызывает эту функцию с информацией о статусе пересечения элемента.

Наблюдатель может следить за несколькими элементами, пересечение которых может изменяться одновременно, поэтому при обратном вызове передается массив элементов. В примере 6.1 наблюдатель следит только за одним изображением, поэтому в массиве только один элемент.

Если несколько элементов одновременно входят в область просмотра (или покидают ее), для каждого элемента организуется одна запись.

Вы хотите проверить свойство `isIntersecting`, чтобы определить, пришло ли время загружать изображение. Оно принимает значение `true`, когда элемент становится хотя бы частично видимым.

Вы также должны сообщить наблюдателю, какой элемент следует просмотреть, вызвав `observe` для объекта `observer`. После этого начнется наблюдение за элементом.

Как только вы прокрутите страницу вниз настолько, что элемент попадет в область просмотра, наблюдатель запустит функцию обратного вызова. Обратный вызов устанавливает URL-адрес изображения и прекращает наблюдение, вызывая `disconnect`. Остановка наблюдения связана с тем, что в нем отпадает необходимость, как только изображение будет загружено.

До появления `IntersectionObserver` было не так много возможностей реализовать описанные действия. Одним из вариантов было прослушивание события `scroll` родительского элемента, а затем вычисление, находится ли элемент в области просмотра, путем сравнения границ родительского и дочернего элементов.

Всё это, конечно, не очень эффективно. А кроме того, считается плохой практикой. Вам пришлось регулировать или отменять сопоставление границ, чтобы оно не выполнялось при каждой прокрутке экрана.

Отложенная загрузка в новых браузерах

`IntersectionObserver` очень хорошо поддерживается браузерами, но если вы ориентируетесь исключительно на современные браузеры, есть способ отложенно загружать изображения без использования JavaScript-кода.

В новых браузерах элемент `<img>` поддерживает атрибут `loading`. Если для этого параметра установлено значение `lazy`, изображение не будет загружено до тех пор, пока элемент не появится в окне просмотра:

```
<img src="/path/to/image.jpg" loading="lazy">
```

Последние данные о поддержке браузерами атрибута `loading` вы можете найти на CanIUse (**https://oreil.ly/coP8C**).

6.2. Обертывание *IntersectionObserver* промисом

Задача

Вы хотите создать `Promise`, который выполняется, как только элемент попадает в область просмотра.

Решение

Оберните `IntersectionObserver` в `Promise`. Как только элемент пересекается со своим родительским элементом, разрешите промис (пример 6.2).

Пример 6.2. Обертывание IntersectionObserver промисом

```
/**
 * Возвращает Promise, который разрешается,
 * как только элемент входит в область просмотра
 */
function waitForElement(element) {
  return new Promise(resolve => {
    const observer = new IntersectionObserver(entries => {
      if (entries[0].isIntersecting) {
        observer.disconnect();
        resolve();
      }
    });

    observer.observe(element);
  });
}
```

Обсуждение

Когда наблюдатель выполняет функцию обратного вызова с параметром, указывающим на то, что элемент входит в область просмотра, вы можете разрешить промис.

Как показано в примере 6.3, можно использовать этот подход для отложенной загрузки изображения, аналогично рецепту 6.1.

Пример 6.3. Использование хелпера waitForElement для отложенной загрузки изображения

```
function lazyLoad(img, url) {
  waitForElement(img)
    .then(() => img.src = url)
}
```

Как только вы разрешите `Promise`, вызывающий код сможет убедиться, что элемент находится в области просмотра. В этот момент функция `lazyLoad` установит атрибут `src` для изображения.

6.3. Автоматическая пауза при воспроизведении видео

Задача

У вас есть элемент `<video>` в прокручиваемом контейнере. Во время воспроизведения видео вы хотите автоматически приостанавливать его, если оно выйдет из поля зрения.

Решение

Используйте `IntersectionObserver` для просмотра видеоэлемента. Как только он выйдет из области просмотра, приостановите видео. Позже, если он снова появится в области просмотра, возобновите его воспроизведение (пример 6.4).

Пример 6.4. Автоматическая приостановка и возобновление воспроизведения видео

```
const observer = new IntersectionObserver(entries => {
  if (!entries[0].isIntersecting) {
    video.pause();
  } else {
    video.play()
      .catch(error => {
        // В случае ошибки разрешения автоматического воспроизведения видео.
        // Так можно избежать необработанной ошибки отклонения, которая
        // способна привести к сбою в работе вашего приложения.
      });
  }
});

observer.observe(video);
```

Обсуждение

Этот наблюдатель следит за элементом `video`. Как только он исчезает из поля зрения, воспроизведение приостанавливается. Позже, если вы прокрутите область просмотра, воспроизведение возобновится.

> **Автоматическое воспроизведение видео**
>
> Браузеры строго относятся к автоматическому воспроизведению видео. При попытке программного воспроизведения видео, как в примере 6.4, браузер может выдать исключение. Если вы не отключите звук видео по умолчанию (установив атрибут `muted` для элемента `video`), вы не сможете автоматически или программно воспроизвести его, пока пользователь не начнет взаимодействие со страницей.
>
> Метод `play` видеоэлемента фактически возвращает промис. Для того чтобы изящно разрешить эту ситуацию, вам следует добавить к промису вызов `catch`.
>
> Однако не следует автоматически запускать воспроизведение видео при первой загрузке страницы. Это раздражает и создает проблемы с доступом. Например, автоматическое воспроизведение видео может оказать негативное влияние на человека с вестибулярным расстройством. Звук также может помешать чтению с экрана.
>
> В реальных приложениях это решение следует использовать только для удобства, уже после того как пользователь нажмет кнопку воспроизведения.

6.4. Анимация изменений высоты

Задача

У вас есть элемент, содержимое которого может меняться. Если содержимое меняется, вам нужен плавный переход по высоте.

Решение

Используйте `MutationObserver` для просмотра дочерних элементов элемента. Если элемент добавляет, удаляет или изменяет какие-либо дочерние элементы, используйте CSS-переход, чтобы плавно анимировать изменение высоты. Поскольку вы не можете анимировать элемент с автоматической высотой `auto`, потребуется некоторая дополнительная работа для явного вычисления высот, изменение которых нужно анимировать (пример 6.5).

Пример 6.5. Анимация высоты элемента в ходе изменений дочернего элемента

```
/**
 * Следит за элементом, если есть изменения в дочернем элементе. Когда высота
 * изменяется из-за дочернего элемента, анимирует это изменение.
```

```
 * @param element - элемент, за которым ведется наблюдение
 */
function animateHeightChanges(element) {
  // Нельзя анимировать элемент с помощью 'height: auto',
  // поэтому требуется явное указание высоты height.
  element.style.height = `${details.offsetHeight}px`;

  // Установите несколько свойств CSS, необходимых для анимированного перехода.
  element.style.transition = 'height 200ms';
  element.style.overflow = 'hidden';

  /**
   * Этот наблюдатель запускается при изменении дочерних элементов
   * родительского элемента.
   * Он измеряет новую высоту, затем использует requestAnimationFrame
   * для обновления высоты. Изменение высоты будет анимировано.
   */
  const observer = new MutationObserver(entries => {
    // entries - это всегда массив. Иногда этот массив может содержать
    // несколько элементов, но здесь вам нужен первый и единственный элемент.
    const element = entries[0].target;
    // Содержимое изменилось, а вместе с ним и высота. Для измерения
    // новой высоты необходимо выполнить несколько шагов.

    // (1) Запомните текущую высоту, которая будет использоваться
    // в качестве начальной точки анимации.
    const currentHeightValue = element.style.height;

    // (2) Установите для высоты значение "auto" и прочитайте свойство offsetHeight.
    // Это новая высота, которую нужно установить.
    element.style.height = 'auto';
    const newHeight = element.offsetHeight;

    // (3) Перед анимацией снова установите текущую высоту.
    element.style.height = currentHeightValue;

    // В следующем кадре анимации измените высоту.
    // Это приведет к запуску анимированного перехода.
    requestAnimationFrame(() => {
      element.style.height = `${newHeight}px`;
    });
  });
```

```
  // Начните отслеживать изменения в элементе.
  observer.observe(element, { childList: true });
}
```

Обсуждение

Как и в случае с другими наблюдателями, при создании `MutationObserver` вам необходимо передать функцию обратного вызова. Наблюдатель вызывает ее всякий раз, когда изменяется наблюдаемый элемент (какие именно изменения запускают обратный вызов, зависит от параметров, переданных функции `observer.observe`). Когда ваше приложение вносит какие-либо изменения в дочерний список элемента (добавляет, удаляет или модифицирует элементы), обратный вызов пересчитывает высоту для размещения нового содержимого.

Здесь много чего происходит, в основном из-за того, что браузер не позволяет анимировать элемент, если значение `height` равно `auto`. Для того чтобы анимация работала, вы должны использовать явные значения для начальной и конечной высоты.

При первом наблюдении за элементом вы вычисляете его высоту, считывая значение смещения `offsetHeight`. Затем функция явно устанавливает эту высоту для элемента. На данный момент можно изменить высоту автоматически при помощи `height: auto`.

Когда дочерние элементы элемента изменяются, родительский элемент не будет автоматически изменять размер, т. к. теперь у него явно задана высота. Функция обратного вызова `observer` вычисляет новую высоту. При явно заданной высоте свойство `offsetHeight` имеет то же значение.

Для того чтобы измерить *новую* высоту, вы должны сначала сбросить значение высоты *обратно* в значение `auto`. Как только вы это сделаете, `offsetHeight` выдаст новое значение высоты. Однако помните, что вы не можете анимировать с помощью установки `height: auto`. Прежде чем обновлять высоту, ее необходимо изменить с автоматической на ту величину, которая была установлена *ранее*.

Итак, у вас есть новая высота. Фактическое обновление высоты выполняется при вызове `requestAnimationFrame`.

Предложенный алгоритм вычисления высот требует написания большого дополнительного кода. В *главе 8* рассматривается API веб-анимации, которое упрощает создание анимаций такого типа.

6.5. Изменение содержимого элемента в зависимости от размера

Задача

Вы хотите отображать различное содержимое внутри элемента в зависимости от его размера. Например, вы можете захотеть что-то изменить в случае очень широкого элемента.

Решение

Используйте ResizeObserver для элемента и обновляйте содержимое, если размер становится больше или меньше установленного вами порога (пример 6.6).

Пример 6.6. Обновление содержимого элемента при изменении его размера

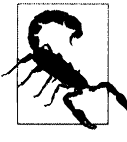
```
// Найдите элемент, за которым вы хотите наблюдать.
const container = document.querySelector('#resize-container');

// Создайте ResizeObserver, который будет отслеживать изменения размера элемента.
const observer = new ResizeObserver(entries => {
  // Наблюдатель запускается немедленно, поэтому вы можете задать начальный текст.
  // Обычно в массиве будет только одна запись - первый элемент.
  // Это интересующий вас элемент.
  container.textContent = `My width is ${entries[0].contentRect.width}px`;
});

// Начните наблюдать за элементом.
observer.observe(container);
```

Обсуждение

ResizeObserver выполняет обратный вызов, который вы передаете каждый раз, когда изменяется размер элемента. Наблюдатель также вызывает его изначально при первом обращении к элементу.

Функция обратного вызова запускается с массивом объектов ResizeObserverEntry. В нашем случае вы наблюдаете только за одним элементом, а значит, будет всего лишь одна запись. Объект entry обладает несколькими свойствами, включая contentRect, который определяет ограничивающую ширину элемента. Отсюда вы можете получить значение ширины.

В результате при изменении размера элемента обратный вызов наблюдателя форматирует его текст на текущую ширину.



Будьте осторожны при работе с ResizeObserver. Убедитесь, что код в вашей функции обратного вызова не запустит observer повторно. Такой обратный вызов может стать причиной бесконечного цикла обратных вызовов ResizeObserver. Это может произойти, если вы измените элемент в рамках обратного вызова, в результате чего снова потребуется изменение его размера.

6.6. Применение перехода в момент появления элемента в поле зрения

Задача

У вас есть контент, который изначально не отображается. Когда контент попадает в область просмотра, вы хотите отобразить его с анимированным переходом. Например, когда изображение прокручивается в поле зрения, вам нужно, чтобы оно появлялось постепенно, изменяя свою непрозрачность.

Решение

Используйте `IntersectionObserver`, чтобы отслеживать появление элемента в поле зрения. Когда это произойдет, примените анимированный переход (пример 6.7).

Пример 6.7. Все изображения на странице появляются при прокрутке в режиме просмотра

```
const observer = new IntersectionObserver(entries => {
  // В каждой строке несколько изображений, поэтому есть несколько элементов.
  entries.forEach(entry => {
    // Как только элемент станет частично видимым, примените анимированный переход.
    if (entry.isIntersecting) {
      // Изображение становится видимым на 25%, начните постепенный переход.
      entry.target.style.opacity = 1;
      // Больше нет необходимости наблюдать за этим элементом.
      observer.unobserve(entry.target);
    }
  });
}, { threshold: 0.25 }); // Срабатывает, когда изображение становится видимым на 25%

// Обратите внимание на все изображения на странице. Наблюдение будет вестись только
// за изображениями с именем класса animate, поскольку вы можете не захотеть
// следить за всеми изображениями на странице.
document.querySelectorAll('img.animate').forEach(image => {
  observer.observe(image);
});
```

Обсуждение

В этом рецепте используется опция порога `IntersectionObserver`. По умолчанию наблюдатель запускается, как только элемент становится видимым (порог `threshold` равен `0`). Нам это не подходит, потому изображение должно быть видно достаточно

хорошо, чтобы пользователь заметил переход. Если установить пороговое значение равным `0.25`, наблюдатель не будет выполнять обратный вызов, пока изображение не станет видимым хотя бы на 25%.

Функция обратного вызова также проверяет, действительно ли изображение пересекает границу области видимости, т. е. стало ли оно хотя бы частично видимым. Это необходимо, потому что, когда наблюдатель начинает следить за элементом, запускается немедленная проверка вхождения элемента в область видимости и его визуализация. Изображения, находящиеся за пределами экрана, еще не пересекают границу, поэтому такая проверка предотвращает их преждевременное отображение.

Если элемент входит в область видимости, вы можете задать новые стили, которые запускают анимацию или переход. Например, функция обратного вызова может установить непрозрачность изображения равной 1. Для того чтобы этот эффект сработал, вам необходимо предварительно установить значение непрозрачности равным 0 и задать свойство `transition: opacity` (пример 6.8).

Пример 6.8. Стилизация изображений для плавного появления

```
img.animate {
  opacity: 0;
  transition: opacity 500ms;
}
```

При использовании этого стиля изображения становятся невидимыми. Когда функция обратного вызова `observer` устанавливает непрозрачность равной 1, переход вступает в силу, и вы увидите проявление изображения.

Но анимация должна выполниться только один раз, поэтому, как только изображение станет видимым, вам больше не нужно за ним наблюдать. Вы можете выполнить очистку, вызвав `observer.unonbserve` и передав элемент, чтобы прекратить наблюдение.

6.7. Использование режима бесконечной прокрутки

Задача

Вы хотите автоматически загружать больше данных, когда пользователь прокручивает список до конца, без необходимости нажимать кнопку **Load More** (Загрузить больше).

Решение

Поместите элемент в конец прокручиваемого списка и наблюдайте за ним с помощью `IntersectionObserver`. Когда элемент начнет появляться в области видимости, загрузите дополнительные данные (пример 6.9).

Пример 6.9. Использование IntersectionObserver при бесконечной прокрутке

```
/**
 * Наблюдает за элементом-заполнителем с помощью IntersectionObserver.
 * Когда заполнитель становится видимым, загружается больше данных.
 *
 * @param placeholder - элемент-заполнитель
 * @param loadMore - функция, которая загружает больше данных
 */
function observeForInfiniteScroll(placeholder, loadMore) {
  const observer = new IntersectionObserver(entries => {
    // Если заполнитель становится видимым, это означает, что пользователь
    // прокрутил список до конца. Значит, пришло время загрузить
    // дополнительные данные.
    if (entries[0].isIntersecting) {
      loadMore();
    }
  });

  observer.observe(placeholder);
}
```

Обсуждение

Элемент-заполнитель может содержать надпись **Загрузить больше** (Load More) или может быть визуально скрыт. IntersectionObserver отслеживает элемент-заполнитель. Как только он попадает в область просмотра, функция обратного вызов начинает загружать дополнительные данные. Применяя этот прием, пользователь может продолжать прокручивать страницу до тех пор, пока не достигнет конца данных.

Можно также использовать заполнитель для запуска загрузки. Когда пользователь переходит к нижней части списка, запуская новый запрос, он видит спиннер во время загрузки новых данных. Так и должно быть, потому что при пороговом значении по умолчанию равном 0,0 наблюдатель запускается непосредственно перед тем, как пользователь прокрутит страницу достаточно далеко, чтобы увидеть спиннер. К этому времени данные уже загружаются, так что спиннер здесь уместен.

Когда наблюдатель впервые начинает наблюдение, немедленно запускается функция обратного вызова. Если список пуст, то отображается заполнитель, который запускает код для загрузки первой страницы данных.

ГЛАВА 7
Формы

7.0. Введение

Формы собирают данные, вводимые пользователем, которые затем отправляются на удаленный URL-адрес или в конечную точку API. Современные браузеры имеют множество встроенных типов форм для ввода текста, чисел, цветов и многого другого. Форма — это один из основных способов получения информации от пользователя.

FormData

FormData API предоставляет модель данных для доступа к данным формы. Так вы избавляетесь от необходимости искать отдельные элементы DOM и получать их значения.

Более того, если у вас есть объект `FormData`, вы можете передать его непосредственно в Fetch API для отправки формы. Перед отправкой вы можете изменить или дополнить данные непосредственно в объекте `FormData`.

Валидация

Для того чтобы пользователи не отправляли некорректные данные, вы можете (и должны) для своих форм добавить проверку на стороне клиента. Это может быть что-то простое, например пометка поля как обязательного для заполнения, или более сложная логика проверки, которая включает согласование нескольких значений формы или вызов API.

В прошлом для проведения валидации формы разработчику обычно требовалось обратиться к библиотеке JavaScript. Это могло вызвать головную боль из-за дублирования данных. Дублирование существует в данных формы и объекте в памяти, используемом библиотекой проверки.

В HTML5 добавлены дополнительные встроенные опции валидации, такие как:

- пометка поля как обязательного для заполнения;

- указание минимального и максимального значений в числовом поле;
- использование регулярного выражения для проверки ввода в поле.

Эти опции используются в качестве атрибутов элементов `<input>`.

Браузер отображает основные сообщения об ошибках проверки (рис. 7.1), но стиль может не сочетаться с дизайном вашего приложения. Вы можете использовать Constraint Validation API для оценки встроенных результатов проверки, а также выполнить пользовательскую логику и настроить свои собственные сообщения о результатах валидации.

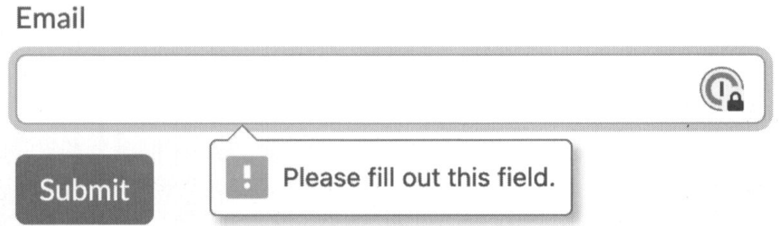


Рис. 7.1. Встроенное валидационное сообщение в браузере Chrome

Для того чтобы проверить заполнение формы, вы вызываете ее метод `checkValidity`. Проверяются все поля в форме. Если все поля действительны, функция `checkValidity` возвращает значение `true`. Если одно или несколько полей являются недопустимыми, функция `checkValidity` возвращает значение `false`, при этом каждое недопустимое поле запускает событие `invalid`. Вы также можете проверить определенный элемент, вызвав функцию `checkValidity` для конкретного поля формы.

Каждое поле формы имеет объект `validity`, который отражает текущее состояние валидности. Он имеет логическое значение `valid`, которое указывает на общее состояние допустимости формы. Этот объект также имеет дополнительные флаги, указывающие на природу ошибки.

7.1. Заполнение поля формы из локального хранилища

Задача

Вы желаете сохранить значение поля формы в локальном хранилище. Например, вы можете захотеть запомнить имя пользователя, введенное в форму входа в систему.

Решение

При отправке формы используйте объект `FormData`, чтобы получить значение поля и сохранить его в локальном хранилище (пример 7.1). Затем при первой загрузке страницы проверьте, сохранилось ли значение. Если вы нашли значение, заполните поле формы.

Пример 7.1. Запоминание поля username (имя пользователя)

```
const form = document.querySelector('#login-form');

const username = localStorage.getItem('username');
if (username) {
  form.elements.username.value = username;
}

form.addEventListener('submit', event => {
  const data = new FormData(form);
  localStorage.setItem('username', data.get('username'));
});
```

Обсуждение

Когда вы передаете форму конструктору `FormData`, ее элементы заполняются введенными пользователем значениями. Затем вы можете вызвать метод `get` для извлечения нужного поля и установки его значения его в локальном хранилище.

Заполнение формы при загрузке немного отличается. Объект `FormData` не синхронизируется с текущими значениями формы; скорее, он включает значения формы на момент создания объекта `FormData`. Верно и обратное: если вы зададите новое значение в объекте `FormData`, оно не будет обновляться в самой форме. Учитывая это, имейте в виду, что объект `FormData` не поможет при заполнении формы. В примере 7.1 используется свойство `elements` формы для поиска поля `username` и установки его значения.

Поиск полей формы с помощью свойства *elements*

Свойство `elements` формы позволяет искать поля формы по их именам. Каждое поле в этой форме, имеющее свойство `name`, имеет соответствующее свойство в `form.elements`. Например, вы можете найти входные данные с именем `username`, обратившись к `form.elements.username`. Обратите внимание, что вам нужно указать атрибут `name` поля, а не его идентификатор `id`.

7.2. Отправка формы с помощью Fetch и FormData API

Задача

Вы хотите отправить форму, используя Fetch API. Возможно, вы захотите сделать это, чтобы добавить в отправляемую форму дополнительную информацию, которую браузер не добавляет, или потому, что для отправки формы может потребоваться токен API, который хранится в памяти, а не вводится в форму.

Другая причина, по которой вы, возможно, захотите это сделать, заключается в том, чтобы предотвратить перенаправление браузера на новую страницу или полное обновление страницы.

Решение

Создайте объект `FormData`, содержащий данные для отправки. Добавьте необходимые дополнительные данные, затем отправьте форму с помощью Fetch API (пример 7.2).

Пример 7.2. Добавление данных с помощью FormData API

```
// В реальном приложении токен API хранился бы где-то и не был бы
// жестко закодирован подобным образом.
const apiToken = 'aBcD1234EfGh5678IjKlM';

form.addEventListener('submit', event => {
  // Важно: отключите автоматическую отправку формы браузером.
  event.preventDefault();

  // Настройте объект FormData и добавьте к нему токен API.
  const data = new FormData(event.target);
  data.set('apiToken', apiToken);

  // Используйте Fetch API для отправки этого объекта FormData в конечную точку.
  fetch('/api/form', {
    method: 'POST',
    body: data
  });
});
```

Обсуждение

Обычно, когда пользователь нажимает кнопку отправки, браузер получает данные формы и отправляет их. Вы же не хотите именно таких действий, потому что вам нужно добавить токен API.

Первое, что делает обработчик отправки, — это вызывает `preventDefault` по событию отправки. Браузер останавливает выполнение действий по отправке по умолчанию, чтобы вы могли использовать свою пользовательскую логику. По умолчанию выполняется полное обновление страницы, а это, вероятно, вам не нужно.

Вы можете создать объект `FormData`, передав объект `form` конструктору `FormData`. Результирующий объект будет содержать существующие данные формы, после чего вы сможете добавить дополнительные данные, такие как токен API.

Наконец, вы можете передать объект `FormData` в качестве тела запроса POST, используя Fetch API. При отправке формы таким образом, текст *не форматируется* в JSON-строку; скорее, браузер отправляет его с типом содержимого `multipart/form-data`.

Рассмотрим объект, представляющий данные вашей формы:

```
{
  username: 'john.doe',
  apiToken: 'aBcD1234EfGh5678IjKlM'
}
```

Эквивалентный текст запроса выглядит примерно так:

```
------WebKitFormBoundaryl6AuUOn9EbuYe9XO
Content-Disposition: form-data; name="username"

john.doe
------WebKitFormBoundaryl6AuUOn9EbuYe9XO
Content-Disposition: form-data; name="apiToken"

aBcD1234EfGh5678IjKlM
------WebKitFormBoundaryl6AuUOn9EbuYe9XO--
```

7.3. Отправка формы в формате JSON

Задача

Вы хотите отправить форму в конечную точку, которая ожидает данные в формате JSON.

Решение

Используйте FormData API для преобразования данных формы в объект JavaScript и примените Fetch API для отправки их в формате JSON (пример 7.3).

Пример 7.3. Отправка формы в формате JSON с помощью Fetch

```
form.addEventListener('submit', event => {
  // Важно: отключите автоматическую отправку формы браузером.
  event.preventDefault();

  // Создайте новый FormData, содержащий данные этой формы, затем добавьте
  // каждую пару "ключ/значение" в тело ответа.
  const data = new FormData(event.target);
```

```
  const body = {};
  for (const [key, value] of data.entries()) {
    body[key] = value;
  }

  // Отправьте тело формы в формате JSON в конечную точку.
  fetch('/api/form', {
    method: 'POST',

    // Объект должен быть преобразован в строку JSON.
    body: JSON.stringify(body),

    // Сообщите серверу, что вы отправляете JSON.
    headers: {
      'content-type': 'application/json'
    }
  })
    .then(response => response.json())
    .then(body => console.log('Got response:', body));
});
```

Обсуждение

Представленный подход аналогичен отправке объекта `FormData` напрямую. Единственное отличие в том, что вы преобразуете данные формы в JSON и отправляете его вместе с правильным заголовком `Content-Type`.

Вы можете выполнить преобразование, создав новый пустой объект и проделав итерацию по парам "ключ/значение" в `FormData`. Каждая пара копируется в объект.

Недостатком рассмотренного подхода является то, что вы не можете использовать его с `FormData`, имеющим несколько значений, привязанных к одному и тому же ключу. Это происходит, когда у вас есть группа флажков под одним и тем же именем, т. е. имеется несколько записей с одним и тем же ключом.

Можно улучшить преобразование, чтобы обнаружить этот случай, и задать массив значений, как показано в примере 7.4.

Пример 7.4. Обработка значений формы в виде массива

```
/**
 * Преобразует данные формы в объект, который можно отправить в формате JSON.
 * @param form - элемент формы
 * @returns объект, содержащий все сопоставленные ключи и значения
 */
```

```
function toObject(form) {
  const data = new FormData(form);
  const body = {};

  for (const key of data.keys()) {
    // Возвращает массив всех значений, привязанных к заданному ключу.
    const values = data.getAll(key);

    // Если в массиве есть только один элемент, установите этот элемент напрямую.
    if (values.length === 1) {
      body[key] = values[0];
    } else {
      // В противном случае установите сразу все элементы массива
      body[key] = values;
    }
  }

  return body;
}
```

В примере 7.4 используется функция `getAll` из `FormData`, которая возвращает массив, содержащий все значения, привязанные к данному ключу. Так можно собрать все значения для данной группы флажков в массив.

`getAll` всегда возвращает массив. Если есть только одно значение, значит, это массив с одним элементом. `toObject` проверяет этот сценарий, и если массив содержит лишь один элемент, он использует этот элемент в качестве единственного значения в результирующем объекте. В противном случае используется массив значений.

7.4. Создание обязательного поля формы

Задача

Вы хотите, чтобы поле формы обязательно имело значение. Если оно останется пустым, это приведет к ошибке при валидации.

Решение

Используйте обязательный атрибут для элемента `<input>` (пример 7.5).

Пример 7.5. Обязательное поле для заполнения

```
<label for="username">Имя пользователя</label>
<input type="text" name="username" id="username" required> ❶
```

❶ Атрибут required значения не имеет.

Обсуждение

Если поле помечено как required (обязательное), оно должно содержать значение. Если поле пустое, его свойству validity.valid будет присвоено значение false, а свойству validity.valueMissing — значение true. Обязательное поле считается пустым только в том случае, если значение представляет собой пустую строку. Пробелы не отбрасываются, поэтому допустимым считается значение, состоящее из нескольких пробелов.

7.5. Ограничения при вводе числа

Задача

Вы хотите указать диапазон допустимых значений для ввода числового значения (<input type="number">).

Решение

Используйте свойства min и max, чтобы указать допустимый диапазон (пример 7.6). Эти значения являются включающими, что означает, что допустимы как минимальные, так и максимальные значения.

Пример 7.6. Указание диапазона для числового поля

```
<label for="quantity">Количество</label>
<input type="number" name="quantity" id="quantity" min="1" max="10">
```

Обсуждение

Если введенное числовое значение ниже минимального или выше максимального, то свойству validity.valid поля ввода присваивается значение false. Если значение ниже минимального, то устанавливается флаг допустимости rangeUnderflow. Аналогично, если оно превышает максимальное значение, устанавливается флаг rangeOverflow.

Когда вы задаете тип number для элемента <input>, браузер добавляет кнопки-стрелки вверх и вниз управления счетчиком (спиннер), которые можно нажимать,

чтобы увеличивать и уменьшать значение. Элемент управления счетчиком ограничен минимальным и максимальным значениями — он не позволит уменьшать значение, если оно уже достигло минимума, или увеличивать значение, если оно уже достигло максимума. Правда, пользователь по-прежнему волен вводить в поле любое значение. Он может ввести число, выходящее за пределы допустимого диапазона, после чего состояние допустимости устанавливается соответствующим образом.

Если вы хотите более точно контролировать допустимые значения, также можете указать значение шага. Однако это ограничивает допустимые значения, поскольку приращение должно быть кратно шагу. Представьте входные данные с минимальным значением 0, максимальным значением 4 и шагом 2. Единственными допустимыми значениями для этого поля будут 0, 2 и 4.

7.6. Определение шаблона валидации

Задача

Вы хотите ограничить значение текстового поля таким образом, чтобы оно соответствовало определенному шаблону.

Решение

Используйте атрибут `pattern` для ввода регулярного выражения (пример 7.7). Поле считается недействительным, если его значение не соответствует регулярному выражению.

Пример 7.7. Ограничение поля только буквенно-цифровыми символами

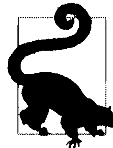
```
<label for="username">Введите имя пользователя</label>
<input type="text" pattern="[A-Za-z0-9]+" id="username" name="username">
```

Поле `username` является недопустимым, если оно содержит что-либо, кроме буквенно-цифровых символов[1]. Если поле является недопустимым, устанавливается флаг `patternMismatch`.

Обсуждение

Это гибкий вариант проверки, уступающий только использованию вашей собственной пользовательской логики проверки (см. рецепт 7.8).

[1] Если в поле формы нужно вводить символы русского алфавита, то атрибут шаблона должен быть таким: `pattern="[А-Яа-я0-9]+"`. — *Прим. ред.*



Создание регулярного выражения для проверки URL-адресов или адресов электронной почты может оказаться непростой задачей. Для того чтобы справиться с такими ситуациями, вы можете задать для входного атрибута `type` значение `url` или `email`, и браузер подтвердит, что это поле является допустимым URL-адресом или адресом электронной почты для вас.

7.7. Валидация формы

Задача

Вы хотите управлять процессом проверки формы и отображать собственные сообщения об ошибках в пользовательском интерфейсе.

Решение

Используйте Constraint Validation API и событие `invalid` для обнаружения и пометки недопустимых полей.

Существует множество способов проверки. Некоторые веб-сайты слишком нетерпеливы и выдают сообщение об ошибке до того, как пользователь успевает ввести значение. Рассмотрим ввод типа `email`, который считается недействительным до тех пор, пока не будет введен допустимый адрес электронной почты. Если проверка выполняется немедленно, пользователь видит сообщение об ошибке с неверным адресом электронной почты до того, как закончит его набирать.

Для того чтобы избежать преждевременных сообщений, приведенный здесь подход проверяет поле только при выполнении двух условий:

- если нажата кнопка отправки формы;
- если на поле был фокус, а затем он был потерян. Считается, что это поле было "затронуто".

Сначала вам нужно отключить встроенный пользовательский интерфейс проверки браузера, добавив в форму атрибут `novalidate`, как показано в примере 7.8.

Пример 7.8. Отключение пользовательского интерфейса проверки браузера

```
<form id="my-form" novalidate>
  <!-- Form elements go here -->
</form>
```

Для каждого поля требуется элемент-заполнитель, содержащий сообщение об ошибке, как показано в примере 7.9.

Пример 7.9. Добавление элементов-заполнителей для сообщений об ошибках

```
<div>
  <label for="email">Email</label>
  <input required type="email" id="email" name="email">
  <div class="error-message" id="email-error"></div>
</div>
```

В этом примере сообщение об ошибке связано с полем ввода при помощи идентификатора. Поле с идентификатором `email` содержит сообщение об ошибке с идентификатором `email-error`, поле `name` должно содержать сообщение об ошибке с `name-error` и т. д.

При таком подходе к проверке каждый элемент формы прослушивает три события.

`invalid`

Срабатывает, когда форма проверена, а поле помечено как недопустимое. При этом выводится сообщение об ошибке.

`input`

Срабатывает при изменении значения в поле. При необходимости выполняется повторная проверка, и сообщение об ошибке отменяется, если поле становится допустимым.

`blur`

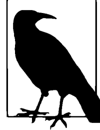
Запускается, когда поле теряет фокус. При этом устанавливается атрибут `data-should-validate`, который помечает поле, в которое что-то вводили, после чего оно проверяется в обработчике событий `input`.

Код проверки приведен в примере 7.10.

Пример 7.10. Настройка проверки поля формы

```
/**
 * Добавляет необходимых прослушивателей событий для участия в валидации формы.
 * Осуществляет настройку и очистку сообщений об ошибках в зависимости
 * от состояния проверки.
 * @param element - элемент для проверки
 */
function addValidation(element) {
  const errorElement = document.getElementById(`${element.id}-error`);

  /**
   * Запускается, когда форма проверена, а данные в поле недопустимы.
   * Задает сообщение об ошибке и стиль, а также устанавливает флаг shouldValidate.
   */
```

```
  element.addEventListener('invalid', () => {
    errorElement.textContent = element.validationMessage;
    element.dataset.shouldValidate = true;
  });

  /**
   * Запускается, когда пользователь вводит данные в поле.
   * Если установлен флаг shouldValidate, программа повторно проверит правильность
   * поля и удалит сообщение об ошибке, если его содержимое станет допустимым.
   */
  element.addEventListener('input', () => {
    if (element.dataset.shouldValidate) {
      if (element.checkValidity()) {
        errorElement.textContent = '';
      }
    }
  });

  /**
   * Запускается, когда поле теряет фокус, с применением флага shouldValidate.
   */
  element.addEventListener('blur', () => {
    // В это поле что-то вводили; теперь оно будет проверяться при
    // последующих событиях input.
    // Атрибуту data-should-validate элемента input
    // присваивается значение true.
    element.dataset.shouldValidate = true;
  });
}
```



В этом примере прослушивается событие ввода. Если ваша форма содержит флажки или переключатели, вам может потребоваться прослушать событие изменения для этих элементов, в зависимости от браузера. Смотрите статью о событиях ввода из MDN (**https://oreil.ly/cFIjY**):

> "Согласно спецификации HTML Living Standard, для элементов `<input>` с `type=checkbox` или `type=radio` событие ввода должно срабатывать всякий раз, когда пользователь переключает элемент управления. Однако исторически так было не всегда. Проверьте совместимость или вместо этого используйте событие `change` для элементов этих типов".

Для того чтобы завершить базовую проверку, добавьте прослушивателей в поля формы, прослушайте событие `submit` формы и запустите валидацию (пример 7.11).

Пример 7.11. Запуск валидации формы

```
// Предположим, что в форме есть два элемента ввода: name и email.
addValidation(form.elements.name);
addValidation(form.elements.email);

form.addEventListener('submit', event => {
  event.preventDefault();
  if (form.checkValidity()) {
    // Валидация пройдена, отправьте форму.
  }
});
```

Обсуждение

Этот код устанавливает хорошую базовую структуру валидации, которая обрабатывает встроенную проверку браузера. Перед отправкой формы вызывается функция `checkValidity`, которая запускает проверку всех входных данных внутри формы. Браузер запускает событие `invalid` для любого элемента ввода, который не прошел проверку. Для того чтобы справиться с этим, вы можете прослушать событие `invalid` в самих элементах ввода. После этого вы можете отобразить соответствующее сообщение об ошибке.

Как только пользователь обнаружит ошибки валидации, нужно удалить сообщения о них, если введенные в поля данные будут исправлены. Вот почему `addValidation` прослушивает событие `input` — оно включается, как только пользователь вводит что-либо в поле ввода. После этого вы можете немедленно перепроверить правильность введенных данных. Если введенное значение теперь допустимо (`checkValidity` возвращает значение `true`), вы можете удалить сообщение об ошибке. Входные данные повторно проверяются только в том случае, если атрибуту `data-should-validate` присвоено значение `true`. Этот атрибут добавляется, когда при отправке формы происходит сбой проверки или когда элемент теряет фокус. Такие действия предотвращают появление ошибок валидации до того, как пользователь закончит вводить текст. Как только поле теряет фокус, начинается повторная проверка при каждом изменении.

7.8. Применение пользовательской логики валидации

Задача

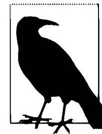
Вы хотите выполнить проверку достоверности, которая не поддерживается Constraint Validation API. Например, нужно проверить, что поле пароля и поле подтверждения пароля имеют одинаковые значения.

Решение

Реализуйте логику пользовательской проверки, прежде чем вызывать функцию `checkValidity` формы. Если пользовательская проверка терпит неудачу, вызовите метод `setCustomValidity`, чтобы сформировать соответствующее сообщение об ошибке. Если проверка прошла успешно, удалите все ранее сгенерированные сообщения с результатами проверки (пример 7.12).

Пример 7.12. Использование пользовательской проверки

```
/**
 * Пользовательская функция проверки, которая гарантирует,
 * что поля password и confirmPassword имеют одинаковые значения.
 * @param form - форма, содержащая два поля
 */
function validatePasswordsMatch(form) {
  const { password, confirmPassword } = form.elements;

  if (password.value !== confirmPassword.value) {
    confirmPassword.setCustomValidity('Пароли не совпадают.');
  } else {
    confirmPassword.setCustomValidity('');
  }
}

form.addEventListener('submit', event => {
  event.preventDefault();
  validatePasswordsMatch(form);

  if (form.checkValidity()) {
    // Валидация пройдена, отправьте форму.
  }
});
```



Если вы используете встроенный в браузер пользовательский интерфейс валидации, вам необходимо вызвать метод `reportValidity` поля формы после составления пользовательского сообщения о проверке. Если вы сами обрабатываете пользовательский интерфейс проверки, в этом нет необходимости, но убедитесь, что сообщение об ошибке отображается в соответствующем месте.

Обсуждение

При вызове функции `setCustomValidity` для элемента с непустой строкой такой элемент теперь считается недопустимым.

Функция `validatePasswordsMatch` проверяет значения полей `password` и `confirmPassword`. Если они не совпадают, программа вызывает `setCustomValidity` для поля `confirmPassword`, чтобы выдать сообщение об ошибке проверки. Если они совпадают, она присваивает ему значение пустой строки, которая снова помечает поле как допустимое.

Обработчик отправки формы вызывает `validatePasswordsMatch` перед выполнением встроенной проверки. Если проверка `validatePasswordsMatch` завершается неудачно, и пользовательская проверка достоверности установлена, `form.checkValidity` завершается ошибкой, и в поле `confirmPassword` срабатывает событие `invalid`, как и в любом другом недопустимом элементе.

7.9. Проверка группы флажков

Задача

Вы хотите убедиться в том, что хотя бы один флажок в группе установлен. Установка атрибута `required` для флажков здесь не поможет, поскольку он применяется только к отдельным входным данным, а не к группе. Браузер проверяет, установлен ли флажок для этого ввода, и выдает ошибку проверки, даже если установлены другие флажки в группе.

Решение

Используйте объект `FormData`, чтобы получить массив всех выбранных флажков, и выдайте пользовательское сообщение об ошибке проверки, если массив пуст.

При выполнении пользовательской проверки используйте метод `getAll` из `FormData`, чтобы получить массив значений выбранных флажков (пример 7.13). Если массив пуст, флажки не установлены, значит, случай является ошибкой валидации.

Пример 7.13. Проверка группы флажков

```
function validateCheckboxes(form) {
  const data = new FormData(form);

  // Для того чтобы избежать установки ошибки проверки для нескольких элементов,
  // выберите первый флажок и используйте его для приостановки вывода сообщения
  // о проверке группы.
  const element = form.elements.option1;
```

```
  if (!data.has('options')) {
    element.setCustomValidity('Пожалуйста, выберите хотя бы один вариант.');
  } else {
    element.setCustomValidity('');
  }
}
```

Для того чтобы сохранить состояние проверки для всей группы в одном месте, установите пользовательское сообщение о проверке только для первого флажка (допустим, его имя — option1). Этот первый флажок служит контейнером для сообщения о проверке группы, что необходимо, поскольку вы можете устанавливать сообщения о проверке только для фактических элементов <input>.

Затем прослушайте события invalid и change. В случае события invalid отобразите сообщение об ошибке. В случае change (когда установлен флажок), выполните пользовательскую проверку и удалите сообщение об ошибке, если проверка прошла успешно (пример 7.14).

Пример 7.14. Установка проверки флажка

```
/**
 * Добавляет необходимые прослушиватели событий к элементу для участия в проверке
 * формы. Обрабатывает настройку и удаление сообщений об ошибках в зависимости
 * от состояния проверки.
 * @param element - входной элемент для проверки
 * @param errorId - ID плейсхолдера элемента, который покажет сообщение об ошибке
 */
function addValidation(element, errorId) {
  const errorElement = document.getElementById(errorId);

  /**
   * Запускается, когда форма проверена, а содержимое поля недопустимо.
   * Задает сообщение об ошибке и его стиль.
   */
  element.addEventListener('invalid', () => {
    errorElement.textContent = element.validationMessage;
  });

  /**
   * Запускается при вводе пользователем данных в поле.
   * Программа перепроверит правильность поля и удалит сообщение об ошибке,
   * если содержимое поля станет допустимым.
   */
```

```
  element.addEventListener('change', () => {
    validateCheckboxes(form);
    if (form.elements.option1.checkValidity()) {
      errorElement.textContent = '';
    }
  });
}
```

Наконец, добавьте подтверждение в каждое поле checkbox и вызовите функцию validateCheckboxes, прежде чем проверять правильность формы. В примере 7.15 предполагается, что у вас есть элемент с идентификатором checkbox-error. Если произойдет ошибка проверки флажка, для этого элемента будет установлено соответствующее сообщение.

Пример 7.15. Проверка флажка формы

```
addValidation(form.elements.option1, 'checkbox-error');
addValidation(form.elements.option2, 'checkbox-error');
addValidation(form.elements.option3, 'checkbox-error');

form.addEventListener('submit', event => {
  event.preventDefault();
  validateCheckboxes(form);
  console.log(form.checkValidity());
});
```

Обсуждение

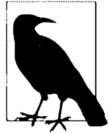
Указание атрибута required для флажков в группе не даст желаемого эффекта. Атрибут хорош для одного флажка, например, требующего, чтобы пользователь принял лицензионное соглашение. Однако использование его в группе сделало бы обязательной установку каждого отдельного флажка, и проверка формы завершилась бы неудачей, если бы не были отмечены они *все*. Поскольку для "группы флажков" нет HTML-элемента, вам потребуется немного поработать, чтобы добиться желаемого поведения.

В этом примере первый флажок в группе выбирается в качестве "контейнера" для валидационного сообщения. Когда пользователь устанавливает флажок, браузер вызывает обработчик изменений и проверяет, установлен ли какой-либо из флажков. Если массив выбора пуст, это означает ошибку. Настраиваемое значение сообщения о допустимости всегда устанавливается только для первого флажка. Это необходимо для того, чтобы сообщение всегда отображалось и при необходимости скрывалось.

Давайте посмотрим, что произойдет, если вместо этого вы примените настраиваемое значение валидации к флажку, состояние которого изменяется.

Если никакие опции не отмечены и пользователь отправляет форму, каждый флажок имеет пользовательское сообщение об ошибке проверки допустимости. Все три опции недействительны. Если вы затем установите один из флажков, сработает событие `change` флажка и проверит флажки всей группы. Теперь выбрана опция, и она удаляет пользовательское сообщение о проверке допустимости. Однако другие флажки по-прежнему находятся в состоянии ошибки. По сути, теперь это эквивалентно установке атрибута `required` для всех флажков.

Вы могли бы обойти это, установив сообщение о проверке во *всех* флажках с помощью функции `validateCheckboxes`, но гораздо проще выбрать один из них и использовать его в качестве целевого объекта для всех пользовательских сообщений о проверке. У группы в целом есть один элемент сообщения об ошибке, который и заполняется.



Поскольку в этом примере используется ваше собственное сообщение о проверке, не забудьте включить в форму атрибут `novalidate`, чтобы избежать отображения интерфейсного сообщения браузера по умолчанию вместе с вашим пользовательским сообщением об ошибке проверки.

7.10. Асинхронная проверка поля формы

Задача

Ваша пользовательская логика проверки требует выполнения асинхронной операции, такой как создание сетевого запроса. Например, в форме регистрации пользователя есть поле для ввода пароля. Форма регистрации должна вызывать API для проверки соответствия введенного пароля стандартам надежности пароля.

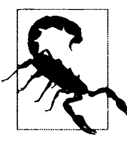
Решение

Выполните сетевой запрос, затем отправьте пользовательское сообщение о допустимости значения. Эти действия реализуйте с помощью функции, которая возвращает `Promise`. В обработчике отправки формы дождитесь выполнения этого промиса, прежде чем в форме вызывать `checkValidity`. Если в коде асинхронной проверки установлено пользовательское значение сообщения о допустимости, проверка формы, инициируемая `checkValidity`, обрабатывает его.

Пример 7.16 содержит саму логику проверки. Здесь вызывается API для проверки надежности пароля и соответствующим образом устанавливается пользовательское сообщение о допустимости.

Пример 7.16. Выполнение асинхронной проверки надежности пароля

```
/**
 * Вызывает API для проверки соответствия пароля требованиям к надежности.
 * @param form - форма, содержащая поле пароля
 */
async function validatePasswordStrength(form) {
  const { password } = form.elements;
  const response = await fetch(`/api/password-strength?password=${password.value}`);
  const result = await response.json();

  // Как и прежде, не забудьте вызвать reportValidity в поле пароля,
  // если вы используете встроенный пользовательский интерфейс проверки браузера.
  if (result.status === 'error') {
    password.setCustomValidity(result.error);
  } else {
    password.setCustomValidity('');
  }
}
```



Убедитесь, что вы отправляете пароли только по защищенному соединению (HTTPS). В противном случае вы отправляете пароль пользователя в виде обычного текста, а это опасная практика.

Поскольку функция помечена как асинхронная, она возвращает промис. Вам просто нужно дождаться этого промиса в обработчике отправки формы, как показано в примере 7.17.

Пример 7.17. Обработчик отправки асинхронной формы

```
form.addEventListener('submit', async event => {
  event.preventDefault();
  await validatePasswordStrength(form);
  console.log(form.checkValidity());
});
```

При отправке это поле помечается как `invalid`, если пароль не соответствует требованиям. Вы можете повторно запустить проверку при изменении поля, только на этот раз вы будете выполнять ее при событии потери фокуса (`blur`), а не при вводе данных, как при синхронной пользовательской проверке (пример 7.18).

Пример 7.18. Повторная валидация при потере фокуса

```
form.elements.password.addEventListener('blur', async event => {
  const password = event.target;
  const errorElement = document.getElementById('password-error');
  if (password.dataset.shouldValidate) {
    await validatePasswordStrength(form);
    if (password.checkValidity()) {
      errorElement.textContent = '';
      password.classList.remove('border-danger');
    }
  }
});
```

Обсуждение

Если бы вы выполнили эту проверку для события ввода, вы бы отправляли сетевой запрос каждый раз, когда пользователь нажимал бы клавишу. Событие потери фокуса `blur` откладывает повторную проверку до тех пор, пока поле не потеряет фокус. Оно снова вызывает API валидации и проверяет новое состояние допустимости.

Вы также можете использовать отложенную (debounced — без дребезга) версию функции проверки. Это приведет к повторной проверке по событию ввода, но только после того, как пользователь прекратит вводить текст на некоторое время.

В статье от freeCodeCamp (**https://oreil.ly/kLRJa**) подробно рассказывается о том, как создать отложенную функцию валидации. Также доступны пакеты NPM, которые позволят создать отложенную версию функции.

ГЛАВА 8

API веб-анимации

8.0. Введение

В современных веб-браузерах существует несколько различных способов анимации элементов. В *главе 1* приведен пример использования API `requestAnimationFrame` для ручной анимации элемента (см. рецепт 1.6). Так вы можете контролировать ситуацию, но не бесплатно. Для этого требуется отслеживать временны́е метки и рассчитывать частоту кадров, а вы должны контролировать каждое постепенное изменение анимации при помощи JavaScript.

Анимация на основе ключевых кадров

В CSS3 появилась анимация ключевых кадров. Вы задаете начальный стиль, конечный стиль и длительность в соответствии с правилами CSS. Браузер автоматически интерполирует, т. е. заполняет, промежуточные кадры анимации. Анимации определяются посредством правила `@keyframes` и используются с помощью свойства `animation`. В примере 8.1 определяется анимация с постепенным появлением элемента.

Пример 8.1. Использование анимации ключевых кадров CSS

```
@keyframes fade {
  from {
  opacity: 0;
 }

  to {
    opacity: 1;
  }
}
```

```
.some-element {
  animation: fade 250ms;
}
```

Плавная анимация начинается с непрозрачности (opacity), равной 0, и заканчивается непрозрачностью, равной 1. При запуске анимации браузер вычисляет и показывает промежуточные кадры в течение 250 миллисекунд. Анимация начинается, как только элемент попадает в DOM или применяется класс `some-element`.

Анимация ключевых кадров с помощью JavaScript

API веб-анимации позволяет использовать анимацию ключевых кадров в вашем коде на JavaScript. В интерфейсе `Element` есть метод `animate`, с помощью которого вы можете определять ключевые кадры и другие параметры анимации. В примере 8.2 показана анимация из примера 8.1 с применением API веб-анимации.

Пример 8.2. Появление элемента с помощью API веб-анимации

```
const element = document.querySelector('.some-element');
element.animate([
  { opacity: 0 },
  { opacity: 1 }
], {
  // Анимировать в течение 250 миллисекунд
  duration: 250
});
```

Результат тот же. Элемент полностью проявляется в течение 250 миллисекунд. В этом случае анимация запускается вызовом `element.animate`.

Объекты анимации

При вызове `element.animate` возвращается объект анимации. Через него вы можете приостановить, возобновить, отменить или даже повернуть анимацию вспять. Он также предоставляет вам `Promise`, который вы можете использовать, чтобы дождаться завершения анимации.

Будьте осторожны и следите за тем, какие свойства вы анимируете. Некоторые из них, например высота или отступы, влияют на макет остальной части страницы; их анимация может вызвать проблемы с производительностью, а анимация, как правило, не будет плавной. Наилучшие свойства для анимации — это `opacity` и `transform`, поскольку они не влияют на макет страницы, их даже можно ускорить с помощью графического процессора системы.

8.1. Применение эффекта "пульсации" при нажатии кнопки

Задача

Вы хотите показать анимацию "пульсации" (ripple) при нажатии на кнопку, начиная с того места внутри кнопки, на которое нажал пользователь.

Решение

Когда кнопка нажата, создайте временный дочерний элемент для "пульсации". Он и будет анимирован.

Сначала создайте несколько стилей для элемента `ripple`. К кнопке также необходимо применить несколько стилей (пример 8.3).

Пример 8.3. Стили для элементов button и ripple

```
.ripple-button {
  position: relative;
  overflow: hidden;
}

.ripple {
  background: white;
  pointer-events: none;
  transform-origin: center;
  opacity: 0;
  position: absolute;
  border-radius: 50%;
  width: 150px;
  height: 150px;
}
```

В обработчике нажатия кнопки динамически создайте новый элемент пульсации `ripple` и добавьте его к кнопке, затем обновите его положение и выполните анимацию (пример 8.4).

Пример 8.4. Выполнение анимации пульсации

```
button.addEventListener('click', async event => {
  // Создайте временный элемент для ripple, задайте его класс и добавьте его к кнопке.
  const ripple = document.createElement('div');
  ripple.className = 'ripple';
```

```
  // Найдите наибольший размер (ширину или высоту) кнопки и
  // используйте его в качестве размера пульсации.
  const rippleSize = Math.max(button.offsetWidth, button.offsetHeight);
  ripple.style.width = `${rippleSize}px`;
  ripple.style.height = `${rippleSize}px`;

  // Отцентрируйте элемент ripple в месте щелчка.
  ripple.style.top = `${event.offsetY - (rippleSize / 2)}px`;
  ripple.style.left = `${event.offsetX - (rippleSize / 2)}px`;

  button.appendChild(ripple);

  // Выполните анимацию пульсации и дождитесь ее завершения.
  await ripple.animate([
    { transform: 'scale(0)', opacity: 0.5 },
    { transform: 'scale(2.5)', opacity: 0 }
  ], {
    // Анимировать в течение 500 миллисекунд.
    duration: 500,
    // Используйте функцию упрощения настройки ease-in.
    easing: 'ease-in'
  }).finished;

  // Все сделано, удалите элемент ripple.
  ripple.remove();
});
```

Обсуждение

Элемент ripple представляет собой круг, размер которого соответствует размеру кнопки. Эффект ряби достигается за счет анимации его непрозрачности и изменения масштаба.

Здесь следует обратить внимание на несколько особенностей стилей элементов. Во-первых, для свойства position самой кнопки задано положение relative. Это делается для того, чтобы при установке абсолютного положения пульсации оно было относительно самой кнопки.

Абсолютное позиционирование CSS

Когда вы устанавливаете для свойства position элемента значение absolute, браузер удаляет его из макета документа и позиционирует относительно ближайшего

элемента-предка, которому уже назначена *позиция*. Элемент считается позиционированным, если для его свойства `position` установлено значение, отличное от значения по умолчанию `static`.

Если у вас есть элемент с абсолютным позиционированием и позиция кажется неправильной, проверьте и убедитесь, что для него используется предок с правильным позиционированием. Правда, невозможно заранее утверждать, что предком будет непосредственный родитель элемента.

Кнопка также имеет свойство `overflow: hidden`. Эта настройка предотвращает видимость эффекта пульсации за пределами кнопки.

Вы также можете заметить, что для пульсации установлена опция `pointer-events: none`. Поскольку пульсация находится внутри кнопки, браузер делегирует все события пульсации кнопке. Это означает, что нажатие на область пульсации запускает новую пульсацию, но ее положение неверно, поскольку оно основано на положении нажатия внутри пульсации, а не внутри кнопки.

Самый простой способ обойти это — установить значение `pointer-events: none`, что заставляет элемент `ripple` игнорировать события нажатия. Если вы нажимаете на рябь во время ее анимации, событие щелчка переходит к кнопке, что необходимо для правильного расположения следующей пульсации.

Далее код пульсации устанавливает значения `top` и `left` так, чтобы центр ряби находился в том месте, где вы только что щелкнули.

Затем пульсация анимируется. Анимация, возвращаемая `ripple.animate`, имеет свойство `finished`, являющееся промисом, которого вы можете дождаться. Как только этот промис будет выполнен, анимация пульсации будет завершена, и вы сможете удалить элемент из DOM.

Если вы нажмете на кнопку во время выполнения эффекта ripple, начнется другая пульсация, и они будут анимироваться вместе — первая анимация не прерывается. С помощью обычной CSS-анимации такого эффекта добиться сложнее.

Функции сглаживания переходов (easing functions)

Функция сглаживания, определяемая свойством `easing`, назначает скорость изменения анимируемых свойств. Их настройка выходит за рамки данной книги, но есть несколько встроенных функций, с которыми необходимо разобраться.

`linear` *(по умолчанию)*

Анимация выполняется с постоянной скоростью.

`ease-out`

Анимация начинается быстрее, затем постепенно замедляется.

`ease-in`

Анимация начинается медленно, затем постепенно ускоряется.

`ease-in-out`

Анимация начинается медленно, ускоряется и под конец снова замедляется.

8.2. Запуск и остановка анимации

Задача

Вы хотите иметь возможность программно запускать или останавливать анимацию.

Решение

Используйте функции паузы и воспроизведения анимации (пример 8.5).

Пример 8.5. Переключение режима воспроизведения анимации

```
/**
 * При наличии анимации переключает ее состояние.
 * Если анимация запущена, она будет приостановлена.
 * Если анимация приостановлена, она будет возобновлена.
*/
function toggleAnimation(animation) {
  if (animation.playState === 'running') {
    animation.pause();
  } else {
    animation.play();
  }
}
```

Обсуждение

Объект `Animation`, возвращаемый из вызова `element.animate`, обладает свойством `playState`, которое можно использовать для определения того, запущена анимация в данный момент или нет. Если анимация запущена, ее значением будет строка `running`. Есть и другие значения.

`paused`

Анимация была запущена, но была приостановлена до ее завершения.

`finished`

Анимация завершилась и остановилась.

В зависимости от свойства `playState` функция переключения анимации вызывает либо паузу, либо воспроизведение, чтобы установить желаемое состояние анимации.

8.3. Анимация вставки и удаления элементов DOM

Задача

Вы хотите добавить в DOM или удалить из него элементы с анимационным эффектом.

Решение

Решения немного различаются для операций.

Для того чтобы *добавить* элемент, сначала добавьте его в DOM, а затем сразу запустите анимацию (например, постепенное появление). Поскольку анимировать можно только элемент DOM, вам необходимо добавить его перед запуском анимации (пример 8.6).

Пример 8.6. Отображение элемента с анимацией

```
/**
 * Показывает элемент, который только что был добавлен в DOM, с анимацией появления.
 * @param element - элемент для анимации
 */
function showElement(element) {
  document.body.appendChild(element);
  element.animate([
    { opacity: 0 },
    { opacity: 1 }
  ], {
    // Анимировать в течение 250 миллисекунд.
    duration: 250
  });
}
```

Для того чтобы *удалить* элемент, сначала нужно запустить анимацию и дождаться ее окончания (например, исчезновение). Как только анимация завершится, немедленно удалите элемент из DOM (пример 8.7).

Пример 8.7. Удаление элемента с анимацией

```
/**
 * Удаляет элемент из DOM после выполнения анимации исчезновения.
 * @param element - элемент для анимации
 */
```

```
async function removeElement(element) {
  // Сначала выполните анимацию и заставьте элемент исчезнуть из поля зрения.
  // Свойство finished для анимации является промисом.
  await element.animate([
    { opacity: 1 },
    { opacity: 0 }
  ], {
    // Анимировать в течение 250 миллисекунд.
    duration: 250
  }).finished;

  // Анимация завершена, теперь удалите элемент из DOM.
  element.remove();
}
```

Обсуждение

Когда вы запускаете анимацию одновременно с добавлением элемента, он начинает анимироваться с нулевой непрозрачностью до того, как у него появится шанс на первоначальный рендеринг. Это создает желаемый эффект — скрытый элемент, который проявляется в поле зрения.

Когда вы удаляете элемент, можете использовать промис в функции `finished` анимации, чтобы дождаться завершения анимации. Не следует удалять элемент из DOM до тех пор, пока анимация не будет полностью завершена, иначе эффект может сработать только частично, и элемент исчезнет раньше времени.

8.4. Реверсирование анимации

Задача

Вы хотите отменить текущую анимацию, например эффект наведения курсора, и плавно вернуть ее в исходное состояние.

Решение

Используйте метод `reverse` анимационного объекта, чтобы начать воспроизведение в обратном направлении.

Можете отслеживать текущую анимацию с помощью переменной. Если вы меняете состояние анимации и эта переменная имеет значение, то уже выполняется другая анимация, и браузер должен запустить ее в обратную сторону.

В примере 8.8 можно запустить анимацию в момент наведения курсора мыши на элемент.

Пример 8.8. Эффект наведения курсора

```
element.addEventListener('mouseover', async () => {
  if (animation) {
    // Анимация уже выполняется. Вместо того чтобы запускать новую анимацию,
    // измените текущую, настроив ее проигрывание в обратном порядке.
    animation.reverse();
  } else {
    // Теперь ничего не происходит, поэтому запустите новую анимацию.
    animation = element.animate([
      { transform: 'scale(1)' },
      { transform: 'scale(2)' }
    ], {
      // Анимировать в течение 1 секунды.
      duration: 1000,
      // Применить начальный и конечный стиль.
      fill: 'both'
    });

    // Как только анимация завершится, установить для текущей анимации
    // значение null.
    await animation.finished;
    animation = null;
  }
});
```

Когда курсор мыши уходит в сторону, применяется та же логика (пример 8.9).

Пример 8.9. Удаление эффекта наведения курсора

```
button.addEventListener('mouseout', async () => {
  if (animation) {
    // Анимация уже выполняется. Вместо того чтобы запускать новую анимацию,
    // измените текущую, настроив ее проигрывание в обратном порядке.
    animation.reverse();
  } else {
    // Теперь ничего не происходит, поэтому запустите новую анимацию.
    animation = button.animate([
      { transform: 'scale(2)' },
      { transform: 'scale(1)' }
    ], {
```

```
      // Анимировать в течение 1 секунды.
      duration: 1000,
      // Применить начальный и конечный стиль.
      fill: 'both'
    });

    // Как только анимация завершится, установить для текущей анимации
    // значение null.
    await animation.finished;
    animation = null;
  }
});
```

Обсуждение

Поскольку ключевые кадры в обоих случаях почти одинаковы (они различаются только порядком расположения), у вас может быть одна функция анимации, которая задает направление анимации при помощи свойства `direction`. При наведении курсора мыши на элемент требуется запустить проигрывание в прямом, или обычном, направлении. Когда курсор мыши уходит, запускается та же анимация, но в обратном направлении (пример 8.10).

Пример 8.10. Одна функция анимации

```
async function animate(element, direction) {
  if (animation) {
    animation.reverse();
  } else {
    animation = element.animate([
      { transform: 'scale(1)' },
      { transform: 'scale(2)' }
    ], {
      // Анимировать в течение 1 секунды.
      duration: 1000,
      // Примените конечный стиль по завершении анимации.
      fill: 'forward',
      // Запустить анимацию вперед (normal) или назад (reverse)
      // в зависимости от параметра direction.
      direction
    });
```

```
    // Как только анимация завершится, установить для переменной значение null,
    // чтобы сигнализировать о том, что анимация больше не выполняется.
    await animation.finished;
    animation = null;
  }
}

element.addEventListener('mouseover', () => {
  animate(element, 'normal');
});

element.addEventListener('mouseout', () => {
  animate(element, 'reverse');
});
```

Результат тот же, что и раньше. Когда вы наводите курсор на элемент, он начинает увеличиваться из-за изменения масштаба `scale(2)`. Если вы отводите мышь, он снова начинает уменьшаться — направление анимации изменяется.

Разница заключается в обработчиках событий. Они оба вызывают одну и ту же функцию, используя разные значения параметра `direction`.

В примере 8.8 задается режим заполнения `fill` анимации в обоих режимах. Режим определяет стиль элемента до и после анимации. По умолчанию используется режим заполнения со значением `none`. Оно определяет, что по завершении анимации стиль элемента возвращается к тому, каким он был до анимации.

На практике это означает, что при наведении курсора мыши на элемент он начинает увеличиваться до тех пор, пока не достигнет своего конечного размера, но затем немедленно возвращается к исходному размеру из-за того, что режим заполнения не установлен.

Существует три варианта (кроме `none`) для режима заполнения `fill`.

`backward`

Перед началом анимации стиль элемента задается начальным ключевым кадром анимации. Обычно это применимо только при использовании задержки анимации, поскольку определяет стиль элемента в пределах периода задержки.

`forward`

После завершения анимации элемент имеет стиль конечного ключевого кадра.

`both`

Применяются правила как обратной, так и прямой анимации.

Анимация в примере 8.10 не имеет задержки, поэтому для сохранения стиля после завершения анимации используется опция `forward`.

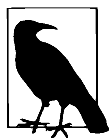
8.5. Отображение индикатора прокрутки

Задача

Вы хотите отобразить полосу в верхней части страницы, которая движется в соответствии с состоянием прокрутки. При прокрутке вниз полоса перемещается вправо.

Решение

Используйте анимацию, связанную с прокруткой, создав `ScrollTimeline` и передав ее методу `animate` элемента. Для того чтобы увеличить размер элемента слева направо, вы можете изменить свойство `transition` со значения `scaleX(0)` на `scaleX(1)`.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/l-hvN**).

Сначала задайте несколько стилей для элемента индикатора выполнения, как показано в примере 8.11.

Пример 8.11. Стили индикатора прокрутки

```
.scroll-progress {
  height: 8px;
  transform-origin: left;
  position: sticky;
  top: 0;
  transform: scaleX(0);
  background: blue;
}
```

Свойство `position: sticky` гарантирует, что элемент остается видимым при прокрутке страницы вниз. Кроме того, его начальный стиль имеет значение `scaleX(0)`, а это эффективно скрывает элемент. Без него индикатор на мгновение появился бы во всю ширину, а затем исчез. А такая настройка гарантирует, что вы вообще не увидите полосу, пока не начнете выполнять прокрутку.

Далее создайте объект `ScrollTimeline` и передайте его в качестве параметра `timeline` анимации, как показано в примере 8.12.

Пример 8.12. Создание временнóй шкалы прокрутки

```
const progress = document.querySelector('.scroll-progress');

// Создайте временную шкалу, привязанную к положению прокрутки документа.
const timeline = new ScrollTimeline({
  source: document.documentElement
});

// Запустите анимацию, проходя по временной шкале, которую вы только что создали.
progress.animate(
  [
    { transform: 'scaleX(0)' },
    { transform: 'scaleX(1)' }
  ],
  { timeline });
```

Теперь у вас есть анимация, связанная с прокруткой страницы.

Обсуждение

Временнáя шкала анимации — это объект, реализующий интерфейс `AnimationTimeline`. По умолчанию анимация использует временную шкалу документа, которая представляет собой объект `DocumentTimeline`. Это временная шкала, привязанная к прошедшему времени (elapsed time). Когда вы запускаете анимацию с временной шкалой по умолчанию, она стартует с начального ключевого кадра и продолжается до тех пор, пока не достигнет конца (или вы не остановите ее вручную). Поскольку этот тип временной шкалы привязан к прошедшему времени, она имеет определенное начальное значение, и оно постоянно увеличивается с течением времени.

Однако анимация, связанная с *прокруткой*, предоставляет временную шкалу, привязанную к положению прокрутки. При прокрутке до самого верха положение прокрутки равно 0, а анимация возвращается в исходное состояние. При прокрутке вниз положение увеличивается, а анимация продвигается вперед. Как только вы прокрутите страницу до конца, анимация завершится. Если вы прокрутите ее снова вверх, анимация запустится в обратном направлении.

В качестве исходного элемента задается `ScrollTimeline`. В примере 8.12 источником является элемент документа (тег `<body>`). В качестве источника можно использовать любой прокручиваемый элемент, а `ScrollTimeline` использует положение прокрутки этого элемента для определения текущего прогресса индикатора прокрутки.

На момент написания книги DocumentTimeline поддерживался во всех современных браузерах, а ScrollTimeline — нет. Обязательно проверяйте поддержку браузера перед использованием ScrollTimeline.

8.6. Создание подпрыгивающего элемента

Задача

Вы хотите применить к элементу эффект мгновенного подпрыгивания.

Решение

Примените серию анимаций, одну за другой. Используйте промис finished анимации, чтобы дождаться ее завершения, прежде чем запускать следующую.

Элемент перемещается вверх и вниз три раза. При каждом проходе элемент перемещается вверх по странице с помощью преобразования translateY, а затем возвращается в исходное положение. При первом проходе элемент отскакивает на 40 пикселов, при втором — на 20 пикселов, а при третьем — на 10. Так создается видимость, будто гравитация замедляет отскок при каждом следующем проходе. Реализовать идею можно с помощью цикла for-of (пример 8.13).

Пример 8.13. Последовательное применение анимации отскока

```
async function animateBounce(element) {
  const distances = [ '40px', '20px', '10px' ];
  for (let distance of distances) {
    // Дождаться завершения этой анимации, прежде чем продолжить.
    await element.animate([
      // Начните с самого низа.
      { transform: 'translateY(0)' },

      // Переместиться вверх на заданное расстояние.
      { transform: `translateY(-${distance})`, offset: 0.5 },

      // Обратно вниз.
      { transform: 'translateY(0)' }
    ], {
      // Анимировать в течение 250 миллисекунд.
      duration: 250,

      // Использовать не линейную (по умолчанию), а более плавную функцию
      // сглаживания.
```

```
      easing: 'ease-in-out'
    }).finished;
  }
}
```

Обсуждение

Этот пример демонстрирует преимущество API веб-анимации: динамические значения ключевых кадров. На каждой итерации цикла используется разное значение `distance` внутри эффекта ключевого кадра.

Цикл `for-of` последовательно проходит через три варианта расстояния (40, 20 и 10 пикселов) и анимирует их по очереди. На каждой итерации элемент перемещается вверх на заданное расстояние и обратно вниз. Ключом является последняя строка, в которой указана ссылка на свойство анимации `finished`. Это гарантирует, что следующая итерация цикла не начнется до завершения текущей анимации. В результате анимации будут выполняться последовательно, одна за другой, обеспечивая эффект отскока.

Возможно, вам интересно, почему в этом примере при обращении к массиву используется цикл `for-of`, а не `forEach()`. Ключевое слово `await` внутри методов массива, таких как `forEach`, работает не так, как вы ожидаете. Эти методы не предназначены для асинхронного использования. Если бы вы применили вызов `forEach`, все вызовы `element.animate` были бы сделаны один за другим, в результате чего воспроизводилась бы только последняя анимация. Использование цикла `for-of` (также подойдет цикл `for`) работает, как и ожидалось, с `async`/`await` и дает желаемый результат.

8.7. Одновременный запуск нескольких анимаций

Задача

Вы хотите применить преобразования к элементу, используя несколько анимаций.

Решение

Вызовите `animate` для элемента несколько раз, используя разные ключевые кадры преобразования. Вы также должны указать свойство `composite` для объединения преобразований, как показано в примере 8.14.

Пример 8.14. Объединение двух преобразований анимации

```
// Первая анимация будет перемещать элемент вперед и назад по оси x.
element.animate([
  { transform: 'translateX(0)' },
```

```
  { transform: 'translateX(250px)' }
], {
  // Анимировать в течение 5 секунд.
  duration: 5000,
  // Запустить анимацию вперед, затем в обратном порядке.
  direction: 'alternate',
  // Повторять анимацию бесконечно.
  iterations: Infinity,
  // Медленно в начале, быстро в середине, медленно в конце.
  easing: 'ease-in-out'
});

// Вторая анимация вращает элемент.
element.animate([
  { transform: 'rotate(0deg)' },
  { transform: 'rotate(360deg)' }
], {
  // Анимировать в течение 3 секунд.
  duration: 3000,
  // Повторять анимацию бесконечно.
  iterations: Infinity,
  // Комбинировать эффекты с другими запущенными анимациями.
  composite: 'add'
});
```

Направление `alternate` означает, что анимация выполняется сначала до завершения, а затем в обратном направлении. Поскольку для параметра `iterations` задано значение бесконечности (`Infinity`), анимация выполняется бесконечно.

Обсуждение

Ключом к этому эффекту является свойство `composite`, добавленное ко второй анимации. Если вы не укажете `composite: add`, вы увидите *только* преобразование `rotate`, потому что оно накладывается на преобразование `translateX`. Элемент будет вращаться, но перемещаться по горизонтали не станет.

Это, по сути, объединяет оба преобразования в одно. Также обратите внимание, что преобразования выполняются с разной скоростью. Вращение длится три секунды, а перемещение — пять. В анимациях также используются разные функции сглаживания. Несмотря на различные опции, браузер легко комбинирует анимации.

8.8. Отображение анимации загрузки

Задача

Вы хотите показать пользователю индикатор загрузки во время ожидания завершения сетевого запроса.

Решение

Создайте и оформите индикатор загрузки, затем примените к нему анимацию бесконечного вращения до тех пор, пока не будет разрешен `Promise`, возвращаемый функцией запроса `fetch`.

Для того чтобы добиться плавного эффекта, можете сначала применить анимацию с плавным переходом. Как только промис разрешится, вы можете ее завершить при помощи анимации исчезновения.

Сначала создайте элемент загрузки и определите некоторые стили, как показано в примере 8.15.

Пример 8.15. Элемент загрузки

```
<style>
  #loader {
    width: 64px;
    height: 64px;

    /* Нарисуйте круг */
    border-radius: 50%;
    border-width: 10px;
    border-style: solid;
    border-color: skyblue blue skyblue blue;

    /* Установите начальную непрозрачность, чтобы появляющаяся анимация */
    /* была плавной. */
    opacity: 0;
  }
</style>

<div id="loader"></div>
```

Загрузчик представляет собой кольцо с чередующимися цветами (рис. 8.1).

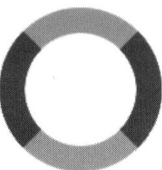


Рис. 8.1. Стилизованный загрузчик

Далее определите функцию, которая запускает анимацию, и дождитесь `Promise`, как показано в примере 8.16.

Пример 8.16. Анимация загрузки

```
async function showLoader(promise) {
  const loader = document.querySelector('#loader');

  // Запустите анимацию вращения перед появлением загрузчика.
  const spin = loader.animate([
    { transform: 'rotate(0deg)' },
    { transform: 'rotate(360deg)' }
  ], { duration: 1000, iterations: Infinity });

  // Поскольку непрозрачность равна 0, загрузчик пока не виден.
  // Отобразите его с помощью анимации появления.
  // Загрузчик будет продолжать вращаться по мере появления.
  loader.animate([
    { opacity: 0 },
    { opacity: 1 }
  ], { duration: 500, fill: 'both' });

  // Дождитесь разрешения промиса.
  await promise;

  // Промис выполнен. Теперь анимируйте исчезновение загрузчика.
  // Не останавливайте анимацию вращения, пока не завершится исчезновение.
  // Можете подождать до промиса 'finished'.
  await loader.animate([
    { opacity: 1 },
    { opacity: 0 }
  ], { duration: 500, fill: 'both' }).finished;

  // Наконец, остановите анимацию вращения.
  spin.cancel();
```

```
  // Верните первоначальный промис, чтобы разрешить объединение в цепочку.
  return promise;
}
```

Теперь можете передать свой вызов fetch в качестве аргумента showLoader, как показано в примере 8.17.

Пример 8.17. Использование загрузчика

```
showLoader(
  fetch('https://example.com/api/users')
    .then(response => response.json())
);
```

Обсуждение

Вам не обязательно использовать API веб-анимации для создания анимированного загрузчика — можете сделать это с помощью стилей CSS. Однако, как показано в примере, API веб-анимации позволяет комбинировать несколько анимаций. Анимация бесконечного вращения продолжается, пока выполняется анимация постепенного появления. Сделать это с помощью обычной CSS-анимации немного сложнее.

8.9. Соблюдение в анимации предпочтений пользователя

Задача

Вы хотите приглушить или отключить анимацию, если пользователь настроил свою операционную систему на замедление движения на экране.

Решение

Используйте window.matchMedia, чтобы проверить запрос prefers-reduced-motion (пример 8.18).

Пример 8.18. Использование запроса prefers-reduced-motion

```
if (!window.matchMedia('(prefers-reduced-motion: reduce)').matches) {
  // Функция замедления движения не включена, поэтому анимируйте в обычном режиме.
} else {
  // Пропустите эту анимацию или запустите менее интенсивную.
}
```

Обсуждение

Реализация идеи замедления анимации чрезвычайно важна для обеспечения доступности. У пользователей, страдающих эпилепсией или вестибулярными расстройствами, могут возникнуть судороги, мигрень или другие симптомы, вызванные большой или быстро движущейся анимацией.

Необязательно полностью отключать анимацию, вместо этого можно использовать более тонкую настройку. Предположим, вы показываете элемент с эффектом отскока, который выглядит действительно хорошо, но может дезориентировать некоторых пользователей. Если у пользователя включена функция замедления движения, вы могли бы вместо этого предоставить простую анимацию с постепенным появлением элемента.

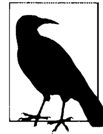
ГЛАВА 9
Web Speach API

9.0. Введение

В век интеллектуальных устройств и помощников голос стал еще одним широко используемым методом ввода. Независимо от того, диктуете ли вы текстовое сообщение или запрашиваете прогноз погоды на завтра, распознавание и синтез речи становятся полезными инструментами при разработке приложений. С помощью Web Speech API вы можете заставить свое приложение говорить или прослушивать голосовой ввод пользователя.

Распознавание речи

Web Speech API обеспечивает *распознавание* речи в браузере. Как только пользователь дает вам разрешение на использование микрофона, микрофон начинает прослушивать звуки. Когда распознается серия слов, API запускает событие с распознанным содержимым.



Возможно, распознавание речи пока поддерживается не всеми браузерами. Вы можете воспользоваться последними данными о совместимости, посетив ресурс CanIUse (**https://oreil.ly/SGLlc**).

Прежде чем начать прослушивание речи, вам потребуется разрешение пользователя. Из-за настроек конфиденциальности при первой попытке прослушивания пользователю будет предложено предоставить приложению разрешение на использование микрофона (рис. 9.1).

Некоторые браузеры, такие как Chrome, используют внешний сервер для анализа записанного аудио с целью распознавания речи. Это означает, что распознавание речи не будет работать в оффлайн-режиме, а также может вызвать проблемы с конфиденциальностью.

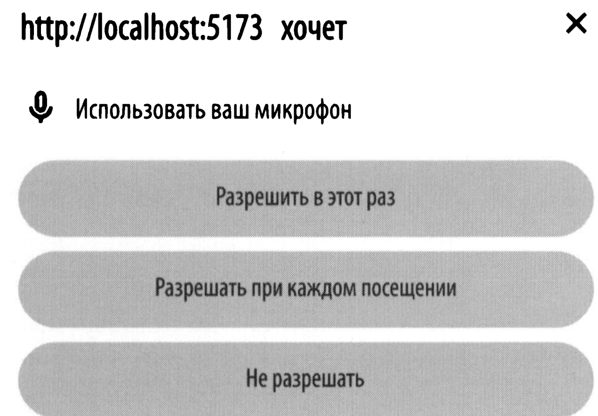


Рис. 9.1. Запрос разрешения доступа к микрофону в браузере Chrome

Распознавание речи в сравнении с обработкой речи

Важно различать распознавание речи (определение того, какие слова были произнесены) и обработку речи (понимание значения этих слов). Web Speech API сам по себе не придает никакого значения распознанным словам; он возвращает их вам в виде строки, и вы сами должны выполнить ту дополнительную обработку, какую пожелаете. Можете интегрировать эти данные с помощью сторонних API-интерфейсов для обработки данных на естественном языке (natural language processing, NLP), таких как Microsoft LUIS или IBM Watson NLP. Рассмотрение этих API-интерфейсов и сервисов выходит за рамки данной книги.

Синтез речи

Web Speech API также обеспечивает *синтез* речи. При наличии некоторого текста он может синтезировать голос, который произносит текст. Браузер имеет набор встроенных тембров, которые он может использовать для озвучивания вашего контента. После того как вы выбрали голос, соответствующий выбранному языку, вы можете настроить высоту голоса и скорость произнесения.

Вы можете комбинировать функции распознавания и синтеза речи для создания пользовательских интерфейсов, позволяющих вести диалог. Они могут прослушивать вопрос или команду и проговаривать ответ в качестве обратной связи.

Поддержка браузеров

На момент написания этой книги поддержка Web Speech API была несколько ограничена.

В спецификацию этого API также добавлены несколько других компонентов, которые улучшат распознавание и синтез речи, как только те будут поддерживаться в браузерах.

Первый из них — пользовательская грамматика, которая позволяет вам точно настроить распознавание речи, указав слова и фразы, которые вы хотите распознать. Например, если бы вы разрабатывали калькулятор с голосовыми вводом, ваша пользовательская грамматика включала бы цифры ("один", "два" и т. д.) и операции калькулятора ("плюс", "минус" и т. д.). Использование пользовательской грамматики помогает механизму распознавания речи захватывать слова, которые ищет ваше приложение.

SpeechSynthesis API поддерживает язык разметки синтеза речи (Speech Synthesis Markup Language, SSML). SSML — это язык XML, который настраивает синтез речи. Вы можете переключаться между мужским и женским голосами или указать, что браузер должен читать что-либо по буквам. На момент написания книги разметка SSML может быть проанализирована и понята браузером (движок не произносит теги разметки), но браузеры в настоящее время игнорируют большинство инструкций.

9.1. Добавление продиктованного текста в текстовое поле

Задача

Вы хотите распознать произносимый текст и добавить его к содержимому текстового поля, т. е. позволить пользователю вводить содержимое в текстовое поле голосом.

Решение

Для прослушивания речи используйте интерфейс распознавания речи. Когда речь будет распознана, извлеките распознанный текст и добавьте его в текстовое поле (пример 9.1).

Пример 9.1. Добавление базового текста под диктовку в текстовое поле

```
/**
 * Начинает прослушивание речи. Когда речь распознана, она добавляется к значению
 * данного текстового поля.
 * Распознавание продолжается до тех пор, пока не будет остановлен
 * возвращаемый объект распознавания.
 *
 * @param textField - текстовое поле
 * @returns распознанный объект
 */
```

```
function startDictation(textField) {
  // Продолжайте, только если этот браузер поддерживает распознавание речи.
  if ('webkitSpeechRecognition' in window || 'SpeechRecognition' in window) {
    const SpeechRecognition = window.SpeechRecognition
    || window.webkitSpeechRecognition;
    const recognition = new SpeechRecognition();
    recognition.continuous = true;

    recognition.addEventListener('result', event => {
      const result = event.results[event.resultIndex];
      textField.value += result[0].transcript;
    });

    recognition.addEventListener('error', event => {
      console.log('error', event);
    });

    recognition.start();

    // Возвращает объект распознавания, чтобы распознавание можно было остановить
    // позже (например, когда пользователь нажимает кнопку переключения).
    return recognition;
  }
}
```

Обсуждение

На момент написания этой книги в браузерах, поддерживающих WebKit, конструктор распознавания речи SpeechRecognition имеет префикс webkitSpeechRecognition. В неподдерживающих браузерах не определены ни функция распознавания речи, ни функция webkitSpeechRecognition, поэтому важно проверить поддержку браузера, прежде чем продолжить разработку.

Для обеспечения будущей надежности кода в примере проверяется версия с префиксом (webkitSpeechRecognition), а также стандартная версия SpeechRecognition. Таким образом, в будущем вам не придется изменять код для адаптации к браузерам, реализующим API.

Затем функция startDictation создает объект распознавания речи и устанавливает для него флаг continuous в значение true. По умолчанию после распознавания результата дальнейшее распознавание не выполняется. Установка флага continuous указывает механизму распознавания речи продолжить прослушивание и выдать дополнительные результаты.

Когда механизм распознавания обрабатывает какую-либо речь, он запускает событие `result`. Это событие имеет свойство `results`, представляющее собой объект, который похож на массив (на самом деле это объект `Speechrecognitionresultlist`), содержащий результаты.

При работе в непрерывном режиме, как в нашем примере, список `results` содержит все результаты, которые распознал механизм распознавания. Когда пользователь начинает говорить в первый раз и часть речи распознается, это приводит к единственному результату. Когда пользователь заговаривает снова и браузер распознает еще несколько слов, появляются два результата — исходный результат и новый, который был только что распознан. Если вы установите значение `continuous` равным `false` (значение по умолчанию), движок распознает только одну фразу, и дальнейшие события `result` не запускаются.

К счастью, у события также есть свойство `resultIndex`, которое указывает на индекс в списке нового результата, вызвавшего это событие.

Результирующий объект представляет собой другой объект, подобный массиву (объект `SpeechRecognitionAlternative`). Когда вы создаете объект `SpeechRecognition`, вы можете присвоить ему свойство `maxAlternatives`. Браузер отображает список возможных совпадений для распознанной речи, каждое из которых имеет свое значение достоверности. Однако значение `maxAlternatives` по умолчанию равно 1, поэтому в данном коде для диктовки в списке всегда присутствует только один объект `SpeechRecognitionAlternative`.

Наконец, этот объект обладает свойством `transcript`, которое является фактической фразой, распознанной движком. Вы можете взять это значение и добавить его к текущему значению текстового поля.

При вызове `start` объект распознавания начинает прослушивать речь, выдавая события, когда он что-то слышит. Затем функция `startDictation` возвращает объект распознавания, так что вы можете остановить распознавание, как только пользователь закончит диктовать.

Как и в случае с любым API, также важно обрабатывать любые возникающие ошибки. При распознавании речи могут возникнуть следующие распространенные ошибки.

Ошибка промиса.

> Если пользователь не предоставил разрешение на использование микрофона. При этом возникает ошибка `not-allowed`.

Сетевая ошибка.

> Если браузеру не удалось подключиться к службе распознавания речи. Это ошибка сети.

Ошибка аппаратного обеспечения.

> Если браузеру не удалось получить доступ к микрофону. При этом отображается код ошибки `audio-capture`.

9.2. Создание Promise-помощника для распознавания речи

Задача

Вы хотите инкапсулировать распознавание речи в один вызов функции.

Решение

Оберните вызов распознавания речи в новый `Promise` внутри вспомогательной функции. В рамках вспомогательной функции создайте новый объект распознавания речи `SpeechRecognition` и прослушайте речь. Вы можете разрешить `Promise`, когда браузер распознает какую-то часть речи (пример 9.2).

Пример 9.2. Promise-помощник для распознавания речи

```
/**
 * Прослушивает речь и выполняет распознавание речи.
 * Предполагается, что распознавание речи доступно в текущем браузере.
 * @returns Promise, который разрешается распознанной речью,
 *          или отклоняется из-за ошибки.
 */
function captureSpeech() {
  const speechPromise = new Promise((resolve, reject) => {
    const SpeechRecognition = window.SpeechRecognition ||
      window.webkitSpeechRecognition;

    // Если этот браузер не поддерживает распознавание речи, отклоните Promise.
    if (!SpeechRecognition) {
      reject('Speech recognition is not supported on this browser.')
    }

    const recognition = new SpeechRecognition();

    // Разрешите промис при успешном распознавании речи.
    recognition.addEventListener('result', event => {
      const result = event.results[event.resultIndex];
      resolve(result[0].transcript);
    });

    recognition.addEventListener('error', event => {
      // Отклоните промис при ошибке распознавания речи.
```

```
      reject(event);
    });

    // Начать прослушивать речь.
    recognition.start();
  });

  // Независимо от того, было ли распознавание речи успешным или произошла ошибка,
  // убедитесь, что механизм распознавания прекратил прослушивание.
  return speechPromise.finally(() => {
    recognition.stop();
  });
}
```

Обсуждение

Помощник captureSpeech не использует непрерывный режим. Это означает, что вы можете применять его только для прослушивания одного события распознавания речи. Если вы хотите записать дополнительную речь после того, как возвращаемый промис будет выполнен, вам нужно снова вызвать captureSpeech и дождаться нового промиса.

Вы можете заметить, что пример 9.2 не возвращает промис напрямую. Вместо этого он вызывает этот промис с помощью функции finally, чтобы остановить механизм распознавания речи независимо от результата. Функция captureSpeech позволяет быстро распознавать речь, просто ожидая выполнения промиса (пример 9.3).

Пример 9.3. Использование помощника captureSpeech

```
const spokenText = await captureSpeech();
```

9.3. Получение доступных голосов

Задача

Вы хотите определить, какие голоса для синтеза речи доступны в текущем браузере.

Решение

Запросите список голосов, вызвав SpeechSynthesis.getVoices, а затем, при необходимости, прослушайте событие voiceschanged, как представлено в примере 9.4.

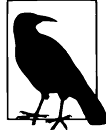
Пример 9.4. Получение списка доступных голосов для синтеза речи

```
function showVoices() {
  speechSynthesis.getVoices().forEach(voice => {
    console.log('Voice:', voice.name);
  });
}

// Некоторые браузеры загружают список голосовых сообщений асинхронно.
// В этих браузерах голоса доступны при запуске события voiceschanged.
speechSynthesis.addEventListener('voiceschanged', () => showVoices());

// Немедленно отображать голоса в тех браузерах, которые это поддерживают.
showVoices();
```

Обсуждение

Некоторые браузеры, такие как Chrome, загружают список голосов асинхронно. Если вы вызовете `getVoices` до того, как список будет готов, вы получите пустой массив. Объект синтеза речи запускает событие `voiceschanged` по готовности списка.

В других браузерах, включая Firefox, список голосов доступен сразу. В этих браузерах событие `voiceschanged` никогда не срабатывает. Код в примере 9.4 обрабатывает оба случая.



У каждого голоса есть свойство `lang`, которое определяет его язык. При произнесении текста голос использует правила произношения для своего языка. Убедитесь, что для синтезируемого текста вы используете голос с правильным языком. В противном случае произношение будет звучать неправильно.

9.4. Синтез речи

Задача

Вы хотите, чтобы ваше приложение озвучило пользователю какой-либо текст.

Решение

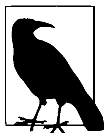
Создайте запрос `SpeechSynthesis` и передайте его методу `SpeechSynthesis.speak` (пример 9.5).

Пример 9.5. Озвучивание некоторого текста с помощью Web Speech API

```
function speakText(text) {
  const utterance = new SpeechSynthesisUtterance(text);
  speechSynthesis.speak(utterance);
}
```

Обсуждение

Высказывание — это набор слов, которые браузер должен произнести. Оно создается с помощью объекта SpeechSynthesisUtterance.



Браузеры разрешат синтез речи только после того, как пользователь каким-либо образом начал взаимодействие со страницей. Это делается для того, чтобы страница не стала озвучиваться сразу после загрузки. Таким образом, функция speakText ничего не будет произносить до тех пор, пока на странице не начнется какая-либо активность пользователя.

При этом текст будет воспроизводиться голосом, используемым по умолчанию. Если вы хотите использовать другой голос, поддерживаемый системой, можете воспользоваться приемом из рецепта 9.3, чтобы получить набор доступных голосов. Вы можете присвоить свойству voice объекта utterance значение одного из голосовых объектов из массива, как показано в примере 9.6.

Пример 9.6. Использование другого голоса

```
// Предполагается, что голоса сейчас доступны.
const aliceVoice = speechSynthesis
  .getVoices()
  .find(voice => voice.name === 'Alice');

function speakText(text) {
  const utterance = new SpeechSynthesisUtterance(text);
  // Убедитесь, что голос Алисы был найден.
  if (aliceVoice) {
    utterance.voice = aliceVoice;
  }

  speechSynthesis.speak(utterance);
}
```

9.5. Настройка параметров синтеза речи

Задача

Вы хотите ускорить, замедлить темп произношения текста или отрегулировать высоту звучания.

Решение

При создании SpeechSynthesisUtterance используйте свойства скорости и высоты звука, чтобы настроить голос говорящего (пример 9.7).

Пример 9.7. Настройка вывода речи

```
const utteranceLow =
new SpeechSynthesisUtterance('This is spoken slowly in a low tone');
utterance.pitch = 0.1;
utterance.rate = 0.5;
speechSynthesis.speak(utterance);

const utteranceHigh =
new SpeechSynthesisUtterance('This is spoken quickly in a high tone');
utterance.pitch = 2;
utterance.rate = 2;
speechSynthesis.speak(utterance);
```

Обсуждение

Параметр высоты звука pitch — это число с плавающей точкой, которое может принимать значения от 0 до 2. Более низкие значения приводят к снижению высоты звука, а более высокие — к его повышению. Снижение высоты звука не влияет на скорость произнесения. В зависимости от используемого браузера или голосовой системы диапазон поддерживаемых значений высоты тона может быть ограничен.

Для того чтобы ускорить или замедлить скорость (частоту) произнесения, вы можете настроить параметр rate. Для каждого голоса по умолчанию используется частота произнесения, равная 1. Значение частоты имеет мультипликативный эффект. Если вы установите значение, равное 0,5, это будет половина от значения по умолчанию. Аналогично, если вы установите скорость 1,5, она будет на 50% выше, чем скорость по умолчанию. В спецификации допустимый диапазон составляет от 0,1 до 10, но браузеры и голосовые системы обычно ограничивают этот диапазон меньшими значениями.

9.6. Автоматическая приостановка речи

Задача

Когда ваше приложение говорит, вы хотите приостановить речь при переключении на другую вкладку, чтобы это не мешало использованию другой вкладки. Вы также хотите прекратить озвучивание, если пользователь покидает страницу.

Решение

Прослушайте событие `visibilitychange` и проверьте свойство `document.visibilityState`. Когда страница станет неактивной, приостановите синтез речи. Когда она снова станет видимой, возобновите воспроизведение (пример 9.8).

Пример 9.8. Приостановка речи, когда страница становится неактивной

```
document.addEventListener('visibilitychange', () => {
  // speechSynthesis.speaking имеет значение true:
  // (1) когда идет озвучивание,
  // (2) когда озвучивание поставлено на паузу.
  if (speechSynthesis.speaking) {
    if (document.visibilityState === 'hidden') {
      speechSynthesis.pause();
    } else if (document.visibilityState === 'visible') {
      speechSynthesis.resume();
    }
  }
});
```

Обсуждение

По умолчанию, если вы переключаетесь на другую вкладку во время того, как Web Speech API произносит какой-либо текст, он продолжает говорить. Возможно, это то, чего вы ожидаете — в конце концов, то же самое происходит, если вы транслируете аудио или видео в потоковом режиме, а затем переключаетесь на другую вкладку; вы продолжите прослушивать аудио, находясь на другой вкладке.

При переключении вкладок срабатывает событие `visibilitychange`. Само событие не содержит никакой информации о состоянии видимости, но вы можете получить ее, проверив свойство `document.visibilityState`. В примере 9.8 речь приостановится, когда вы переключитесь на другую вкладку. Когда вы вернетесь обратно, аудиовоспроизведение продолжится с места остановки.

Некоторые браузеры продолжают воспроизводить речь, даже когда вы уходите со страницы или полностью ее обновляете. Выход со страницы или обновление страницы также запускает событие `visibilitychange`, поэтому код в примере 9.8 корректно останавливает речь и в этих случаях.

ГЛАВА 10
Работа с файлами

10.0. Введение

Чтение и запись файлов являются частью возможностей многих приложений. Были времена, когда работать с локальными файлами в браузере было невозможно. Для чтения данных приходилось загружать файл на сервер, который обрабатывал его и возвращал данные в браузер.

Для записи данных сервер отправлял загружаемый файл. Без подключаемых модулей для браузера отсутствовала возможность напрямую работать с файлами.

Сегодня браузеры обладают первоклассной поддержкой чтения и записи файлов. При выборе типа файла открывается стандартное диалоговое окно выбора файла и предоставляются данные об этом файле. Вы также можете ограничить поддерживаемые файлы с определенными расширениями или типами MIME. После этого File API (API файловой системы) может считывать содержимое файла в память.

Делая еще один шаг вперед, API файловой системы позволяет вашему JavaScript-коду напрямую взаимодействовать с локальной файловой системой, не требуя предварительного выбора файла. (Хотя, в зависимости от настроек, пользователю может потребоваться предоставить разрешение!)

Вы можете использовать эти API для создания текстовых редакторов, средств просмотра изображений, аудио- или видеоплееров и многого другого.

10.1. Загрузка текста из файла

Задача

Вы хотите загрузить некоторые текстовые данные из локальной файловой системы пользователя.

Решение

Используйте тег `<input type="file">` для выбора файла (пример 10.1).

Пример 10.1. Выбор файла

```
<input type="file" id="select-file">
```

Когда вы нажмете на кнопку выбора файла, браузер отобразит диалоговое окно, в котором вы сможете просматривать файлы и папки в локальной системе. Как именно будет выглядеть диалоговое окно, зависит от браузера и версии операционной системы. Перейдите к нужному файлу и выберите его. Если у вас есть выбранный файл, используйте объект `FileReader` для чтения текстового содержимого файла, как показано в примере 10.2.

Пример 10.2. Загрузка обычного текста из файла

```
/**
 * Читает текстовое содержимое из файла.
 * @param file - объект файла, содержащий данные для чтения
 * @param onSuccess - функция, вызываемая, когда данные становятся доступными
 */
function readFileContent(file, onSuccess) {
  const reader = new FileReader();

  // После загрузки контента метод чтения создает событие event.
  reader.addEventListener('load', event => {
    onSuccess(event.target.result);
  });

  // Всегда обрабатывайте ошибки!
  reader.addEventListener('error', event => {
    console.error('Ошибка чтения файла:', event);
  });

  // Начните чтение из файла.
  reader.readAsText(file);
}

const fileInput = document.querySelector('#select-file');

// Событие change создается, когда выбран файл.
fileInput.addEventListener('change', event => {
  // Это массив, поскольку можно выбрать несколько файлов. Здесь выбран только один
  // файл. Здесь используется синтаксис деструктурирования массива для получения
  // первого файла.
```

```
  const [file] = fileInput.files;
  readFileContent(file, content => {
    // Теперь доступно текстовое содержимое файла. Представьте, что
    // у вас есть элемент textarea, в который вы хотите поместить текст.
    const textArea = document.querySelector('.file-content-textarea');
    textArea.textContent = content;
  });
});
```

Обсуждение

`FileReader` — это объект, который считывает файлы асинхронно. Он может считывать содержимое файла несколькими различными способами в зависимости от типа файла. В примере 10.2 используется метод `readAsText`, который извлекает содержимое файла в виде обычного текста.

Если у вас есть двоичный файл, такой как ZIP-архив или изображение, вы можете использовать `readAsBinaryString`. Изображение может быть прочитано как URL-адрес данных с данными изображения, закодированными в Base64, с использованием `readAsDataURL` (см. рецепт 10.2).

Этот API основан на событиях, поэтому функция `readFileContent` принимает функцию обратного вызова, которая вызывается вместе с содержимым, когда оно готово.

Вы также могли бы обернуть это с помощью API, основанном на Promise, как показано в примере 10.3.

Пример 10.3. Функция readFileContent, обернутая в Promise

```
function readFileContent(file) {
  const reader = new FileReader();
  return new Promise((resolve, reject) => {
    reader.addEventListener('load', event => {
      resolve(event.target.result);
    });
    reader.addEventListener('error', reject);
    reader.readAsText(file);
  });
}

try {
  const content = await readFileContent(inputFile);
  const textArea = document.querySelector('.file-content-textarea');
```

```
  textArea.textContent = content;
} catch (error) {
  console.error('Ошибка чтения содержимого файла:', error);
}
```

Получив текстовое содержимое, вы можете добавить его на страницу несколькими способами. Вы можете задать его как `textContent` узла DOM или даже загрузить в текстовую область, чтобы сделать содержимое доступным для редактирования.

10.2. Загрузка изображения из URL-адреса данных

Задача

Вы хотите, чтобы пользователь мог выбрать локальный файл изображения, а затем отобразить это изображение на странице.

Решение

Используйте метод `readAsDataURL` из `FileReader`, чтобы получить URL-адрес данных в кодировке Base64, затем задайте его в качестве атрибута `src` тега `img` (примеры 10.4 и 10.5).

Пример 10.4. Ввод файла и заполнитель изображения

```
<input
  type="file"
  id="select-file"
  accept="image/*" ❶
>
<img id="placeholder-image">
```

❶ Ограничивает выбор файла, позволяя выбирать только изображения. Здесь используется шаблон подстановки (wildcard), но вы также можете указать точный MIME-тип, например `image/png`.

Пример 10.5. Загрузка изображения на страницу

```
/**
 * Загружает изображение из файла и показывает его на странице.
 * @param file - объект файла, содержащего изображение
 * @param imageElement - заполнитель, который будет заменен изображением
 */
```

```
function showImageFile(file, imageElement) {
  const reader = new FileReader();

  reader.addEventListener('load', event => {
    // Задайте URL-адрес данных непосредственно в качестве атрибута
    // изображения src, чтобы загрузить изображение.
    imageElement.src = event.target.result;
  });

  reader.addEventListener('error', event => {
    console.log('error', event);
  });

  reader.readAsDataURL(file);
}

const fileInput = document.querySelector('#select-file');
  fileInput.addEventListener('change', event => {
    showImageFile(
      fileInput.files[0],
      document.querySelector('#placeholder-image')
    );
});
```

Обсуждение

URL-адрес данных имеет схему `data` URL-адресов. Она определяет MIME-тип данных, после чего данные изображения включаются в формат, закодированный в Base64:

```
data:image/png;base64,UHJldGVuZCB0aGlzIGlzIGltYWdlIGRhdGE=
```

Когда объект `FileReader` возвращает изображение, закодированное в виде URL-адреса данных, URL-адрес данных устанавливается в качестве атрибута `src` тега `<image>`. При этом изображение отображается на странице.

Важно отметить, что все это выполняется локально в браузере пользователя. На удаленный сервер ничего не загружается, т. к. файловый API работает в локальной файловой системе.

В рецепте 4.4 *главы 4* приведен пример использования тега `<input type="file">` для загрузки файла данных на удаленный сервер, но здесь используется FormData API вместо File API.

Для получения более подробной информации об URL-адресах данных и кодировке Base64 смотрите статью из MDN (**https://oreil.ly/kMtDy**).

10.3. Загрузка видео в качестве URL-адреса объекта

Задача

Вы хотите, чтобы пользователь выбрал видеофайл, а затем воспроизвел его в браузере.

Решение

Создайте URL-адрес объекта для файлового объекта и установите его в качестве атрибута src для тега <video>.

Сначала вам понадобятся теги <video> и <input type="file">, чтобы выбрать видеофайл (пример 10.6).

Пример 10.6. HTML-разметка видеопроигрывателя

```
<input
  id="file-upload"
  type="file"
  accept="video/*" ❶
>

<video
  id="video-player"
  controls ❷
>
```

❶ Позволяет выбирать только видеофайлы.

❷ Указывает браузеру включить элементы управления воспроизведением.

Затем прослушайте событие change изменения входного файла и создайте объект из URL-адреса, как показано в примере 10.7.

Пример 10.7. Воспроизведение видеофайла

```
const fileInput = document.querySelector('#file-upload');
const video = document.querySelector('#video-player');

fileInput.addEventListener('change', event => {
  const [file] = fileInput.files;
```

```
  // Файл является расширением большого двоичного объекта Blob,
  // который может быть передан в createObjectURL.
  const objectUrl = URL.createObjectURL(file);

  // Элемент <video> может принять URL-адрес объекта для загрузки видео.
  video.src = objectUrl;
});
```

Обсуждение

URL-адрес объекта — это специальный URL-адрес, который ссылается на содержимое файла. Вы можете создать его без объекта `FileReader`, поскольку в самом файле есть метод `createObjectURL`. Этот URL-адрес можно передать тегу `<video>`.

> **URL-адреса данных и URL-адреса объектов**
>
> Существуют некоторые важные различия между URL-адресами данных и URL-адресами объектов.
>
> URL-адрес данных содержит данные *в* URL-адресе. Данные (обычно двоичные) кодируются в формате Base64 и добавляются к самому URL-адресу.
>
> URL-адрес объекта представляет собой некоторые данные, которые были загружены в память браузера, обычно это большие двоичные объекты и файлы. Сам URL-адрес не содержит данных, но является ссылкой на фактические данные. Когда вы закончите его использовать, URL-адрес объекта можно отменить, чтобы предотвратить утечки памяти.

10.4. Загрузка изображения с помощью перетаскивания

Задача

Вы хотите иметь возможность перетаскивать файл изображения в окно браузера и показывать это изображение на странице после перетаскивания.

Решение

Определите элемент, который будет использоваться в качестве целевой области перетаскивания, и элемент изображения-заполнителя (пример 10.8).

Пример 10.8. Цель перетаскивания и элемент изображения

```
<label id="drop-target">
  <div>Перетащите изображение сюда</div>
```

```
  <input type="file" id="file-input">
</label>
<img id="placeholder">
```

Обратите внимание, что в этом примере все еще используется ввод файла input. Это сделано для того, чтобы те, кто использует вспомогательные технологии, также могли загружать изображение, не прибегая к операции перетаскивания. Поскольку объект перетаскивания представляет собой метку, содержащую входные данные файла, вы можете щелкнуть в любом месте внутри объекта перетаскивания, чтобы открыть средство выбора файла.

Сначала создайте функцию, которая получает файл изображения и считывает его как URL-адрес данных (пример 10.9).

Пример 10.9. Чтение перетаскиваемого файла

```
function showDroppedFile(file) {
  // Читает данные из файла и вставляет загруженное изображение
  // в страницу.
  const reader = new FileReader();
  reader.addEventListener('load', event => {
    const image = document.querySelector('#placeholder');
    image.src = event.target.result;
  });

  reader.readAsDataURL(file);
}
```

Затем создайте функции-обработчики для событий dragover и drop. Эти события привязываются к целевому элементу перетаскивания (пример 10.10).

Пример 10.10. Добавление кода для перетаскивания

```
const target = document.querySelector('#drop-target');
target.addEventListener('drop', event => {
  // Отмените событие drop. Иначе браузер покинет эту страницу
  // и перейдет непосредственно к файлу.
  event.preventDefault();

  // Получите данные выбранного файла. dataTransfer.items - это DataTransferItemList.
  // Каждый элемент DataTransferItem в списке содержит данные о перетаскиваемом
  // элементе. Поскольку в этом примере рассматривается только один файл,
  // это первый элемент в списке.
  const [item] = event.dataTransfer.items;
```

```
  // Получите данные, которые вы перетащили, в виде файлового объекта.
  const file = item.getAsFile();

  // Продолжайте только в том случае, если вы файл изображения все-таки перетащили.
  if (file.type.startsWith('image/')) {
    showDroppedFile(file);
  }
});

// Вам также необходимо отменить событие перетаскивания, чтобы
// предотвратить замену файлом всего содержимого страницы.
target.addEventListener('dragover', event => {
  event.preventDefault();
});
```

Наконец, не забудьте подключить резервный ввод файла. Вам просто нужно получить выбранный файл, а затем передать его методу `showDroppedFile`, чтобы получить тот же результат (пример 10.11).

Пример 10.11. Обработка входных данных файла

```
const fileInput = document.querySelector('#file-input');
fileInput.addEventListener('change', () => {
  const [file] = fileInput.files;
  showDroppedFile(file);
});
```

Обсуждение

По умолчанию, когда вы перетаскиваете изображение на страницу, браузер покидает текущую страницу. URL-адрес меняется на путь к файлу, и изображение появляется в окне браузера. В этом примере вместо этого вы хотите загрузить данные изображения в элемент `<img>` и остаться на текущей странице.

Для того чтобы предотвратить поведение по умолчанию, обработчик перетаскивания вызывает `preventDefault` при событии `drop`. Для того чтобы полностью предотвратить такое поведение, вам также необходимо вызвать `preventDefault` при событии `dragover`, поэтому вам нужен второй прослушиватель событий. Таким образом элемент действительно может получать события `drop`.

10.5. Проверка и запрос разрешений

Задача

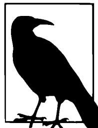
Вы хотите проверить, а при необходимости — запросить, разрешения на доступ к файлу в локальной файловой системе.

Решение

Отобразите средство выбора файлов и, когда файл будет выбран, вызовите запрос разрешения, чтобы проверить наличие существующего разрешения. Если проверка разрешений возвращает `prompt`, вызовите `requestPermission`, чтобы отобразить запрос на разрешение (пример 10.12).

Пример 10.12. Выбор и проверка разрешений для файла

```
/**
 * Выбирает файл, проверяет разрешения, в случае необходимости показывает запрос
 * файла.
 * @returns true, если в файл разрешена запись, иначе false
 */
async function canAccessFile() {
  if ('showOpenFilePicker' in window) {
    // showOpenFilePicker может выбрать несколько файлов, просто
    // получите первый из них (с разрушением массива).
    const [file] = window.showOpenFilePicker();

    let result = await file.queryPermission({ mode: 'readwrite' });
    if (result === 'prompt') {
      result = await file.requestPermission({ mode: 'readwrite' });
    }

    return result === 'granted';
  }

  // Если вы попали сюда, это означает, что API не поддерживается.
  return false;
}
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в соответствующем разделе CanIUse (**https://oreil.ly/AfNpL**).

Обсуждение

Функция `queryPermission` возвращает либо `granted` (разрешение было предоставлено ранее), либо `denied` (доступ запрещен), либо `prompt` (необходимо запросить разрешение).

Запрашиваемый режим — `readwrite`; это означает, что браузер может выполнять запись в вашу локальную файловую систему, если вы предоставите соответствующее разрешение. Вот почему проверка разрешений важна с точки зрения безопасности и конфиденциальности.

`queryPermission` проверяет только разрешение и не показывает `prompt`. Если оно возвращается как `prompt`, вы можете вызвать метод `requestPermission`, который покажет запрос на разрешение в браузере. Файл считается доступным для записи, если любой из вызовов возвращается как `granted`.

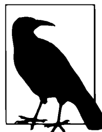
10.6. Экспорт данных API в файл

Задача

Вы запрашиваете данные в формате JSON из API и хотите предоставить пользователю возможность загрузить необработанные данные JSON.

Решение

Позвольте пользователю выбрать выходной файл, а затем запишите данные JSON в локальную файловую систему.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в соответствующем разделе CanIUse (**https://oreil.ly/tsT_j**).

Сначала определите вспомогательную функцию, которая показывает средство выбора файла и возвращает выбранный файл (пример 10.13).

Пример 10.13. Выбор выходного файла

```
/**
 * Показывает средство выбора файлов и возвращает обработчик выбранного файла.
 * @returns обработчик выбранного файла или null, если пользователь нажмет Cancel.
 */
async function selectOutputFile() {
  // Убедитесь, что API поддерживается в этом браузере.
  if (!('showSaveFilePicker' in window)) {
```

```
    return null;
  }

  try {
    return window.showSaveFilePicker({
      // Имя выходного файла по умолчанию.
      suggestedName: 'users.json',

      // Ограничьте доступные расширения файлов.
      types: [
        { description: "JSON", accept: { "application/json": [".json"] } }
      ]
    });
  } catch (error) {
    // Если пользователь нажимает Cancel, генерируется исключение.
    // В этом случае возвращает значение null - признак, что файл не был выбран.
    return null;
  }
}
```

Затем определите функцию, которая использует эту вспомогательную функцию, и выполните фактический экспорт (пример 10.14).

Пример 10.14. Экспорт данных в локальный файл

```
async function exportData(data) {
  // Используйте вспомогательную функцию, определенную ранее.
  const outputFile = await selectOutputFile();

  // Продолжайте только в том случае, если выходной файл действительно был выбран.
  if (outputFile) {
    try {
      // Подготовьте доступный для записи поток, который будет использоваться
      // для сохранения файла на диск.
      const stream = await outputFile.createWritable();

      // Запишите JSON в поток в удобочитаемом формате.
      await stream.write(JSON.stringify(userList, null, 2));
      await stream.close();

      // Отобразите сообщение об успешном завершении.
      document.querySelector('#export-success').classList.remove('d-none');
```

```
    } catch (error) {
      console.error(error);
    }
  }
}
```

Обсуждение

Это хороший подход, позволяющий пользователю создавать резервные копии или экспортировать свои данные из вашего приложения. Некоторые нормативные акты, такие как Общие правила защиты данных (General Data Protection Regulation, GDPR) в Европейском союзе, требуют, чтобы данные пользователя были доступны для скачивания[1].

В этом случае текстовые данные записываются в поток, который имеет тип `FileSystemWritableFileStream`. Такие потоки также поддерживают запись `ArrayBuffer`, `TypedArray`, `DataView` и Blob-объектов.

Для того чтобы создать текст для записи в файл, `exportData` вызывает `JSON.stringify` с некоторыми дополнительными аргументами. Второй `null`-аргумент — это функция `replacer`, которую вы видели в *главе 2*. Этот аргумент необходимо указать, чтобы предоставить третий аргумент, который определяет количество применяемых пробелов в отступах. Это создает более удобочитаемый формат вывода.

На момент написания книги этот API все еще считался экспериментальным. Вам не следует использовать его в рабочем приложении, пока он не получит лучшую поддержку браузеров.

10.7. Экспорт данных API со ссылкой для скачивания

Задача

Вы хотите обеспечить функциональность экспорта, но не хотите беспокоиться о разрешениях файловой системы, как в рецепте 10.6.

Решение

Поместите данные API в Blob-объект и создайте URL-адрес объекта, который будет использоваться в качестве атрибута `href` для ссылки. Затем вы сможете экспортировать данные с помощью обычной загрузки файлов из браузера, который не требует разрешений файловой системы.

[1] Российскому разработчику стоит ознакомиться с Федеральным законом от 27.07.2006 № 149-ФЗ "Об информации, информационных технологиях и о защите информации". — *Прим. ред.*

Сначала добавьте на страницу ссылку-заполнитель, которая потом станет ссылкой для экспорта (пример 10.15).

Пример 10.15. Экспортная ссылка-заполнитель

```
<a id="export-link" download="users.json">Export User Data</a> ❶
```

❶ Атрибут download предоставляет имя файла по умолчанию, которое будет использоваться при загрузке.

После того как вы получите данные из API и отобразите их в пользовательском интерфейсе, создайте URL-адрес Blob-объекта (пример 10.16).

Пример 10.16. Подготовка экспортной ссылки

```
const exportLink = document.querySelector('#export-link');

async function getUserData() {
  const response = await fetch('/api/users');
  const users = await response.json();

  // Отображает пользовательские данные в пользовательском интерфейсе, предполагая,
  // что у вас где-то есть функция renderUsers, которая это делает.
  renderUsers(users);

  // Очистите предыдущие экспортные данные, если они существуют.
  const currentUrl = exportLink.href;
  if (currentUrl) {
    URL.revokeObjectURL(currentUrl);
  }

  // Нужен Blob-объект для создания URL-адреса обьекта
  const blob = new Blob([JSON.stringify(userList, null, 2)], {
    type: 'application/json'
  });

  // URL-адрес обьекта ссылается на содержимое Blob-обьекта - укажите это в ссылке.
  const url = URL.createObjectURL(blob);
  exportLink.href = url;
}
```

Обсуждение

Этот метод экспорта не требует специального разрешения. При нажатии на ссылку с установленным URL-адресом объекта содержимое Blob-объекта загружается в виде файла с использованием предложенного имени файла users.json.

Большой двоичный объект (binary large object, Blob) — это специальный объект, содержащий некоторые данные. Обычно это двоичные данные, такие как файл или изображение, но вы также можете создать большой двоичный объект со строковым содержимым, что и делается в этом рецепте.

Blob-объект хранится в памяти, и URL-адрес созданного объекта ссылается на него. Как только URL-адрес объекта задан в элементе `link`, он становится ссылкой для загрузки при экспорте. Ссылка при нажатии кнопки URL-адреса объекта возвращает необработанные строковые данные. Поскольку ссылка имеет атрибут `download`, она загружается в локальный файл.

Для того чтобы предотвратить утечку памяти, очистите старый URL-адрес, вызвав `URL.revokeObjectURL` и передав URL-адрес объекта в качестве аргумента. Вы можете сделать это, как только URL-адрес объекта вам больше не понадобится — например, после того, как пользователь загрузит файл или перед тем, как он покинет страницу.

10.8. Загрузка файла с помощью перетаскивания

Задача

Вы хотите разрешить пользователю перетаскивать файл, например изображение, а затем загружать этот файл в удаленный сервис.

Решение

Передайте полученный файловый объект в Fetch API в обработчике события `drop` (пример 10.17).

Пример 10.17. Загрузка перетащенного файла

```
const target = document.querySelector('.drop-target');

target.addEventListener('drop', event => {
  // Отмените событие перетаскивания. В противном случае браузер покинет страницу
  // и перейдет непосредственно к файлу.
  event.preventDefault();

  // Получите данные выбранного файла.
  const [item] = event.dataTransfer.items;
  const file = item.getAsFile();
```

```
  if (file.type.startsWith('image/')) {
    fetch('/api/uploadFile', {
      method: 'POST',
      body: file
    });
  }
});

// Вам также необходимо отменить событие перетаскивания, чтобы предотвратить
// замену файлом всего содержимого страницы.
target.addEventListener('dragover', event => {
  event.preventDefault();
});
```

Обсуждение

Когда вы вызываете `getAsFile` для объекта передачи данных, вы получаете файловый объект. Файл получается из Blob-объекта, поэтому вы можете использовать Fetch API для отправки содержимого файла (Blob-объекта) на удаленный сервер.

В этом примере проверяется MIME-тип загружаемого файла, и он будет загружен только в том случае, если это файл изображения.

ГЛАВА 11

Интернационализация

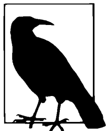
11.0. Введение

Современные браузеры включают в себя надежный API интернационализации (Internationalization API). На самом деле это целый набор API, ориентированных на задачи, связанные с языком, или локалью, такие как:

- форматирование дат и времени;
- форматирование чисел;
- валюты;
- правила плюрализации (правила множественного числа).

До появления этого API вам, возможно, приходилось обращаться к сторонним библиотекам, таким как Moment.js (для дат и времени) или Numeral.js (для чисел). Однако современные браузеры сами поддерживают многие из тех же вариантов использования, и в вашем приложении эти библиотеки могут больше не понадобиться.

Большинство из этих API используют концепцию *локали* (locale), которая обычно представляет собой комбинацию языка и региона. Например, локалью для английского языка в США является `en-US`, а для канадского английского — `en-CA`. Вы можете применять их с локалью по умолчанию, которая используется браузером, или указать конкретный языковой стандарт, чтобы для вашего желаемого региона отформатировать данные соответствующим образом.



В разработке находится новый JavaScript date and time API, который называется Temporal. На момент написания книги это было предложение ECMAScript. В ближайшем будущем он может стать частью языка, но пока что в этой книге будет рассмотрен стандартный Date API.

11.1. Форматирование даты

Задача

Вы хотите отобразить объект даты в формате, соответствующем языковому стандарту пользователя (пользовательской локали).

Решение

Используйте `Intl.DateTimeFormat` для форматирования объекта `Date` в строковое значение. Создайте форматируемый объект с двумя аргументами: желаемая локаль и объект параметров, в котором вы можете указать стиль форматирования. Для дат поддерживаются следующие стили форматирования (примеры приведены для локали `en-US`):

- `short`: 10/16/23;
- `medium`: Oct 16, 2023;
- `long`: October 16, 2023;
- `full`: Monday, October 16, 2023.

Для того чтобы узнать текущую локаль пользователя, вы можете проверить значение свойства `navigator.language` (пример 11.1).

Пример 11.1. Форматирование даты

```
const formatter = new Intl.DateTimeFormat(navigator.language, { dateStyle: 'long' });
const formattedDate = formatter.format(new Date());
```

Обсуждение

Вы также можете включить информацию о времени из объекта `Date`, указав свойство `timeStyle` в объекте `options` вместе с `dateStyle` (пример 11.2).

Пример 11.2. Форматирование даты и времени

```
const formatter = new Intl.DateTimeFormat(navigator.language, {
dateStyle: 'long', timeStyle: 'long' });
const formattedDateAndTime = formatter.format(new Date());
```

11.2. Получение частей отформатированной даты

Задача

Вы хотите разделить отформатированную дату на отдельные токены. Это полезно, например, если вы хотите по-разному оформить разные части отформатированной даты.

Решение

Используйте метод formatToParts из Intl.DateTimeFormat для форматирования даты и возврата массива токенов (пример 11.3).

Пример 11.3. Получение частей отформатированной даты

```
const formatter = new Intl.DateTimeFormat(navigator.language,
  { dateStyle: 'short' });
const parts = formatter.formatToParts(new Date());
```

Обсуждение

Для короткой даты 10/1/23 объект parts, представленный в примере 11.3, выглядит так, как показано в примере 11.4.

Пример 11.4. Отформатированные части даты

```
[
  { type: 'month', value: '10' },
  { type: 'literal': value: '/' },
  { type: 'day': value: '1' },
  { type: 'literal', value: '/' },
  { type: 'year', value: '23' }
]
```

11.3. Форматирование относительной даты

Задача

Вы хотите отформатировать разницу между заданной датой и сегодняшним днем в приблизительном, удобочитаемом формате. Например, вам нужна отформатированная строка типа "2 дня назад" ("2 days ago") или "через 3 месяца" ("in 3 months").

Решение

Используйте Intl.RelativeTimeFormat. В нем есть метод format, который вы вызываете со значением смещения, например -2 (в прошлом) или 3 (в будущем), и единицей измерения, такой как "день" ("day"), "месяц" ("month") и т. д. Например, при вызове format(-2, *day*) в локали en-US будет получена строка "2 дня назад" ("2 days ago").

На самом деле это двухэтапный процесс. Intl.RelativeTimeFormat напрямую не вычисляет разницу между двумя датами. Скорее всего, вам нужно сначала определить

смещение и единицу измерения для перехода к методу `format`. Идея состоит в том, чтобы найти наибольшую единицу измерения, на которую отличаются две даты.

Сначала создайте вспомогательную функцию, которая возвращает объект, содержащий смещение и единицу измерения, как показано в примере 11.5.

Пример 11.5. Поиск смещения и единицы измерения

```
function getDateDifference(fromDate) {
  const today = new Date();
  if (fromDate.getFullYear() !== today.getFullYear()) {
    return { offset: fromDate.getFullYear() - today.getFullYear(), unit: 'year' };
  } else if (fromDate.getMonth() !== today.getMonth()) {
    return { offset: fromDate.getMonth() - today.getMonth(), unit: 'month' };
  } else {
    // Вы могли бы даже перейти к более детальному анализу:
    // вплоть до часов, минут или секунд!
    return { offset: fromDate.getDate() - today.getDate(), unit: 'day' };
  }
}
```

Эта функция возвращает объект с двумя свойствами: `offset` и `unit`, которые можно передать в формат `Intl.RelativeTimeFormat` (пример 11.6).

Пример 11.6. Форматирование относительной даты

```
function getRelativeDate(fromDate) {
  const { offset, unit } = getDateDifference(fromDate);
  const format = new Intl.RelativeTimeFormat();
  return format.format(offset, unit);
}
```

Вот ожидаемый результат, если вы вызываете эту функцию с заданной датой 7 октября 2023 года (имейте в виду, что при создании объектов `Date` таким образом месяцы начинаются с 0, а дни — с 1):

- October 1, 2023: `getRelativeDate(new Date(2023, 9, 1))`: "6 days ago";
- May 2, 2023: `getRelativeDate(new Date(2023, 4, 2))`: "5 months ago";
- June 2, 2025: `getRelativeDate(new Date(2025, 5, 2))`: "in 2 years".

Обсуждение

`getDateDifference` работает путем сравнения года, месяца и дня (в указанном порядке) заданной даты с сегодняшней датой, пока не будет найдено значение, которое

не соответствует. Затем он возвращает разницу и название единицы измерения, которые передаются в `Intl.RelativeTimeFormat`.

Функция `getRelativeDate` не дает точного относительного времени в месяцах, днях, часах, минутах и секундах. Она дает приблизительное представление о величине разницы во времени. Сравните 2 мая 2023 года с 7 октября 2023 года. Разница составляет 5 месяцев и 5 дней, но `getRelativeDate` указывает только "5 месяцев назад" в качестве приблизительного значения.

11.4. Форматирование чисел

Задача

Вы хотите отформатировать число с помощью разделителей тысяч и десятичных знаков в соответствии с языковым стандартом.

Решение

Передайте число в метод `format` формата `Intl.NumberFormat`. Этот метод возвращает строку, содержащую отформатированное число.

По умолчанию в `Intl.NumberFormat` используется локаль по умолчанию[1] (предположим, что локаль по умолчанию в примере 11.7 — это `en-US`).

Пример 11.7. Форматирование числа в соответствии с локалью по умолчанию

```
// Выводит '5,200.55' для локали en-US.
console.log(
  new Intl.NumberFormat().format(5200.55)
);
```

Вы также можете указать другой языковой стандарт для конструктора `Intl.NumberFormat` (пример 11.8).

Пример 11.8. Форматирование числа в соответствии с локалью de-DE

```
// Выводит '5.200,55'.
console.log(
  new Intl.NumberFormat('de-DE').format(5200.55)
);
```

[1] Если у вас установлена операционная система (ОС) с поддержкой вашего национального языка, то локалью по умолчанию будет ваш национальный язык. Так, в ОС с локалью русского языка пример 11.7 выведет `5 200,55`. — *Прим. ред.*

Обсуждение

`Intl.NumberFormat` применяет правила, специфичные для конкретной локали, для форматирования отдельных чисел. Вы также можете использовать его для форматирования диапазона чисел, передав два значения в `formatRange`, как показано в примере 11.9.

Пример 11.9. Форматирование диапазона чисел

```
// Выводит '1,000-5,000' для локали en-US.
console.log(
  new Intl.NumberFormat().formatRange(1000, 5000)
);
```

11.5. Округление знаков после точки

Задача

Вы хотите взять дробное число, которое может содержать много знаков после точки, и округлить его до заданного количества знаков.

Решение

Используйте параметр `maximumFractionDigits`, чтобы указать количество цифр после точки. В примере 11.10 показано, как округлять числа до двух знаков после точки.

Пример 11.10. Округление числа

```
function roundToTwoDecimalPlaces(number) {
  const format = new Intl.NumberFormat(navigator.language, {
    maximumFractionDigits: 2
  });

  return format.format(number);
}

// Выводит "5.49".
console.log(roundToTwoDecimalPlaces(5.49125));

// Выводит "5.5".
console.log(roundToTwoDecimalPlaces(5.49621));
```

11.6. Форматирование ценового диапазона

Задача

Вы имеете массив цен, хранящихся в виде чисел, и хотите создать отформатированный диапазон цен, который отражает минимальную и максимальную цены в массиве.

Решение

Определите минимальную и максимальную цены, затем укажите параметр `style: currency` при создании формата `Intl.NumberFormat`. Используйте этот `Intl.NumberFormat` для создания диапазона. Вы также можете указать валюту, чтобы получить соответствующий символ в выходных данных. Наконец, вызовите `formatRange` с нижней и верхней ценовыми границами (пример 11.11).

Пример 11.11. Форматирование ценового диапазона

```
function formatPriceRange(prices) {
  const format = new Intl.NumberFormat(navigator.language, {
    style: 'currency'.

    // Для style: 'currency' нужен код валюты.
    currency: 'USD'
  });
  return format.formatRange(
    // Найдем нижнюю ценовую границу массива.
    Math.min(...prices),

    // Найдем верхнюю ценовую границу массива.
    Math.max(...prices)
  );
}

// выводит '$1.75–$11.00'
console.log(
  formatPriceRange([5.5, 3, 1.75, 11, 9.5])
);
```

Обсуждение

Функции `Math.max` и `Math.min` принимают несколько аргументов и возвращают максимальное или минимальное значения из всего набора этих аргументов. В примере 11.11 используется синтаксис массива `spread` для передачи всех элементов из массива `prices` в функции `Math.max` и `Math.min`.

11.7. Форматирование единиц измерения

Задача

Вы хотите отформатировать число вместе с единицей измерения.

Решение

Используйте стиль `unit` при создании объекта `Intl.NumberFormat` и укажите целевую единицу измерения. В примере 11.12 показано, как отформатировать число в гигабайтах.

Пример 11.12. Форматирование числа в гигабайтах

```
const format = new Intl.NumberFormat(navigator.language, {
  style: 'unit',
  unit: 'gigabyte'
});

// Печатает "1,000 GB".
console.log(format.format(1000));
```

Обсуждение

Вы также можете настроить метку элемента, указав для параметра `unitDisplay` опцию `NumberFormat`. Возможны следующие значения.

`short`

: Выводит сокращенную единицу измерения с пробелом в качестве разделителя: `1,000 GB`.

`narrow`

: Выводит сокращенную единицу измерения без пробела: `1,000GB`.

`long`

: Выводит полное наименование единицы измерения: `1,000 gigabytes`.

11.8. Применение правил плюрализации

Задача

Вы хотите использовать правильную терминологию при описании разного количества элементов. Например, рассмотрим список пользователей. В английском языке вы бы сказали "один пользователь" (в единственном числе), но "три пользователя" (во множественном числе). В других языках существуют более сложные правила, и вы должны быть уверены, что соблюдаете их.

Решение

Используйте `Intl.PluralRules` для выбора правильной строки со словами во множественном числе.

Сначала создайте объект `Intl.PluralRules` с требуемой локалью и вызовите его метод `select` с указанием количества пользователей (пример 11.13).

Пример 11.13. Определение формы множественного числа

```
// Массив пользователей.
const users = getUsers();

const rules = new Intl.PluralRules('en-US');
const form = rules.select(users.length);
```

Метод `select` возвращает строку в зависимости от используемой формы множественного числа и указанной локали. Для локали `en-US` он возвращает либо "one" (когда количество пользователей равно единице), либо "other" (когда количество пользователей не равно единице). Вы можете определять сообщения, используя эти значения в качестве ключа, как показано в примере 11.14.

Пример 11.14. Полное решение с правилами множественного числа

```
function formatUserCount(users) {
  // Варианты сообщения, зависящие от количества
  const messages = {
    one: 'There is 1 user.',
    other: `There are ${users.length} users.`
  };

  // Используйте Intl.PluralRules, чтобы определить,
  // какое сообщение должно отображаться.
  const rules = new Intl.PluralRules('en-US');
  return messages[rules.select(users.length)];
}
```

Обсуждение

Это решение требует заблаговременного знания различных форм, чтобы вы могли определить правильные сообщения.

`Intl.PluralRules` также поддерживает порядковый режим, который работает несколько иначе. Вы можете использовать этот режим для форматирования *порядковых* значений, таких как "1-й", "2-й", "3-й" ("1st", "2nd", "3rd") и т. д. Правила фор-

матирования различаются в зависимости от языка, и вы можете сопоставить их с суффиксами, которые вы применяете к числам.

Например, в локали en-US порядковый Intl.PluralRules возвращает такие значения, как:

- one для чисел, кончающихся на 1 — "1st", "21st" и т. д.;
- two для чисел, кончающихся на 2 — "2nd"," 42nd" и т. д.;
- few для чисел, кончающихся на 3 — "3rd", "33rd" и т. д.;
- other для других чисел — "5th", "47th" и т. д.

11.9. Подсчет символов, слов и предложений

Задача

Вы хотите рассчитать количество символов, слов и предложений в строке, используя специфичные для локали правила.

Решение

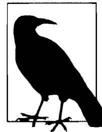
Используйте Intl.Segmenter для разделения строки и подсчета вхождений.

Вы можете создать сегментатор с детализацией графем (отдельных символов), слов или предложений. Степень детализации определяет границы сегментов. Каждый сегмент может иметь только одну степень детализации, поэтому вам понадобятся три сегментатора (пример 11.15).

Пример 11.15. Подсчет количества символов, слов и предложений в строке

```
function getCounts(text) {
  const characters = new Intl.Segmenter(
    navigator.language,
    { granularity: 'grapheme' }
  );

  const words = new Intl.Segmenter(
    navigator.language,
    { granularity: 'word' }
  );

  const sentences = new Intl.Segmenter(
    navigator.language,
    { granularity: 'sentence' }
  );
```

```
    // Преобразуйте каждый сегмент в массив, затем получите его длину.
    return {
      characters: [...characters.segment(text)].length,
      words: [...words.segment(text)].length,
      sentences: [...sentences.segment(text)].length
    };
}
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/OL9G0**).

Обсуждение

Когда вы вызываете `segment` для сегментатора с некоторым текстом, он возвращает повторяемый объект, содержащий все сегменты. Существует несколько способов получить длину элементов в этой итеративной таблице, но в этом примере используется синтаксис массивов `spread`, который создает массив, содержащий все элементы. Затем вам просто нужно получить длину каждого массива.

Возможно, вы уже решали эту проблему в прошлом, используя метод разделения строки `split`. Например, вы могли бы разделить строку по пробелам, чтобы получить массив слов и подсчитать количество слов. Этот подход может работать на вашем языке, но преимущество использования `Intl.Segmenter` заключается в том, что он использует правила данного языка для разделения слов и предложений.

11.10. Форматирование списков

Задача

У вас есть массив элементов, которые вы хотите отобразить в виде списка, разделенного запятыми. Например, массив пользователей отображается как "пользователь1, пользователь2 и пользователь3" ("user1, user2, and user3.").

Решение

Используйте `Intl.ListFormat`, чтобы объединить элементы в список с помощью правил вашей локали. В примере 11.16 используется массив пользователей, у каждого из которых есть свойство `username`.

Пример 11.16. Форматирование списка объектов-пользователей

```
function getUserListString(users, locale = 'en-US') {
  // Локаль для ListFormat можно настроить.
  const listFormat = new Intl.ListFormat(locale);
  return listFormat.format(users.map(user => user.username));
}
```

Обсуждение

`Intl.ListFormat` при необходимости добавляет слова и знаки препинания. Например, в локали `en-US` вы получите следующее:

- 1 пользователь: " user1";
- 2 пользователя: " user1 and user2";
- 3 пользователя: "user1, user2, and user3".

Вот еще один пример использования локали `de-DE`:

- 1 пользователь: "user1";
- 2 пользователя: "user1 und user2";
- 3 пользователя: " user1, user2 und user3".

Обратите внимание на применение "und" вместо "and", а также на то, что в третьем случае после user2 нет запятой, как в `en-US`. Это связано с тем, что в немецкой грамматике не используется эта запятая (так называемая "оксфордская запятая").

Как вы можете видеть, использование `Intl.ListFormat` намного надежнее, чем использование метода `join` для объединения значений массива с помощью запятой. Этот метод, конечно, не учитывает специфические для региона правила локали.

11.11. Сортировка массива имен

Задача

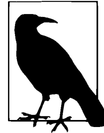
У вас есть массив имен, которые вы хотите отсортировать, используя правила сортировки, специфичные для конкретной локали.

Решение

Создайте `Intl.Collator` для обеспечения логики сравнения, затем используйте его функцию `compare` для передачи в `Array.prototype.sort` (пример 11.17). Эта функция сравнивает две строки. Она возвращает отрицательное значение, если первая строка идет перед второй; ноль, если строки равны, или положительное значение, если первая строка идет после второй.

Пример 11.17. Сортировка массива имен с помощью Intl.Collator

```
const names = [
  'Елена',
  'Марио',
  'Андре',
  'Рене',
  'Лео',
  'Ольга',
  'Гектор',
]

const collator = new Intl.Collator();
names.sort(collator.compare);
```



`Collator` может возвращать любое отрицательное или положительное значение, а не только –1 или 1.

Обсуждение

Это краткий способ сортировки массива строк. Перед использованием `Intl.Collator` вы могли бы выполнить что-то вроде примера 11.18.

Пример 11.18. Прямая сортировка массива строк

```
names.sort((a, b) => a.localeCompare(b));
```

Этот код работает нормально, но с одним существенным отличием: вы не можете указать, какие правила сортировки следует применять при сравнении строк в конкретной локали. Еще одно преимущество `Intl.Collator` заключается в его гибкости. Вы можете точно настроить логику, используемую для сравнения строк.

Например, рассмотрим массив `[1, 2, 20, 3]`. При использовании `collator` по умолчанию, это был бы порядок сортировки, поскольку он использует логику сравнения строк. Вы можете передать параметр `numeric: true` в `Intl.Collator`, и отсортированный массив затем становится `[1, 2, 3, 20]`.

ГЛАВА 12

Веб-компоненты

12.0. Введение

Веб-компоненты — это набор технологий для создания новых HTML-элементов с их собственным поведением. Это поведение инкапсулируется в *пользовательский элемент*.

Создание компонента

Вы можете создать веб-компонент, определив класс, расширяющий `HTMLElement`, как показано в примере 12.1.

Пример 12.1. Базовый веб-компонент

```
class MyComponent extends HTMLElement {
  connectedCallback() {
    this.textContent = 'Привет из MyComponent';
  }
}
```

Когда вы добавляете пользовательский элемент в DOM, браузер вызывает метод `connectedCallback`. Обычно именно здесь находится бóльшая часть логики вашего компонента. Это один из обратных вызовов жизненного цикла. Далее перечислены другие *обратные вызовы жизненного цикла.*

`disconnectedCallback`

Вызывается при удалении пользовательского элемента из DOM. Это хорошее действие для выполнения очистки, например удаления прослушивателей событий.

`attributeChangedCallback`

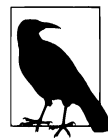
Вызывается при изменении одного из отслеживаемых атрибутов элемента.

Регистрация пользовательского элемента

После того как вы создали свой класс пользовательского элемента, вы должны зарегистрировать его в браузере, прежде чем использовать в HTML-документе. Вы можете зарегистрировать свой пользовательский элемент, вызвав функцию `define` для глобального объекта `customElements`, как показано в примере 12.2.

Пример 12.2. Регистрация пользовательского элемента в браузере

```
customElements.define('my-component', MyComponent);
```

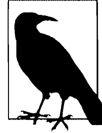


Если вы попытаетесь определить пользовательский элемент, который уже был определен, браузер выдаст сообщение об ошибке. Если у вас есть такая возможность, попытайтесь вызвать `customElements.get('my-component')`, чтобы проверить, был ли он определен ранее. Если возвращается значение `undefined`, можно с уверенностью вызвать `customElements.define`.

Как только вы зарегистрируете элемент, вы сможете использовать его как любой другой HTML-элемент (пример 12.3).

Пример 12.3. Использование пользовательского элемента

```
<my-component></my-component>
```



Пользовательские элементы всегда должны иметь имена, написанные через дефис. Этого требует спецификация. Они также всегда должны иметь закрывающий тег, даже если в них нет дочернего содержимого.

Шаблоны

Существует несколько способов включить HTML-разметку в веб-компонент. Например, в `connectedCallback` вы можете вручную создавать элементы, вызывая `document.createElement` и добавляя их вручную.

Вы также можете указать разметку компонента с помощью элемента `<template>`. Он содержит некоторый HTML-код, который вы будете использовать для `connectedCallback`, чтобы предоставить вашему компоненту некоторый контент. Эти шаблоны очень просты — они не поддерживают привязку данных, различную интерполяцию или какую-либо логику. Они служат только отправной точкой для внедрения HTML-контента. В `connectedCallback` вы можете выбирать элементы, задавать динамические значения и добавлять прослушивателей событий по мере необходимости.

Слоты

<slot> — это специальный элемент, который вы можете использовать в шаблоне. Слот — это заполнитель для некоторого передаваемого дочернего содержимого. Компонент может иметь слот по умолчанию, а также один или несколько *именованных* слотов. Вы можете использовать именованные слоты для размещения нескольких фрагментов содержимого внутри вашего компонента.

В примере 12.4 показан простой шаблон с именованными слотами и слотом по умолчанию.

Пример 12.4. Шаблон со слотами

```
<template>
  <h2><slot name="name"></slot></h2>
  <slot></slot>
</template>
```

Предположим, что этот шаблон используется в компоненте <author-bio>, как показано в примере 12.5.

Пример 12.5. Установка содержимого для слотов

```
<author-bio>
  <span slot="name">Джон Доу</span>
  <p>Джон - великий автор, написавший множество книг.</p>
</author-bio>
```

В дочернем содержимом компонента вы можете указать атрибут slot, соответствующий именованному слоту в шаблоне компонента. Элемент span, содержащий текст "Джон Доу", будет помещен в слот с именем "name", внутри элемента h2. Любой другой дочерний контент, без элемента slot, помещается в слот по умолчанию (тот, у которого нет названия).

Shadow DOM

Shadow DOM — это набор элементов, изолированных от остальной части основного DOM. Веб-компоненты широко используют shadow DOM. Одним из основных преимуществ использования shadow DOM является возможность применения CSS-стилей с ограниченной областью действия. Любые стили, которые вы определяете в shadow DOM, применяются *только* к элементам внутри этого shadow DOM. К другим элементам в документе, даже если они обычно соответствуют селектору CSS-правила, CSS не применяется.

Это определение области стилей выполняется в обоих направлениях. Если у вас на странице есть глобальные стили, они не будут применяться ни к одному из элементов shadow DOM.

Shadow DOM, созданный путем присоединения *shadow root* к веб-компоненту, может быть открытым или закрытым. Когда shadow DOM открыт, вы можете получать доступ к его элементам и изменять их с помощью JavaScript. Когда он закрыт, свойство `shadowRoot` веб-компонента имеет значение `null`, поэтому вы не можете получить доступ к содержимому внутри него.

Light DOM

Однако использование shadow DOM совершенно необязательно. *Light DOM* относится к обычному, неинкапсулированному DOM внутри веб-компонента. Поскольку light DOM не изолируется от остальной части страницы, к его дочерним элементам будут применены глобальные стили.

12.1. Создание компонента для отображения сегодняшней даты

Задача

Вам нужен веб-компонент, который форматирует и отображает сегодняшнюю дату в соответствии с локалью браузера.

Решение

Используйте `Intl.DateTimeFormat` внутри веб-компонента для форматирования текущей даты (пример 12.6).

Пример 12.6. Пользовательский элемент, который форматирует текущую дату

```
class TodaysDate extends HTMLElement {
  connectedCallback() {
    const formatter = new Intl.DateTimeFormat(
      navigator.language,
      { dateStyle: 'full' }
    );

    this.textContent = formatter.format(new Date());
  }
}

customElements.define('todays-date', TodaysDate);
```

Теперь вы можете отображать текущую дату с помощью этого веб-компонента без каких-либо атрибутов или дочернего содержимого, как показано в примере 12.7.

Пример 12.7. Отображение текущей даты

```
<p>
  Сегодя: <todays-date></todays-date>
</p>
```

Обсуждение

Когда элемент <todays-date> попадает в DOM, браузер вызывает connectedCallback. В connectedCallback класс TodaysDate форматирует текущую дату с помощью объекта Intl.DateTimeFormat, который вы, возможно, помните из *главы 11*. connectedCallback устанавливает эту отформатированную строку даты в качестве textContent элемента, который наследуется от Node (предка HTMLElement).

12.2. Создание компонента для форматирования пользовательской даты

Задача

Вам нужен веб-компонент, который форматирует произвольное значение даты.

Решение

Присвойте веб-компоненту атрибут date и используйте его для создания отформатированной даты (пример 12.8). Вы можете отслеживать изменения этого атрибута и переформатировать дату, если атрибут даты изменится.

Пример 12.8. Пользовательский компонент даты

```
class DateFormatter extends HTMLElement {
  // Браузер будет уведомлять компонент об изменениях только через
  // attributeChangedCallback для атрибутов, которые перечислены здесь.
  static observedAttributes = ['date'];

  constructor() {
    super();

    // Создайте формат здесь, чтобы вам не приходилось создавать его
    // заново каждый раз, когда меняется дата.
```

```
    this.formatter = new Intl.DateTimeFormat(
      navigator.language,
      { dateStyle: 'full' }
    );
  }

  /**
   * Форматирует данные, представленные текущим значением атрибута 'date',
   * если таковой имеется.
   */
  formatDate() {
    if (this.hasAttribute('date')) {
      this.textContent = this.formatter.format(
        new Date(this.getAttribute('date'))
      );
    } else {
      // Если дата не указана, ничего не выводится.
      this.textContent = '';
    }
  }

  attributeChangedCallback() {
    // Просматривается только один атрибут, поэтому здесь должно быть изменение
    // атрибута даты. Обновите отформатированную дату, если таковая имеется.
    this.formatDate();
 }

  connectedCallback() {
    // Элемент был только что добавлен. Покажите начальную
    // отформатированную дату, если таковая имеется.
    this.formatDate();
  }
}

customElements.define('date-formatter', DateFormatter);
```

Теперь вы можете передать дату в атрибут `date`, чтобы она была отформатирована в соответствии с локалью пользователя (пример 12.9).

Пример 12.9. Использование элемента date-formatter

```
<date-formatter date="2023-10-16T03:52:49.955Z"></date-formatter>
```

Обсуждение

Этот рецепт дополняет рецепт 12.1, добавляя возможность указывать собственную дату с помощью атрибута.

По умолчанию, если вы измените значение атрибута, передаваемого пользовательскому элементу, ничего не произойдет. Логика в `connectedCallback` запускается только при первом добавлении компонента в DOM. Для того чтобы заставить компонент реагировать на изменения атрибутов, вы можете реализовать метод `attributeChangedCallback`. В компоненте `date-formatter` этот метод использует обновленный атрибут `date` и создает новую отформатированную дату. Когда атрибут изменяется, браузер вызывает этот метод с именем атрибута, старым значением и новым значением.

Однако это само по себе не решит проблему. Если вы просто реализуете `attributeChangedCallback`, вы все равно не получите уведомления об изменениях атрибута, поскольку браузер вызывает `attributeChangedCallback` только для *наблюдаемых* атрибутов. В этом случае вы можете определить подмножество атрибутов таким образом, чтобы браузер вызывал `attributeChangedCallback` лишь для тех атрибутов, которые вас интересуют. Для того чтобы определить эти атрибуты, добавьте статическое свойство `observedAttributes` в ваш класс компонентов. Это должен быть массив имен атрибутов.

В компоненте `date-formatter` вы просматриваете только один атрибут (атрибут `date`). Из-за этого в `attributeChangedCallback` вам не нужно проверять аргумент `name`, поскольку вы уже знаете, что изменился атрибут `date`. В компоненте с несколькими отслеживаемыми атрибутами вы можете проверить атрибут по названию, чтобы узнать, какой атрибут изменился.

Если вы измените значение атрибута `date` с помощью JavaScript, будет запущен `attributeChangedCallback` для обновления отформатированной даты.

12.3. Создание компонента обратной связи

Задача

Вы хотите создать многоразовый компонент, в котором пользователь может оставить отзыв о том, полезна ли страница или нет.

Решение

Создайте веб-компонент для отображения кнопок обратной связи и отправки пользовательского события, когда пользователь нажимает на одну из них.

Сначала вам нужно создать элемент шаблона, содержащий разметку, используемую этим компонентом, как показано в примере 12.10.

Пример 12.10. Создание шаблона

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    .feedback-prompt {
      display: flex;
      align-items: center;
      gap: 0.5em;
    }

    button {
      padding: 0.5em 1em;
    }
  </style>

  <div class="feedback-prompt">
    <p>Было ли это полезно?</p>
    <button type="button" data-helpful="true">Да</button>
    <button type="button" data-helpful="false">Нет</button>
  </div>
`;
```

Этот компонент использует shadow DOM, содержащий разметку шаблона (пример 12.11). Правила стиля CSS применимы только к этому компоненту.

Пример 12.11. Реализация компонента

```
class FeedbackRating extends HTMLElement {
  constructor() {
    super();

    // Создайте shadow DOM и отобразите в нем шаблон.
    const shadowRoot = this.attachShadow({ mode: 'open' });
    shadowRoot.appendChild(template.content.cloneNode(true));
  }

  connectedCallback() {
    this.shadowRoot.querySelector('.feedback-prompt').addEventListener('click',
```

```
    event => {
      const { helpful } = event.target.dataset;

      if (typeof helpful !== 'undefined') {
        // Как только выбрана опция обратной связи, скройте кнопки
        // и покажите подтверждение.
        this.shadowRoot.querySelector('.feedback-prompt').remove();
        this.shadowRoot.textContent = 'Спасибо за Вашу обратную связь!';

        // В JavaScript нет функции типа "parseBoolean", поэтому
        // преобразуйте значение string в соответствующее логическое значение.
        this.helpful = helpful === 'true';

        // Отправьте пользовательское событие, чтобы ваше приложение могло
        // получать уведомления при нажатии кнопки обратной связи.
        this.shadowRoot.dispatchEvent(new CustomEvent('feedback', {
          composed: true, // Это необходимо для того, чтобы "избежать"
                          // границы shadow DOM.
          bubbles: true   // Это необходимо для распространения вверх по DOM.
        }));
      }
    });
  }
}

customElements.define('feedback-rating', FeedbackRating);
```

Теперь вы можете добавить этот компонент обратной связи в свое приложение (пример 12.12).

Пример 12.12. Использование компонента обратной связи для оценки страницы

```
<h2>Обратная связь</h2>
<feedback-rating></feedback-rating>
```

Вы можете прослушивать уведомления о пользовательском событии обратной связи feedback, когда пользователь выбирает опцию обратной связи (пример 12.13). Вам решать, что делать с этой информацией; возможно, вы захотите отправить данные с помощью Fetch API в конечную точку для анализа.

Пример 12.13. Прослушивание события обратной связи

```
document.querySelector('feedback-rating').addEventListener('feedback', event => {
  // Получаем значение свойства "helpful" компонента обратной связи и
  // отправляем его в конечную точку с запросом POST.
  fetch('/api/analytics/feedback', {
    method: 'POST',
    body: JSON.stringify({ helpful: event.target.helpful }),
    headers: {
      'Content-Type': 'application/json'
    }
  });
});
```

Обсуждение

Компонент `feedback-rating` содержит приглашение и две кнопки. Пользователь нажимает одну из двух кнопок в зависимости от того, считает ли он содержимое веб-сайта полезным или нет.

Прослушиватель событий `click` использует делегирование событий. Вместо того чтобы добавлять прослушивателя к каждой кнопке, добавляется один прослушиватель, который реагирует на щелчок в любом месте приглашения к обратной связи. Если выбранный элемент не имеет атрибута `data-helpful`, то пользователь, должно быть, не нажимал на кнопку обратной связи, поэтому ничего не нужно делать. В противном случае он преобразует строковое значение в логическое и устанавливает его в качестве свойства пользовательского элемента, которое может быть получено позже. Также генерируется событие, которое вы можете прослушать в другом месте.

Для того чтобы это событие перешло из shadow DOM в обычный DOM, вы должны установить параметр `composed: true`. В противном случае ни один прослушиватель событий, добавленный вами к пользовательскому элементу, не будет запущен.

Как только это событие будет запущено, вы можете проверить сам элемент обратной связи (доступный как свойство `event.target`) на наличие свойства `helpful`, позволяющего определить, на какую кнопку обратной связи нажал пользователь.

Поскольку стили и разметка содержатся в shadow DOM, правила CSS не влияют ни на какие элементы за пределами shadow DOM. Это важно отметить, поскольку в противном случае селектор элемента, например кнопки, придал бы стиль каждой кнопке на странице. Так как стили ограничены областью действия, они применяются только к кнопкам внутри пользовательского элемента.

Однако содержимое, передаваемое в слот компонента, *может* быть оформлено в соответствии с глобальными правилами CSS. Выделенный контент не перемещается в shadow DOM, а остается в стандартном, т. е. light DOM.

12.4. Создание компонента профильной карточки

Задача

Вы хотите создать многоразовый компонент карточки для отображения профиля пользователя.

Решение

Используйте слоты в вашем веб-компоненте для передачи контента в определенные области приложения.

Сначала определите шаблон с некоторыми стилями и разметкой, как показано в примере 12.14.

Пример 12.14. Шаблон профильной карточки

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    :host {
      display: grid;
      border: 1px solid #ccc;
      border-radius: 5px;
      padding: 8px;
      grid-template-columns: auto 1fr;
      column-gap: 16px;
      align-items: center;
      margin: 1rem;
    }

    .photo {
      border-radius: 50%;
      grid-row: 1 / span 3;
    }

    .name {
      font-size: 2rem;
      font-weight: bold;
    }

    .title {
      font-weight: bold;
```

```
    }
  </style>

  <div class="photo"><slot name="photo"></slot></div>
  <div class="name"><slot name="name"></slot></div>
  <div class="title"><slot name="title"></slot></div>
  <div class="bio"><slot></slot></div>
`;
```

В этом шаблоне есть три именованных слота (фотография, имя и титул) и один слот по умолчанию для биографии. Сама реализация компонента довольно минимальна; он просто создает и присоединяет теневой корень к шаблону (пример 12.15).

Пример 12.15. Реализация компонента

```
class ProfileCard extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.appendChild(template.content.cloneNode(true));
  }
}

customElements.define('profile-card', ProfileCard);
```

Для того чтобы использовать компонент, вы можете указать атрибут slot в дочерних элементах, чтобы показать, в какой слот должен помещаться контент (пример 12.16). Элемент биографии, у которого нет атрибута slot, помещается в слот по умолчанию.

Пример 12.16. Использование профессиональной карты

```
<profile-card>
  <img slot="photo" src="/api/portraits/chavez.jpg" />
  <div slot="name">Филлип Чавес</div>
  <div slot="title">Генеральный директор</div>
  <p>Филлип - отличный генеральный директор.</p>
</profile-card>

<profile-card>
  <img slot="photo" src="/api/portraits/lynch.jpg" />
  <div slot="name">Джеми Линч</div>
```

```
  <div slot="title">Вице-президент</div>
  <p>Джеми - отличный вице-президент.</p>
</profile-card>
```

На рис. 12.1 показан результат визуализации компонента "карточка профиля".

Рис. 12.1. Визуализация карточки профиля

Обсуждение

В CSS-стилях вы, возможно, заметили селектор `:host`, который представляет стили, применяемые к *теневому хосту* пользовательского элемента. Это элемент, к которому привязан shadow DOM.

На этом примере вы можете увидеть, как веб-компоненты позволяют создавать повторно используемый контент и макеты. Слоты — это мощный инструмент, который позволяет вставлять контент именно туда, куда необходимо.

12.5. Создание компонента изображения с отложенной загрузкой

Задача

Вам нужен повторно используемый компонент, содержащий изображение, которое не загружается до тех пор, пока оно не появится в окне просмотра.

Решение

Используйте `IntersectionObserver`, чтобы дождаться появления элемента в поле зрения при прокрутке, а затем установите элемент `src` на содержащемся изображении.

Этот рецепт адаптирует рецепт 6.1 из *главы 6*, представляя его решение в веб-компоненте (примеры 12.17 и 12.18).

Пример 12.17. Компонент LazyImage

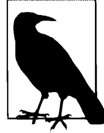
```
class LazyImage extends HTMLElement {
  constructor() {
    super();

    const shadowRoot = this.attachShadow({ mode: 'open' });
    this.image = document.createElement('img');
    shadowRoot.appendChild(this.image);
  }

  connectedCallback() {
    const observer = new IntersectionObserver(entries => {
      if (entries[0].isIntersecting) {
        console.log('Loading image');
        this.image.src = this.getAttribute('src');
        observer.disconnect();
      }
    });

  observer.observe(this);
  }
}
customElements.define('lazy-image', LazyImage);
```

Пример 12.18. Использование компонента LazyImage

```
<lazy-image src="https://placekitten.com/200/138"></lazy-image>
```

Обсуждение

Как только элемент начинает появляться в поле зрения, обратный вызов `IntersectionObserver` получает атрибут `src` и устанавливает его в качестве атрибута `src` изображения, что запускает загрузку изображения.



В этом примере показано, как создать пользовательский элемент, расширяющий встроенный элемент, но при отложенной загрузке изображений он может вам и не понадобиться. Более новые браузеры для тегов `img` поддерживают атрибут `loading="lazy"`, который имеет тот же эффект — изображение не загружается до тех пор, пока оно не появится в поле зрения.

12.6. Создание компонента раскрытия информации

Задача

Вы хотите отобразить или скрыть какой-либо контент, нажав некую кнопку. Например, у вас может быть раздел **Дополнительно** (Advanced) в форме, который по умолчанию скрыт, но который можно развернуть, нажав кнопку.

Решение

Создайте веб-компонент для раскрытия информации (disclosure). Компонент состоит из двух частей: кнопки, которая переключает содержимое, и самого содержимого. У каждой из этих двух частей будет свой слот. Слот по умолчанию будет предназначен для содержимого, а для кнопки будет выделен именованный слот. Этот компонент также можно развернуть или свернуть программным способом, изменив значение его атрибута `open`.

Сначала определите шаблон для компонента раскрытия, как показано в примере 12.19.

Пример 12.19. Шаблон компонента раскрытия

```
const template = document.createElement('template');
template.innerHTML = `
  <div>
    <button type="button" class="toggle-button">
      <slot name="title"></slot>
    </button>
    <div class="content">
      <slot></slot>
    </div>
  </div>
`;
```

Реализация компонента показана в примере 12.20.

Пример 12.20. Реализация компонента раскрытия

```
class Disclosure extends HTMLElement {
  // Следите за атрибутом 'open', чтобы реагировать на изменения.
  static observedAttributes = ['open'];

  constructor() {
    super();
```

```
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.appendChild(template.content.cloneNode(true));

    this.content = this.shadowRoot.querySelector('.content');
  }

  connectedCallback() {
    this.content.hidden = !this.hasAttribute('open');
    this.shadowRoot.querySelector('.toggle-button')
    .addEventListener('click', () => {
      if (this.hasAttribute('open')) {
        // В данный момент содержимое отображается; удалите атрибут 'open',
        // чтобы скрыть содержимое.
        this.removeAttribute('open');
        this.content.hidden = true;
      } else {
        // В данный момент содержимое скрыто; добавьте атрибут 'open',
        // чтобы отобразить содержимое.
        this.setAttribute('open', '');
        this.content.hidden = false;
      }
    });
  }

  attributeChangedCallback(name, oldValue, newValue) {
    // Обновите скрытое состояние содержимого на основе нового значения атрибута.
    if (newValue !== null) {
      this.content.hidden = false;
    } else {
      this.content.hidden = true;
    }
  }
}

// Имя элемента должно быть написано через дефис.
customElements.define('x-disclosure', Disclosure);
```

И последнее — вам нужно добавить немного CSS на страницу. В противном случае дочерний контент будет мигать на странице некоторое время, а затем исчезнет. Это связано с тем, что до регистрации пользовательского элемента его поведение не

задано, и браузер не знает о его слотах. Это означает, что на странице будет отображаться любое дочернее содержимое.

Затем, как только пользовательский элемент определен, дочернее содержимое перемещается в слот и исчезает.

Для того чтобы исправить это, вы можете использовать CSS для скрытия содержимого элемента до тех пор, пока он не будет зарегистрирован с помощью псевдокласса :defined.

Пример 12.21. Устранение проблемы с мерцанием

```
x-disclosure:not(:defined) {
  display: none;
}
```

Сначала содержимое будет скрыто. Как только пользовательский элемент будет определен, он будет показан. Вы не увидите мерцания, потому что содержимое уже перемещено в слот.

Наконец, вы можете использовать элемент раскрытия, как показано в примере 12.22.

Пример 12.22. Использование элемента раскрытия

```
<x-disclosure>
   <div slot="title">Подробности</div>
   Это подробное дочернее содержимое, которое будет развернуто или свернуто
   при нажатии заголовочной кнопки.
</x-disclosure>
```

Кнопка переключения будет содержать текст "Подробности" ("Details"), поскольку он помещен в слот title. Остальное содержимое будет помещено в слот по умолчанию.

Обсуждение

Компонент раскрытия информации использует свой атрибут open, чтобы определить, показывать ли дочерний контент. При нажатии кнопки переключения он добавляет или удаляет атрибут в зависимости от текущего состояния, а затем условно применяет атрибут hidden к дочернему контенту.

Вы также можете программно переключать дочернее содержимое, добавляя или удаляя атрибут open. Это работает, потому что компонент наблюдает за атрибутом open. Если вы измените его с помощью JavaScript или даже в инструментах разработчика браузера, браузер вызывает метод attributeChangedCallback компонента с новым значением.

Атрибут open не имеет значения. Если вы хотите, чтобы содержимое было открыто по умолчанию, просто добавьте атрибут open без значения, как показано в примере 12.23.

Пример 12.23. Отображение содержимого по умолчанию

```
<x-disclosure open>
  <div slot="title">Подробности</div>
   Это подробное дочернее содержимое, которое будет развернуто или свернуто
   при нажатии заголовочной кнопки.
</x-disclosure>
```

Если вы удалите атрибут, аргумент newValue для параметра attributeChangedCallback будет равен null. В этом случае дочернее содержимое будет скрыто путем применения атрибута hidden. Если вы добавите атрибут без значения, как показано в примере 12.23, newValue будет представлять собой пустую строку. В этом случае атрибут hidden будет удален.

12.7. Создание стилизованного компонента кнопки

Задача

Вы хотите создать повторно используемый компонент кнопки с различными вариантами стиля.

Решение

Кнопка будет доступна в трех вариантах:

- вариант по умолчанию, с серым фоном;
- "primary" (основной) вариант с синим фоном;
- "danger" (предупреждение, опасность) вариант с красным фоном.

Сначала создайте шаблон с пользовательским стилем кнопок, а также классы CSS для основного и предупреждающего вариантов, как показано в примере 12.24.

Пример 12.24. Шаблон кнопки

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    button {
      background: #333;
      padding: 0.5em 1.25em;
```

```
      font-size: 1rem;
      border: none;
      border-radius: 5px;
      color: white;
    }

    button.primary {
      background: #2563eb;
    }

    button.danger {
      background: #dc2626;
    }
  </style>

  <button>
    <slot></slot>
  </button>
`;
```

Бóльшая часть этого шаблона — это CSS. Сама разметка для компонента довольно проста: элемент `button` со слотом по умолчанию.

Сам компонент будет поддерживать два атрибута.

`variant`

Название варианта кнопки (`primary` или `danger`).

`type`

Атрибут `type`, который передается в базовый элемент `button`. Установите для него значение `button`, чтобы предотвратить отправку формы (пример 12.25).

Пример 12.25. Компонент button

```
class StyledButton extends HTMLElement {
  static observedAttributes = ['variant', 'type'];

  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.appendChild(template.content.cloneNode(true));
    this.button = this.shadowRoot.querySelector('button');
  }
```

```
  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'variant') {
      this.button.className = newValue;
    } else if (name === 'type') {
      this.button.type = newValue;
    }
  }
}
customElements.define('styled-button', StyledButton);
```

Для того чтобы добавить прослушивателя нажатий, вам на самом деле не нужно выполнять никакой дополнительной работы. Вы можете добавить прослушивателя нажатий к элементу styled-button, и он будет активирован при нажатии основной кнопки благодаря делегированию событий. С помощью делегирования событий вы можете добавить прослушивателя событий к родительскому элементу, и события в его дочерних элементах также будут запускать прослушивателя событий родительского элемента.

Наконец, вот как вы будете использовать компонент styled-button (пример 12.26).

Пример 12.26. Использование компонента стилизованной кнопки

```
<styled-button id="default-button" type="button">По умолчанию</styled-button>
<styled-button id="primary-button" type="button" variant="primary">
  Основной
</styled-button>
<styled-button id="danger-button" type="button" variant="danger">
  Опасность
</styled-button>
```

Обсуждение

Стиль применяется путем установки имени класса в элементе button равным имени параметра. Это приведет к тому, что соответствующее правило CSS применит желаемый цвет фона.

Вам не нужно иметь никакого кода в connectedCallback, чтобы применить класс, потому что браузер вызовет attributeChangedCallback с начальными значениями, а также с любыми последующими обновленными значениями.

Вы можете добавить прослушивателя нажатий к стилизованной кнопке таким же образом, как и к обычной кнопке (пример 12.27).

Пример 12.27. Добавление прослушивателя нажатий

```
<script>
document.querySelector('#default-button').addEventListener('click', () => {
  console.log('Нажата кнопка По умолчанию');
});
</script>

<styled-button id="default-button" type="button">По умолчанию</styled-button>
```

ГЛАВА 13

Элементы интерфейса пользователя

13.0. Введение

Современные браузеры имеют несколько мощных встроенных элементов пользовательского интерфейса, которые вы можете использовать в своем приложении. Ранее для подобных компонентов интерфейса требовались сторонние библиотеки (или вы могли создать собственные).

Диалоговые окна

Всплывающие диалоговые окна являются основой многих приложений, обеспечивая обратную связь и запрашивая ввод данных. Существует бесчисленное множество библиотек диалоговых окон, и вы можете создать собственную. Современные браузеры уже многое делают с помощью элемента `<dialog>`. Это всплывающее (popup) диалоговое окно, которое включает в себя фон, закрывающий остальную часть страницы. Вы можете применить стили как к фону, так и к диалоговому окну с помощью небольшого количества стилей CSS. По умолчанию диалоговое окно представляет собой просто всплывающее окно, за которым находится некоторый фон. Вам решать, добавлять ли заголовок, кнопки и другой контент.

Некоторые диалоговые окна содержат несколько кнопок, и вы хотите запустить разный код в зависимости от выбранного варианта нажатия. Например, в режиме подтверждения могут быть кнопки подтверждения и отмены. Вам также нужно будет справиться с этим самостоятельно, добавив к кнопкам прослушивателей событий нажатия. В каждом прослушивателе событий вы можете закрыть диалоговое окно, вызвав для него метод `close` — встроенное в диалоговое окно средство, которое принимает необязательный аргумент, позволяющий указать возвращаемое значение (return value). Его можно проверить позже с помощью свойства `returnValue` диалогового окна. Так можно передавать данные из диалогового окна обратно на страницу, с которой оно было открыто.

Элемент *details*

Элемент `<details>` — это компонент, содержимое которого можно сворачивать. У него есть некое краткое содержание (summary), отображаемое в интерактивном элементе. Щелкнув по этому элементу, вы можете отобразить или скрыть подробное содержимое. Как и в случае с диалоговыми окнами, вы можете стилизовать компонент с помощью CSS и переключать его видимость с помощью JavaScript.

Всплывающие окна

Всплывающее окно (popover) похоже на диалоговое окно. Это другой тип всплывающего элемента. Между всплывающим и диалоговым окнами есть несколько отличий:

- щелчок за пределами всплывающего окна приведет к его закрытию;
- вы по-прежнему можете взаимодействовать с остальной частью страницы, пока отображается всплывающее окно;
- вы можете превратить любой HTML-элемент во всплывающее окно.

Уведомления

Смартфоны широко используют уведомления, и более новые операционные системы также поддерживают уведомления. Современные браузеры имеют API для отображения собственных уведомлений операционной системы, запускаемых с помощью JavaScript. Пользователь должен предоставить разрешение на отправку этих уведомлений. Они создаются в вашем JavaScript-коде по запросу во время работы приложения.

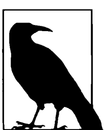
13.1. Создание диалогового окна предупреждения

Задача

Вы хотите отобразить диалоговое окно с простым сообщением и кнопкой **OK**, нажатие на которую закрывает окно.

Решение

Используйте элемент `<dialog>` с кнопкой **OK**.



Этот API может не поддерживаться старыми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/tk52g**).

Сначала определите HTML-код для вашего диалогового окна, как показано в примере 13.1.

Пример 13.1. Разметка диалогового окна

```
<dialog id="alert">
  <h2>Предупреждение</h2>
  <p>Это диалоговое окно предупреждения.</p>

  <button type="button" id="ok-button">OK</button>
</dialog>

<button type="button" id="show-dialog">Показать диалог</button>
```

Вам понадобятся два фрагмента JavaScript-кода. Во-первых, вам будет нужна функция для запуска отображения диалогового окна, а затем вам потребуется прослушиватель событий для кнопки **OK**, чтобы закрыть диалоговое окно (пример 13.2).

Пример 13.2. JavaScript-код для диалогового окна

```
// Задайте диалоговое окно, кнопку OK и элементы запуска окна.
const dialog = document.querySelector('#alert');
const okButton = document.querySelector('#ok-button');
const trigger = document.querySelector('#show-dialog');

// Закройте диалоговое окно нажатием кнопки OK.
okButton.addEventListener('click', () => {
  dialog.close();
});

// Отобразите диалоговое окно при нажатии кнопки запуска.
trigger.addEventListener('click', () => {
  dialog.showModal();
});
```

В результате откроется диалоговое окно, показанное на рис. 13.1.

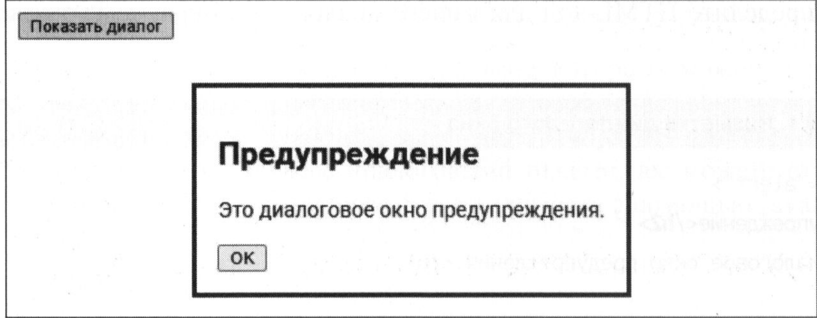


Рис. 13.1. Диалоговое окно предупреждения

Обсуждение

Метод showModal, используемый в диалоговом окне, показывает *модальный* диалог. Модальный диалог блокирует остальную часть страницы до тех пор, пока модальное окно не будет закрыто. Это означает, что если вы откроете модальный диалог, щелчок по другим элементам страницы не будет иметь эффекта. В модальном диалоговом окне фокус находится "в ловушке" внутри диалогового окна. Использование клавиши <Tab> приведет к циклическому переключению фокуса только на элементы, которые находятся в диалоговом окне. Если это не то, что вам нужно, вы также можете вызвать метод show. При этом откроется *немодальное* диалоговое окно, которое позволит вам взаимодействовать с остальной частью страницы, пока оно открыто.

Нажатие кнопки **OK** закроет диалоговое окно, поскольку прослушиватель нажатий вызовет функцию dialog.close, но вы также можете закрыть модальное окно, нажав клавишу <Esc>. Для того чтобы зафиксировать это, вы можете прослушать событие cancel диалога. Отмена диалогового окна с помощью клавиши <Esc> также вызовет событие close закрытия диалогового окна. Наконец, закрытие диалогового окна вручную с помощью команды close также вызовет событие закрытия.

Элемент <dialog> также имеет несколько приятных функций, доступных с клавиатуры. Когда вы нажимаете кнопку **Показать диалог** (Show Dialog) и диалоговое окно открывается, появляется кнопка первого фокусируемого элемента, которая автоматически получает фокусировку. В данном случае это кнопка **OK**. Вы можете изменить это поведение, добавив атрибут autofocus к элементу, на котором вы хотите получить фокусировку при открытии диалогового окна.

Когда вы закроете диалоговое окно, нажав клавишу <Esc> или кнопку **OK**, фокус клавиатуры вернется к кнопке **Показать диалог**.

Вы можете оформить как само диалоговое окно, так и его полупрозрачный фон с помощью CSS. Для диалогового окна вы можете добавить правило CSS, ориентированное на сам элемент <dialog>. Для того чтобы придать стилю фон — например, вы можете захотеть, чтобы он был полупрозрачного черного цвета — вы можете использовать псевдоэлемент ::background (пример 13.3).

Пример 13.3. Стилизация фона

```
#alert::backdrop {
  background: rgba(0, 0, 0, 0.75);
}
```

13.2. Создание диалогового окна подтверждения

Задача

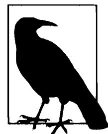
Вы хотите предложить пользователю подтвердить операцию. В приглашении должен отображаться вопрос и должны быть кнопки подтверждения и отмены.

Решение

Это еще один отличный вариант использования элемента `<dialog>`. Сначала создайте содержимое вашего диалога с подсказками и кнопками[1], как показано в примере 13.4.

Пример 13.4. Разметка диалога подтверждения

```
<dialog id="confirm">
  <h2>Подтверждение</h2>
  <p>Вы уверены, что хотите сделать это?</p>

  <button type="button" id="confirm-button">Подтвердить</button>
  <button type="button" id="cancel-button">Отмена</button>
</dialog>

<button type="button" id="show-dialog">Показать диалог</button>
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/tk52g**).

Вы хотите, чтобы обе кнопки закрывали диалоговое окно, но выполняли разные действия. Для этого вы можете передать в `dialog.close` строковый аргумент. Это

[1] В исходном коде на GitHub в html-файле ошибка. Теги `<button>` должны иметь атрибут `id`, как показано здесь, а не атрибуты `class`, как в GitHub. — *Прим. ред.*

установит свойство returnValue для самого диалогового окна, которое вы сможете проверить при получении события close (пример 13.3).

Пример 13.5. Прослушиватели событий для диалогового окна подтверждения

```
const dialog = document.querySelector('#confirm');
const confirmButton = document.querySelector('#confirm-button');
const cancelButton = document.querySelector('#cancel-button');
const trigger = document.querySelector('#show-dialog');

confirmButton.addEventListener('click', () => {
  // Закройте диалоговое окно с возвращаемым значением 'confirm'
  dialog.close('confirm');
});

cancelButton.addEventListener('click', () => {
  // Закройте диалоговое окно, вернув значение 'cancel'
  dialog.close('cancel');
});

dialog.addEventListener('cancel', () => {
  // Отмена с помощью клавиши <Esc> не приводит к возвращаемому значению.
  // Установите здесь значение 'cancel', чтобы обработчик
  // события закрытия получил правильное значение.
  dialog.returnValue = 'cancel';
});

dialog.addEventListener('close', () => {
  if (dialog.returnValue === 'confirm') {
    // Пользователь нажал кнопку подтверждения. Выполните действие,
    // например, создайте или удалите данные.
  } else {
    // Пользователь нажал кнопку отмены или клавишу <Esc>.
    // Не выполняйте действие.
  }
});

trigger.addEventListener('click', () => {
  dialog.showModal();
});
```

Появившееся в результате диалоговое окно подтверждения выглядит так, как показано на рис. 13.2.

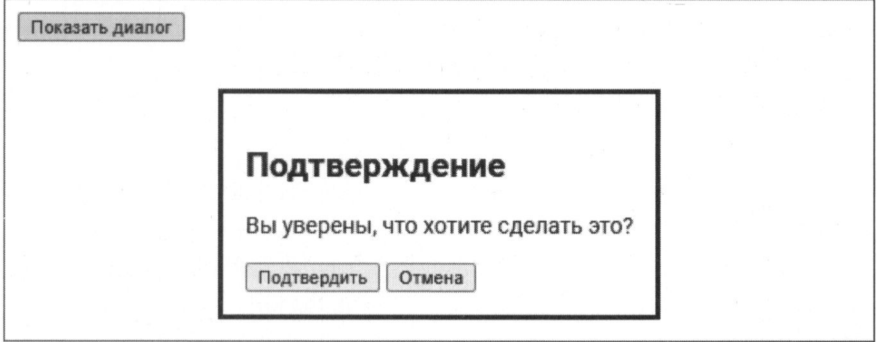


Рис. 13.2. Диалоговое окно подтверждения

Обсуждение

Если пользователь нажимает одну из кнопок, диалоговое окно закрывается с возвращаемым значением, которое зависит от того, какая кнопка была нажата. После закрытия диалогового окна будет выдано событие `close`, в котором вы можете проверить свойство `returnValue`. Если возвращаемое значение равно `confirm`, вы знаете, что пользователь нажал кнопку подтверждения. В противном случае возвращаемое значение равно `cancel`, и вы можете отменить операцию.

В этом примере также прослушивается событие `cancel`. Оно запускается, если диалоговое окно закрывается нажатием клавиши <Esc>. Когда диалоговое окно закрывается таким образом, `returnValue` не обновляется и сохраняет то предыдущее значение, которое у него было. Для того чтобы убедиться в правильности `returnValue`, обработчик события `cancel` устанавливает его. Это работает, потому что событие `close` запускается *после* события `cancel`. Поскольку клавиша <Esc> запускает это событие, вам не нужно на самом деле прослушивать нажатие клавиши <Esc>.

Зачем вам нужно обрабатывать этот случай? Ну, если вы закроете диалоговое окно, оно не будет удалено. Оно по-прежнему существует в DOM, просто скрыто, и по-прежнему имеет то же значение `returnValue`. Предположим, что ранее вы открывали диалоговое окно и нажимали кнопку **Подтвердить** (Confirm). Возвращаемое значение равно теперь `confirm` (подтверждение). Если вы снова откроете диалоговое окно подтверждения и отмените его нажатием кнопки <Esc>, при обработке события `close` возвращаемое значение по-прежнему будет `confirm`. Для того чтобы избежать этой потенциальной ошибки, вы можете использовать обработчик события `cancel`, чтобы явно задать `returnValue` значением `cancel`.

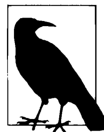
13.3. Создание веб-компонента диалогового окна подтверждения

Задача

Вы хотите создать настраиваемое диалоговое окно подтверждения. Когда вы показываете диалоговое окно, вместо того чтобы прослушивать несколько событий, вы хотите получить `Promise`, который соответствует возвращаемому значению.

Решение

Перенесите диалоговое окно в веб-компонент, используя слот для сообщения о подтверждении. Компонент предоставляет метод `showConfirmation`, который использует `Promise`.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/tk52g**).

Как и в случае с большинством веб-компонентов, начните с определения шаблона, как показано в примере 13.6.

Пример 13.6. Шаблон для компонента диалогового окна подтверждения

```
const template = document.createElement('template');
template.innerHTML = `
  <dialog id="confirm">
    <h2>Подтверждение</h2>
    <p><slot></slot></p>

    <button type="button" id="confirm-button">Подтвердить</button>
    <button type="button" id="cancel-button">Отмена</button>
  </dialog>
`;
```

Шаблон содержит слот, в который будет загружаться дочернее содержимое компонента. Далее в примере 13.7 показана реализация компонента.

Пример 13.7. Реализация компонента подтверждения

```
class ConfirmDialog extends HTMLElement {
  connectedCallback() {
```

```
    const shadowRoot = this.attachShadow({ mode: 'open' });
    shadowRoot.appendChild(template.content.cloneNode(true));

    this.dialog = shadowRoot.querySelector('dialog');
    this.dialog.addEventListener('cancel', () => {
      this.dialog.returnValue = 'cancel';
    });

    shadowRoot.querySelector('.confirm-button')
    .addEventListener('click', () => {
      this.dialog.close('confirm');
    });

    shadowRoot.querySelector('.cancel-button')
    .addEventListener('click', () => {
      this.dialog.close('cancel');
    });
  }

  showConfirmation() {
    this.dialog.showModal();
    return new Promise(resolve => {
      // Прослушайте следующее событие закрытия и разрешите Promise.
      // Разрешите Promise с помощью логического значения, указывающего,
      // нажал ли пользователь кнопку подтверждения.
      this.dialog.addEventListener('close', () => {
        resolve(this.dialog.returnValue === 'confirm');
      }, {
        // Прослушайте событие только один раз, затем удалите прослушиватель.
        once: true
      });
    });
  }
}

customElements.define('confirm-dialog', ConfirmDialog);
```

Предположим, вы хотите использовать этот компонент для подтверждения операции удаления. Вы можете добавить элемент на свою страницу с запросом подтверждения в качестве дочернего содержимого (пример 13.8).

Пример 13.8. Разметка компонента

```
<confirm-dialog id="confirm-delete">
  Вы уверены, что хотите удалить этот элемент?
</confirm-dialog>
```

Для того чтобы отобразить диалоговое окно, выберите элемент DOM и вызовите его метод `showConfirmation`. Дождитесь возвращения `Promise`, чтобы получить возвращаемое значение (пример 13.9).

Пример 13.9. Использование компонента диалогового окна подтверждения

```
const confirmDialog = document.querySelector('#confirm-delete');
if (await confirmDialog.showConfirmation()) {
  // Выполните операцию удаления.
}
```

Как и в случае с рецептом 12.6 из *главы 12*, вам нужно добавить немного стилей CSS, чтобы скрыть дочернее содержимое, пока оно не будет помещено в слоты, и таким образом предотвратить мерцание содержимого диалогового окна (пример 13.10).

Пример 13.10. Устранение проблемы с мерцанием

```
confirm-dialog:not(:defined) {
  display: none;
}
```

Обсуждение

Это хороший пример полезности веб-компонентов для инкапсуляции пользовательского поведения. В этом случае вы также добавили пользовательский метод, который будет вызываться извне. Этот метод отображает диалоговое окно и избавляет от необходимости прослушивать множество событий. Вы просто открываете диалоговое окно и ждете результата.

13.4. Использование элемента раскрытия информации

Задача

У вас есть какой-то контент, который вы хотите показать или скрыть с помощью переключения кнопки.

Решение

Используйте встроенный элемент <details> (пример 13.10).

Пример 13.11. Использование элемента details

```
<details>
  <summary>Дополнительные сведения</summary>
  Вот некоторые дополнительные детали, которые вы можете изменить.
</details>
```

Когда сведения details будут свернуты, вы просто увидите кнопку запуска дополнительной информации (рис. 13.3).

▶ Дополнительные сведения

Рис. 13.3. Свернутый элемент details

Когда вы нажимаете на **Дополнительные сведения** (More Info), открывается подробная информация, а стрелка меняется, указывая на то, что содержимое развернуто (рис. 13.4).

▼ Дополнительные сведения
Вот некоторые дополнительные детали, которые вы можете изменить.

Рис. 13.4. Элемент расширенных сведений

Обсуждение

По умолчанию внутреннее содержимое скрыто, и вы увидите только элемент раскрытия с содержимым элемента <summary>. В этом случае будет отображаться только надпись "Дополнительные сведения" (More Info). Когда вы нажмете кнопку-треугольник, появится скрытое содержимое. Если вы нажмете на нее еще раз, содержимое снова станет скрытым.

Вы можете изменить это поведение по умолчанию с помощью атрибута open. Если вы добавите этот атрибут, содержимое станет видимым (пример 13.12).

Пример 13.12. Управление состоянием по умолчанию с помощью атрибута open

```
<details open>
  <summary>Дополнительные сведения</summary>
  Это содержимое отображается по умолчанию.
</details>
```

Наконец, вы также можете переключать содержимое с помощью JavaScript. Вы можете напрямую изменить значение атрибута open элемента, как показано в примере 13.13.

Пример 13.13. Переключение видимости с помощью JavaScript

```
// Показать содержимое
document.querySelector('details').open = true;
```

Большинство браузеров имеют хорошую поддержку возможностей для этого элемента, идентифицируя элемент переключения для программ чтения с экрана и указывая его развернутое или свернутое состояние.

13.5. Отображение всплывающего окна

Задача

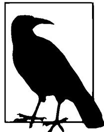
Вы хотите отобразить всплывающее содержимое при нажатии пользователем на кнопку, но при этом позволить пользователю взаимодействовать с остальной частью страницы.

Решение

Присвойте элементу атрибут popover и добавьте атрибут popovertarget к кнопке запуска (пример 13.14).

Пример 13.14. Автоматическое подключение всплывающего окна

```
<button type="button" popovertarget="greeting">Открыть всплывающее окно</button>
<div popover id="greeting">Привет, мир!</div>
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/YFjQX**).

Обсуждение

Всплывающее окно отличается от диалогового окна:

- вы можете открыть его без использования JavaScript;
- в нем нет фона, как в диалоговом окне;

- в отличие от диалогового окна, вам не запрещено взаимодействовать с основной страницей во время отображения всплывающего окна;
- когда вы щелкаете за пределами всплывающего окна, оно закрывается.

Для того чтобы превратить элемент во всплывающее окно, вы присваиваете ему атрибут `popover`. Для элемента `popover` также требуется атрибут `id`. Для того чтобы связать триггерную кнопку со всплывающим окном, кнопке присваивается атрибут `popovertarget`. Значение этого атрибута должно соответствовать `id` всплывающего окна.

Одним из недостатков API всплывающего окна в его текущем состоянии является отсутствие механизма позиционирования этого окна относительно его триггера. По умолчанию всплывающее окно всегда отображается по центру экрана. Если вы хотите изменить его положение, вам нужно будет сделать это вручную с помощью CSS.

В будущем вы сможете использовать привязку CSS для позиционирования всплывающего окна относительно его триггера. В настоящее время существуют сторонние библиотеки, такие как Floating UI, которые вы можете использовать для расширения этого решения, чтобы позиционировать всплывающее окно.

13.6. Ручное управление всплывающим окном

Задача

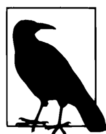
Вы хотите использовать атрибут `popover`, но желаете программно контролировать с помощью JavaScript отображение и скрытие этого всплывающего окна.

Решение

Установите для атрибута `popover` значение `manual` и вызовите соответствующие методы `showPopover`, `hidePopover` или `togglePopover` (пример 13.15).

Пример 13.15. Разметка всплывающего окна и его триггера

```
<button type="button" id="trigger">Показать всплывающее окно</button>
<div id="greeting" popover="manual">Привет, мир!</div>
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/YFjQX**).

Атрибут `popover="manual"` сообщает браузеру, что всплывающее окно будет управляться вручную (пример 13.16). Для того чтобы отобразить всплывающее окно, вы-

берите элемент popover и вызовите его метод togglePopover. Эта манипуляция отобразит всплывающее окно, когда оно скрыто, и скроет его после появления.

Пример 13.16. Код кнопки переключения

```
const trigger = document.querySelector('#trigger');
const popover = document.querySelector('#greeting');
trigger.addEventListener('click', () => {
  popover.togglePopover();
});
```

Обсуждение

Если вы хотите вручную управлять видимостью всплывающего окна, убедитесь, что для атрибута popover установлено значение manual. Если для элемента popover установлено значение manual, щелчок за пределами всплывающего окна не приведет к его закрытию. Для того чтобы закрыть такое окно, вам нужно вызвать либо его метод hidePopover, либо метод togglePopover.

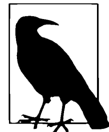
13.7. Позиционирование всплывающего окна относительно элемента

Задача

Вы желаете отобразить всплывающее окно, но не хотите, чтобы оно находилось в середине экрана. Вы хотите расположить его относительно другого элемента, например кнопки, которая его активировала.

Решение

Рассчитайте ограничивающий прямоугольник элемента, затем соответствующим образом отрегулируйте положение всплывающего окна. В этом примере мы рассмотрим расположение всплывающей подсказки непосредственно под элементом.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/YFjQX**).

Сначала вам нужно будет применить некоторые стили к элементу popover, как показано в примере 13.17.

Пример 13.17. Стили для всплывающего окна

```
.popover {
  margin: 0;
  margin-top: 1em;
  position: absolute;
}
```

По умолчанию браузер для центрирования всплывающего окна использует края окна браузера. Для того чтобы расположить всплывающее окно относительно другого элемента, вам необходимо удалить эти поля (margins). Поскольку вы размещаете всплывающую подсказку под другим элементом, вы можете установить `margintop` таким образом, чтобы между элементом и всплывающим окном оставалось небольшое пространство. Наконец, чтобы всплывающее окно прокручивалось вместе с элементом, вам нужно задать `position: fixed`.

Далее, вы можете использовать атрибут `popovertarget` в триггере, чтобы автоматически показывать всплывающее окно при нажатии[2] (пример 13.18).

Пример 13.18. Разметка всплывающего окна и его триггера

```
<button type="button" id="trigger" popovertarget="popover">Показать всплывающее
окно</button>
<div id="popover" popover>
  Это всплывающее содержимое, привязанное к кнопке запуска.
</div>
```

Последний шаг — обновлять положение всплывающего окна всякий раз, когда оно отображается. Вы можете прослушать событие `toggle` элемента `popover`, которое срабатывает, когда всплывающее окно отображается или скрывается. При обработке этого события вы можете вычислить положение триггера и использовать его для обновления положения всплывающего окна (пример 13.19).

Пример 13.19. Установка положения всплывающего окна

```
const popover = document.querySelector('.popover');
const trigger = document.querySelector('.trigger');

popover.addEventListener('toggle', event => {
  // Обновите позицию, если открывается всплывающее окно.
  if (event.newState === 'open') {
```

[2] В исходном коде на GitHub в html-файле ошибка. Теги `<button>` и `<div>` должны иметь атрибут `id`, как показано здесь, а не атрибуты `class`, как в GitHub. — *Прим. ред.*

```
    // Найдите положение триггерного элемента.
    const triggerRect = trigger.getBoundingClientRect();

    // Поскольку всплывающее окно расположено относительно окна
    // просмотра, вам необходимо учитывать смещение прокрутки.
    popover.style.top = `${triggerRect.bottom + window.scrollY}px`;
    popover.style.left = `${triggerRect.left}px`;
  }
});
```

Обсуждение

Если вы знакомы с позиционированием в CSS, вас может немного смутить поведение `position: absolute` в данном случае. Обычно `position: absolute` позиционирует элемент относительно его ближайшего предка. Однако в этом случае всплывающее окно всегда будет расположено относительно окна просмотра.

Это связано с тем, что всплывающие окна располагаются внутри *верхнего слоя* браузера. Это специальный слой, который находится поверх всех остальных слоев в документе. Независимо от того, где в DOM находится ваш элемент `popover`, содержимое `popover` размещается на верхнем уровне. Поскольку элемент находится в этом специальном верхнем слое, настройка `position: absolute` установит его относительно области просмотра.

Положение всплывающего окна вычисляется путем вызова функции `getBoundingClientRect` для элемента `trigger`. При прокрутке страницы верхнее и нижнее положения этого прямоугольника будут меняться. Для того чтобы убедиться, что всплывающее окно расположено правильно под триггером, вам также необходимо включить в расчет `window.scrollY`.

В этой реализации есть несколько ограничений, на которые следует обратить внимание. Во-первых, если элемент `trigger` находится в нижней части документа, под ним может не хватить места для отображения всплывающего окна. Возможно, вы захотите проверить это и, если места не хватает, расположить всплывающее окно *над* триггером.

Еще одна проблема, с которой вам, возможно, придется столкнуться, заключается в том, что при изменении размера окна браузера положение всплывающего окна может быть изменено неправильно. Вы могли бы использовать `ResizeObserver` или событие `resize` окна, чтобы справиться с этим случаем.

13.8. Отображение всплывающей подсказки

Задача

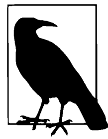
Вы хотите отображать всплывающую подсказку при наведении курсора мыши на элемент, т. е. при его фокусировке.

Решение

Используйте всплывающее окно с ручным управлением, отображая и скрывая его с помощью соответствующих событий мыши. При этом будет использоваться тот же подход к позиционированию, что и в рецепте 13.7, поэтому сначала вам нужно определить пользовательские стили для всплывающего окна (пример 13.20).

Пример 13.20. Стили всплывающих подсказок

```
#tooltip {
  margin: 0;
  margin-top: 1em;
  position: absolute;
}
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/YFjQX**).

Реализуйте всплывающую подсказку в виде всплывающего окна с атрибутом `popover`, установленным в значение `manual`, как показано в примере 13.21.

Пример 13.21. Разметка всплывающей подсказки

```
<button type="button" id="trigger">Наведи на меня курсор</button>
<div id="tooltip" popover="manual" role="tooltip">Вот некоторые всплывающие
подсказки</div>
```

При наведении курсора мыши на триггерный элемент вычислите положение и отобразите всплывающий элемент в событии `mouseover` курсора мыши. В событии `mouseout` курсора мыши скройте всплывающий элемент (пример 13.22).

Пример 13.22. Отображение и скрытие всплывающей подсказки

```
const button = document.querySelector('#trigger');
const tooltip = document.querySelector('#tooltip');

function showTooltip() {
  // Найдите положение триггерного элемента.
  const triggerRect = trigger.getBoundingClientRect();

  // Поскольку всплывающее окно расположено относительно окна
  // просмотра, вам необходимо учитывать смещение прокрутки.
  tooltip.style.top = `${triggerRect.bottom + window.scrollY}px`;
  tooltip.style.left = `${triggerRect.left}px`;

  tooltip.showPopover();
}

// Показывать и скрывать всплывающую подсказку в ответ на действия мыши.
button.addEventListener('mouseover', () => {
  showTooltip();
});

button.addEventListener('mouseout', () => {
  tooltip.hidePopover();
});

// Для удобства работы с клавиатурой также отображайте и скрывайте
// всплывающую подсказку в ответ на события фокусировки.
button.addEventListener('focus', () => {
  showTooltip();
});

button.addEventListener('blur', () => {
  tooltip.hidePopover();
});
```

Обсуждение

Поскольку здесь используется тот же метод позиционирования, что и в рецепте 13.7, он имеет те же ограничения:

- не учитывается случай, когда недостаточно места для отображения всплывающей подсказки под элементом;
- не учитывается изменение размера окна.

13.9. Отображение уведомления

Задача

Вы хотите уведомлять пользователя, когда что-то происходит в вашем приложении.

Решение

Используйте объект `Notification` для отображения собственного уведомления операционной системы.

Для того чтобы показывать уведомления, вы должны сначала запросить разрешение у пользователя. Это делается с помощью метода `Notification.requestPermission`. Для того чтобы проверить, предоставил ли пользователь разрешение, вы можете проверить свойство `Notification.permission`.

> **Уведомления в сравнении с push-уведомлениями**
>
> Уведомления, описанные в этом рецепте, запускаются только тогда, когда пользователь находится на странице. Это отличается от push-уведомлений, которые могут отправляться, даже если страница не активна. Это более сложный процесс, и обычно для него требуется использование стороннего сервиса.

В примере 13.23 показана вспомогательная функция, которая проверяет разрешение, при необходимости запрашивает разрешение у пользователя и возвращает логическое значение, указывающее, могут ли отображаться уведомления.

Пример 13.23. Проверка разрешений на отправку уведомлений

```
async function getPermission() {
  // Если пользователь уже явно отказал в разрешении, не спрашивайте об этом снова.
  if (Notification.permission !== 'denied') {
    // Результатом этого запроса разрешения будет обновление свойства
    // Notification.permission. Запрос разрешения возвращает Promise.
    await Notification.requestPermission();
  }

  // Показывайте уведомление, если Notification.permission "предоставлено".
  return Notification.permission === 'granted';
}
```

После проверки наличия разрешения вы можете отправить уведомление, создав новый экземпляр уведомления. Используйте вспомогательный инструмент `getPermission`, чтобы определить, должно ли отображаться уведомление (пример 13.24).

Пример 13.24. Отображение уведомления

```
if (await getPermission()) {
  new Notification('Hello!', {
    body: 'This is a test notification'
  });
}
```

Если вы попытаетесь отобразить уведомление, когда разрешение не было предоставлено, объект `Notification` вызовет событие `error`.

На рис. 13.5 показано, как это уведомление может выглядеть на настольном компьютере.

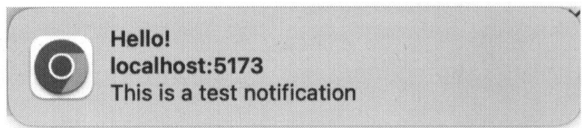


Рис. 13.5. Уведомление, отображаемое в macOS 14

Обсуждение

Уведомления могут отображаться только из приложений, запущенных в *защищенном контексте*. Как правило, это означает, что они должны отправляться по протоколу HTTPS или с URL-адреса `localhost`.

Свойство `Notification.permission` имеет одно из трех значений.

`granted`

Пользователь явно предоставил разрешения на показ уведомлений.

`denied`

Пользователь недвусмысленно отказал в разрешении показывать уведомления при появлении соответствующего запроса.

`default`

Пользователь не ответил на запрос разрешения на получение уведомления. Браузеры отнесутся к этому так же, как к отказу (denied).

Уведомление может вызвать и другие события.

`show`

Срабатывает, когда отображается уведомление.

`close`

Срабатывает, когда уведомление закрывается.

`click`

Срабатывает, когда на уведомлении происходит щелчок мыши.

ГЛАВА 14

Интеграция устройств

14.0. Введение

Современная платформа веб-браузера включает API для взаимодействия со всеми видами информации, а также данными о технических возможностях устройства, включая:

- состояние батареи;
- состояние сети;
- геолокацию;
- буфер обмена устройства;
- общий доступ к контенту;
- тактильную обратную связь.

На момент написания этой книги не все из перечисленных API имели полную поддержку. Некоторые из них все еще считаются экспериментальными, поэтому пока не стоит использовать их в рабочих приложениях.

Некоторые из этих API могут поддерживаться определенным браузером, например Chrome, но все равно не будут работать, если на самом устройстве отсутствуют необходимые функции. Например, Vibration API хорошо поддерживается Chrome, но не будет работать на ноутбуке или другом устройстве без поддержки вибрации.

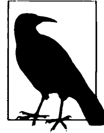
14.1. Считывание состояния батареи

Задача

Вы хотите в своем приложении отобразить состояние зарядки аккумулятора устройства.

Решение

Для определения состояния батареи используйте Battery Status API.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/DWFvk**).

Вы можете запросить состояние батареи, вызвав `navigator.getBattery`. Этот метод возвращает `Promise`, который преобразуется в объект, содержащий информацию о батарее.

Сначала напишите несколько элементов-заполнителей HTML для отображения состояния батареи, как показано в примере 14.1.

Пример 14.1. Разметка состояния батареи

```
<ul>
  <li>Уровень заряда батареи:<span id="battery-level">--</span></li>
  <li>Состояние заряда батареи:<span id="battery-charging">--</span></li>
</ul>
```

Затем вы можете запросить API-интерфейс состояния батареи, чтобы получить ее уровень заряда и статус зарядки, добавив их к соответствующим элементам DOM (пример 14.2).

Пример 14.2. Запрос API-интерфейса состояния батареи

```
const batteryLevelItem = document.querySelector('#battery-level');
const batteryChargingItem = document.querySelector('#battery-charging');

navigator.getBattery().then(battery => {
  // Уровень заряда батареи - это число от 0 до 1. Умножьте на 100,
  // чтобы преобразовать его в процентный формат.
  batteryLevelItem.textContent = `${battery.level * 100}%`;

  batteryChargingItem.textContent = battery.charging ? 'Заряжается' : 'Не заряжается';
});
```

Что, если вы отключите ноутбук от сети? Отображаемое состояние зарядки больше не будет соответствовать действительности. Для того чтобы справиться с этим, вы можете прослушать некоторые события.

`levelchange`

Срабатывает при изменении уровня заряда аккумулятора.

`chargingchange`

Срабатывает при запуске или прекращении зарядки аккумулятора.

Вы можете обновить пользовательский интерфейс при возникновении этих событий. Убедитесь, что у вас есть ссылка на объект battery, затем добавьте прослушивателей событий (пример 14.3).

Пример 14.3. Отслеживание событий, связанных с батареей

```
battery.addEventListener('levelchange', () => {
  batteryLevelItem.textContent = `${battery.level * 100}%`;
});

battery.addEventListener('chargingchange', () => {
  batteryChargingItem.textContent = battery.charging ? 'Заряжается' : 'Не заряжается';
});
```

Теперь состояние батареи постоянно обновляется. Если вы отключите ноутбук от сети, статус зарядки изменится с "Заряжается" ("Charging") на "Не заряжается" ("Not charging").

Обсуждение

На момент написания книги некоторые браузеры вообще не поддерживали этот API. Вы можете использовать код из примера 14.4, чтобы проверить, поддерживается ли Battery Status API в браузере пользователя.

Пример 14.4. Проверка поддержки API состояния батареи

```
if ('getBattery' in navigator) {
  // здесь запросите статус батареи
} else {
  // не поддерживается
}
```

В объекте battery также доступны некоторые дополнительные свойства.

`chargingTime`

Количество секунд, оставшееся до полной зарядки аккумулятора, если аккумулятор заряжается. Если аккумулятор не заряжается, это значение равно `Infinity` (бесконечности).

`dischargingTime`

Количество секунд, оставшихся до полной разрядки аккумулятора, если аккумулятор не заряжается. Если аккумулятор не разряжается, это значение равно `Infinity`.

У этих двух свойств также есть собственные события, называемые `chargingtimechange` и `dischargingtimechange` соответственно, которые вы можете прослушивать.

С информацией, предоставляемой Battery Status API для определения состояния батареи, можно выполнять множество действий. Например, если уровень заряда батареи низкий, вы можете отключить фоновые задачи или другие энергоемкие операции. Или это может быть даже что-то простое, например, уведомление пользователя о том, что он должен сохранить внесенные изменения, поскольку уровень заряда батареи устройства низкий.

Вы также можете использовать этот API для отображения простого индикатора состояния батареи. Если у вас есть ряд значков, представляющих различные состояния батареи (полностью заряжена, не заряжается, заряжается, низкая зарядка), вы можете поддерживать отображаемый значок в актуальном состоянии, отслеживая события изменения.

14.2. Считывание состояния сети

Задача

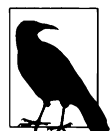
Вы хотите знать, какова скорость подключения пользователя к сети.

Решение

Используйте Network Information API для получения данных о сетевом подключении пользователя (пример 14.5).

Пример 14.5. Проверка возможностей сети

```
if (navigator.connection.effectiveType === '4g') {
  // Пользователь может выполнять действия с высокой скоростью.
}
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/krDAV**).

Обсуждение

Информация о сети содержится в объекте `navigator.connection`. Для того чтобы получить приблизительное представление о возможностях сетевого подключения, проверьте свойство `navigator.connection.effectiveType`. На момент написания книги

допустимые значения параметра `navigator.connection.effectiveType` в зависимости от скорости загрузки были следующими:

- `slow-2g` — до 50 Кбит/с;
- `2g` — до 70 Кбит/с;
- `3g` — до 700 Кбит/с;
- `4g` — 700 Кбит/с и выше.

Эти значения рассчитаны на основе реальных пользовательских данных. В спецификации указано, что эти значения могут быть обновлены в будущем. Вы можете использовать их для приблизительного определения сетевых возможностей устройства. Так, значение `slow-2g` для `effectiveType`, вероятно, не позволит выполнять операции с высокой пропускной способностью, такие как потоковая передача HD-видео.

Если сетевое подключение изменится, пока страница открыта, объект `navigator.connection` может инициировать событие `change`. Вы можете прослушать это событие и настроить ваше приложение на основе новой полученной информации о сетевом подключении.

14.3. Определение местоположения устройства

Задача

Вы хотите узнать местоположение устройства.

Решение

Для того чтобы получить местоположение по широте и долготе, используйте Geolocation API. API геолокации предоставляет объект `navigator.geolocation`, который используется для запроса местоположения пользователя с помощью метода `getCurrentPosition`. Это API, основанный на обратном вызове. `getCurrentPosition` принимает два аргумента. Первый аргумент — это успешный обратный вызов, а второй указывает на ошибку (пример 14.6).

Пример 14.6. Запрос местоположения устройства

```
navigator.geolocation.getCurrentPosition(position => {
  console.log('Latitude: ' + position.coords.latitude);
  console.log('Longitude: ' + position.coords.longitude);
}, error => {
  // Либо пользователь отказал в разрешении, либо местоположение
  // устройства не удалось определить.
  console.log(error);
});
```

Для этого API требуется разрешение пользователя. При первом вызове getCurrentPosition браузер запрашивает у пользователя разрешение на публикацию своего местоположения. Если пользователь не предоставляет разрешение, запрос геолокации не выполняется и браузер возвращает ошибку через функцию обратного вызова.

Если вы хотите проверить наличие разрешения заранее, чтобы избежать обнаружения ошибки, можете использовать Permissions API для проверки его статуса (пример 14.7).

Пример 14.7. Проверка разрешения на геолокацию

```
const permission = await navigator.permissions.query({
  name: 'geolocation'
});
```

Возвращаемый объект разрешения имеет свойство state, которое может иметь одно из значений: granted (предоставлено), denied (отказано) или prompt (приглашение). Получая состояние "отказано", вы знаете, что пользователю уже был отправлен запрос, и он отклонил его, поэтому вам не следует пытаться узнать его местоположение, поскольку это приведет к сбою.

Обсуждение

Браузер может попробовать определить местоположение пользователя несколькими способами. Он может попытаться использовать GPS-навигатор устройства, информацию о подключении пользователя к Wi-Fi или IP-адрес. В некоторых случаях, например, когда пользователь использует VPN, геолокация на основе IP может возвращать неправильное местоположение устройства пользователя.

Сегодня Geolocation API имеет очень хорошую поддержку браузеров, поэтому вам не нужно проверять наличие поддержки функций, если вы не ориентируетесь на старые браузеры.

В дополнение к координатам объект position содержит еще несколько интересных сведений, которые, однако, могут быть доступны не на всех устройствах.

altitude

Высота устройства над уровнем моря в метрах.

heading

Направление движения устройства по компасу в градусах.

speed

Скорость устройства, если оно движется, в метрах за секунду.

Вы также можете отслеживать изменения местоположения устройства, вызывая navigator.geolocation.watchCurrentPosition. Браузер периодически делает обратный

вызов, который вы передаете этому методу при изменении местоположения, предоставляя обновленные координаты.

Геолокация в сравнении с геокодированием

Geolocation API может получать только координаты устройства (широту и долготу). Он не может определить область (регион), город или конкретный адрес, по которому вы находитесь. Для этого вам понадобится API *геокодирования* (geocoding API), который не встроен в браузер. Существует множество внешних API геокодирования. API доступны от таких поставщиков, как Microsoft и Google. Геокодирование — это процесс получения адреса и преобразования его в широту и долготу. Некоторые из этих сервисов также могут выполнять *обратное геокодирование*, которое принимает координаты широты и долготы и преобразует их в адрес.

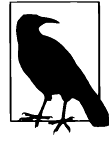
14.4. Отображение местоположения устройства на карте

Задача

Вы хотите отобразить карту местоположения устройства.

Решение

Используйте такие сервисы, как Google Maps API или OpenStreetMaps, для создания карты, передавая координаты широты и долготы из Geolocation API.



В этом рецепте вам необходимо зарегистрироваться для получения ключа Google Maps API. Инструкции по регистрации ключа API можно найти на веб-сайте разработчиков Google (**https://oreil.ly/9Uujk**).

В этом примере показано, как внедрить карту с помощью Google Maps Embed API. Вы можете использовать Google Maps Embed API, внедрив элемент `iframe` со специально созданным URL-адресом. URL-адрес должен содержать:

- тип карты (для этого примера вам нужна карта местности, place map);
- ваш API-ключ;
- координаты геолокации.

Запросите местоположение устройства, и в случае успешного завершения обратного вызова вы сможете создать `iframe` и встроить его в документ (пример 14.8).

Пример 14.8. Создание iframe-карты

```
// Предполагается, что у вас есть элемент-заполнитель с идентификатором 'map'
const map = document.querySelector('#map');

navigator.geolocation.getCurrentPosition(position => {
  const { latitude, longitude } = position.coords;

  // Отрегулируйте размер iframe по желанию.
  const iframe = document.createElement('iframe');
  iframe.width = 450;
  iframe.height = 250;

  // Тип карты является частью пути к URL-адресу.
  const url = new URL('https://www.google.com/maps/embed/v1/place');

  // Параметр 'key' содержит ваш API-ключ.
  url.searchParams.append('key', 'YOUR_GOOGLE_MAPS_API_KEY');

  // Параметр 'q' содержит координаты широты и долготы, разделенные запятой.
  url.searchParams.append('q', `${latitude},${longitude}`);
  iframe.src = url;

  map.appendChild(iframe);
});
```

Обсуждение

Ознакомьтесь со статьей от Google (**https://oreil.ly/WhO-r**), чтобы узнать больше о том, как правильно защитить API-ключ карты Google Maps.

Это лишь одна из многих возможных интеграций карт, которые вы можете использовать, как только получите информацию о местоположении устройства. В Google Maps есть другие типы API, а также другие сервисы, такие как Mapbox или OpenStreetMap. Вы также можете интегрировать API геокодирования для отображения маркера на карте с фактическим адресом.

14.5. Копирование и вставка текста

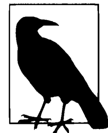
Задача

Вы хотите для текстовой области добавить функцию копирования и вставки.

Пользователь должен иметь возможность выделить какой-либо текст и скопировать его, а при вставке вставляемая копия должна заменить любой выделенный фрагмент.

Решение

Используйте Clipboard API для взаимодействия с выделенным текстом в текстовой области. Вы можете добавить кнопки копирования и вставки в свой пользовательский интерфейс, которые вызывают соответствующие функции Clipboard API.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/4i7sm**).

Для того чтобы скопировать текст, выделите подстроку из текстовой области для определения начальной и конечной точек копируемого фрагмента. Затем запишите этот текст в системный буфер обмена (пример 14.9).

Пример 14.9. Копирование текста выделенной области

```
async function copySelection(textarea) {
  const { selectionStart, selectionEnd } = textarea;
  const selectedText = textarea.value.slice(selectionStart, selectionEnd);

  try {
    await navigator.clipboard.writeText(selectedText);
  } catch (error) {
    console.error('Ошибка буфера обмена:', error);
  }
}
```

Процедура вставки аналогична, но в ней есть дополнительный шаг. Если в текстовой области выделена часть текста, вам нужно удалить выделенный текст и вставить новый текст из буфера обмена (пример 14.10). Clipboard API является асинхронным, поэтому вам нужно будет дождаться промиса, чтобы получить значение в системном буфере обмена.

Пример 14.10. Вставка текста в выделенную область

```
async function pasteToSelection(textarea) {
  const currentValue = textarea.value;
  const { selectionStart, selectionEnd } = textarea;
```

```
  try {
    const clipboardValue = await navigator.clipboard.readText();
    const newValue = currentValue.slice(0, selectionStart)
    + clipboardValue + currentValue.slice(selectionEnd);
    textarea.value = newValue;
  } catch (error) {
    console.error('Clipboard error:', error);
  }
}
```

При этом выделенный в данный момент фрагмент будет заменен текстом из буфера обмена.

Обсуждение

Обратите внимание, что даже если вы ничего не делаете с возвращаемым значением `navigator.clipboard.writeText`, вы все равно ожидаете выполнения `Promise`. Это связано с тем, что вам нужно обработать случай, когда `Promise` отклоняется с ошибкой.

Кроме того, при вставке следует учитывать два других сценария:

- если текст не выделен, но текстовая область имеет фокус, текст вставляется в соответствии с положением курсора;
- если текстовая область не имеет фокуса, текст вставляется в конец текстовой области.

Как вы могли ожидать, программное чтение из системного буфера обмена может быть вопросом конфиденциальности, поэтому требуется разрешение пользователя. При первой попытке чтения из буфера обмена браузер запрашивает разрешение у пользователя. Если пользователь разрешает, операция с буфером обмена завершается. Если он отказывает в разрешении, промис, возвращаемый Clipboard API, отклоняется с ошибкой.

Если вы хотите избежать ошибок с разрешениями, можете использовать Permissions API, чтобы проверить, предоставил ли пользователь разрешение на чтение из системного буфера обмена (пример 14.11).

Пример 14.11. Проверка разрешения на чтение из буфера обмена

```
const permission = await navigator.permissions.query({
  name: 'clipboard-read'
});

if (permission.state !== 'denied') {
  // Продолжите операцию чтения из буфера обмена.
}
```

Для параметра `permission.state` возможны следующие три значения.

`granted`

Пользователь уже явно предоставил разрешение.

`denied`

Пользователь уже явно отказал в разрешении.

`prompt`

Разрешение у пользователя еще не запрашивалось.

Если в `permission.state` указано значение `prompt`, браузер автоматически запросит у пользователя разрешение на операцию чтения из буфера обмена.

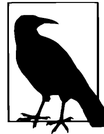
14.6. Совместное использование контента с помощью Web Share API

Задача

Вы хотите предоставить пользователю простой способ поделиться ссылкой, используя встроенные возможности общего доступа к своему устройству.

Решение

Используйте Web Share API для организации совместного использования контента.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/1IwEq**).

Вызовите `navigator.share` и передайте объект, содержащий заголовок и URL-адрес (пример 14.12). На поддерживаемых устройствах и в браузерах откроется знакомый интерфейс общего доступа, который позволяет обмениваться ссылкой различными способами.

Пример 14.12. Общий доступ к ссылке

```
if ('share' in navigator) {
  navigator.share({
    title: 'Web API Cookbook',
    text: 'Загляните на этот потрясающий сайт!',
    url: 'https://browserapis.dev'
  });
}
```

Отсюда пользователь может создать текстовое сообщение, электронное письмо или извещение другого типа, содержащее ссылку на контент.

Обсуждение

Интерфейс общего доступа выглядит по-разному в зависимости от устройства и операционной системы. Например, на рис. 14.1 показан скриншот интерфейса общего доступа на моем компьютере под управлением macOS 14.

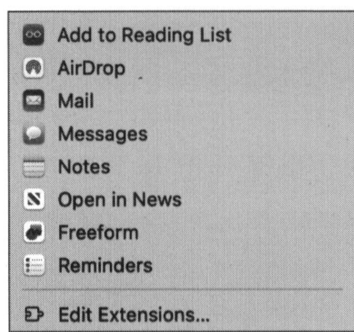


Рис. 14.1. Интерфейс общего доступа в macOS 14

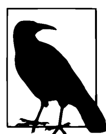
14.7. Создание вибрации устройства

Задача

Вы хотите добавить в свое приложение некоторую тактильную обратную связь, заставляющую устройство пользователя вибрировать.

Решение

Используйте Vibration API для программного запуска вибрации устройства.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/G0d6m**).

Для того чтобы выполнить однократную вибрацию, вы можете вызвать `navigator.vibrate` с единственным целочисленным аргументом (длительность вибрации), как показано в примере 14.13.

Пример 14.13. Запуск однократной вибрации

```
// Однократная вибрация в течение 500 миллисекунд.
navigator.vibrate(500);
```

Вы также можете запустить последовательность вибраций, передав массив в `navigator.vibrate` (пример 14.14). Элементы массива интерпретируются как последовательность вибраций и пауз.

Пример 14.14. Последовательность из трех вибраций

```
// Три вибрации по 500 миллисекунд с паузой в 250 миллисекунд между ними.
navigator.vibrate([500, 250, 500, 250, 500]);
```

Обсуждение

Этот API поддерживается даже на отдельных устройствах, которые не вибрируют, например Chrome на MacBook Pro. Для этих устройств вызов `navigator.vibrate` не имеет никакого эффекта, но и не выдает никакой ошибки.

Если выполняется последовательность вибраций, вы можете вызвать `navigator.vibrate(0)` и отменить все происходящие вибрации.

Как и при автоматическом воспроизведении видео, вы не можете активировать вибрацию автоматически при первой загрузке страницы. Пользователь должен каким-либо образом начать взаимодействовать со страницей, прежде чем можно будет использовать вибрацию.

14.8. Настройка ориентации устройства

Задача

Вы хотите определить, находится ли устройство в книжной или альбомной ориентации.

Решение

Используйте свойство `screen.orientation.type`, чтобы получить ориентацию устройства, или используйте свойство `screen.orientation.angle`, чтобы получить угол ориентации устройства относительно его естественного положения.

Обсуждение

Свойство `screen.orientation.type` может иметь одно из четырех значений в зависимости от устройства и его ориентации (рис. 14.2):

- `portrait-primary` — 0 градусов (естественное положение устройства);
- `portrait-secondary` — 180 градусов;
- `landscape-primary` — 90 градусов;
- `landscape-secondary` — 270 градусов.

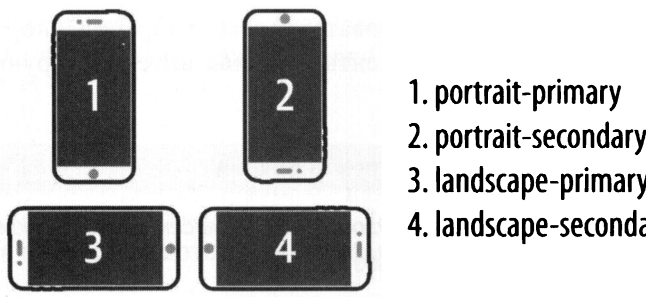


Рис. 14.2. Различные значения ориентации

Предыдущие значения приведены для устройств, таких как телефоны, для которых естественная ориентация — портретная. Для других устройств естественной будет альбомная ориентация, например, для некоторых планшетов значения меняются на противоположные:

- `landscape-primary` — 0 градусов (естественное положение устройства);
- `landscape-secondary` — 180 градусов;
- `portrait-primary` — 90 градусов;
- `portrait-secondary` — 270 градусов.

Объект `screen.orientation` также имеет событие изменения, которое вы можете прослушать, чтобы не пропустить изменения ориентации устройства.

ГЛАВА 15

Измерение производительности

15.0. Введение

Существует множество сторонних инструментов, предназначенных для измерения производительности в JavaScript-приложении. В браузере также есть несколько удобных инструментов, встроенных для сбора показателей производительности.

Navigation Timing API используется для сбора данных о производительности при начальной загрузке страницы. Вы можете проверить, сколько времени потребовалось для загрузки страницы, сколько времени нужно, чтобы DOM стал интерактивным, и многое другое. Он возвращает набор временны́х меток, которые указывают, когда во время загрузки страницы произошло каждое событие.

Resource Timing API позволяет проверить, сколько времени потребовалось для загрузки ресурсов и выполнения сетевых запросов. Это относится к таким ресурсам страницы, как HTML-файлы, CSS-файлы, файлы JavaScript и файлы изображений. Это также относится к асинхронным запросам, например, к запросам, выполняемым с помощью Fetch API.

User Timing API — это способ вычисления времени, затраченного на выполнение произвольных операций. Вы можете создавать *отметки* (marks) производительности, которые представляют собой моменты времени, и *показатели* (measures), которые представляют собой рассчитанные промежутки времени между отметками.

Все эти API создают записи производительности в буфере на странице. Это единая коллекция записей производительности всех типов. Вы можете проверить этот буфер в любое время, а также использовать `PerformanceObserver` для асинхронного прослушивания новых записей, касающихся производительности.

В записях о производительности используются высокоточные временны́е метки. Время измеряется в миллисекундах, но также может содержать и дробные части, которые в некоторых браузерах могут иметь микросекундную точность. В браузере эти временны́е метки сохраняются как объекты `DOMHighResTimeStamp`. Это числа, которые начинаются с нуля при старте загрузки страницы и отражают время, прошедшее с момента старта загрузки страницы до момента совершения данной записи.

В рецептах этой главы рассматриваются решения для сбора показателей (метрик) производительности. Что вы будете делать с этими показателями, решать вам. Вы можете использовать Fetch API или Beacon API для отправки показателей производительности в API для сбора и последующего анализа.

Вы можете использовать эти показатели производительности во время разработки для целей отладки или оставить их до сбора реальных метрик производительности от ваших пользователей. Их можно отправить в службу аналитики для агрегирования и анализа.

15.1. Измерение производительности загрузки страниц

Задача

Вы хотите собрать информацию о времени загрузки страницы.

Решение

Найдите запись о производительности с типом `navigation` и извлеките временны́е метки навигации из объекта записи о производительности (пример 15.1). Затем можете рассчитать интервал между этими временны́ми метками, чтобы определить время, затрачиваемое на различные события загрузки страницы.

Пример 15.1. Поиск записи о времени выполнения навигации

```
// Существует только одна запись о производительности навигации.
const [navigation] = window.performance.getEntriesByType('navigation');
```

Этот объект обладает множеством свойств. В табл. 15.1 приведены несколько примеров полезных вычислений, которые вы можете выполнить.

Таблица 15.1. Расчеты времени навигации

Метрика	Начало	Конец
Время до первого байта	`startTime`	`responseStart`
Время до начала взаимодействия с DOM	`startTime`	`domInteractive`
Общее время загрузки	`startTime`	`loadEventEnd`

Обсуждение

Свойство `startTime` для записи о производительности навигации всегда равно 0.

Этот параметр содержит не только информацию о времени. Он содержит такую информацию, как объем переданных данных, код HTTP-ответа и URL-адрес стра-

ницы. Эта информация полезна для определения того, насколько быстро ваше приложение становится отзывчивым при первой загрузке.

15.2. Измерение эффективности использования ресурсов

Задача

Вы хотите получать информацию о запросах к ресурсам, загруженным на страницу.

Решение

Найдите записи о производительности ресурсов в буфере производительности (пример 15.2).

Пример 15.2. Получение сведений о производительности ресурса

```
const entries = window.performance.getEntriesByType('resource');
```

Вы получите по одной записи для каждого ресурса на странице. К ресурсам относятся файлы CSS, файлы JavaScript-сценариев, изображения и любые другие запросы страницы.

Для каждого ресурса вы можете рассчитать, сколько времени потребовалось для загрузки, используя разницу между свойствами `startTime` и `responseEnd`. URL-адрес ресурса доступен в свойстве `name`.

Обсуждение

Любые сетевые запросы, которые вы отправляете с помощью Fetch API, также отображаются как ресурс. В этом польза API для профилирования реальной производительности ваших конечных точек REST API.

При первой загрузке страницы в буфер производительности заносятся данные обо всех ресурсах, запрошенных при начальной загрузке страницы. Последующие запросы добавляются в буфер производительности по мере их выполнения.

15.3. Поиск самых медленных ресурсов

Задача

Вы хотите получить список ресурсов, загрузка которых занимает больше всего времени.

Решение

Отсортируйте и отфильтруйте список записей о производительности ресурсов. Поскольку этот список представляет собой просто массив, вы можете вызывать для него такие методы, как `sort` и `slice`. Для того чтобы определить, сколько времени потребовалось ресурсу для загрузки, возьмите разницу между временны́ми метками `responseEnd` и `startTime`.

В примере 15.3 показано, как найти пять ресурсов, загружаемых с наименьшей скоростью.

Пример 15.3. Поиск пяти ресурсов, загружаемых с наименьшей скоростью

```
const slowestResources = window.performance.getEntriesByType('resource')
  .sort((a, b) =>
    (b.responseEnd - b.startTime) - (a.responseEnd - a.startTime))
  .slice(0, 5);
```

Обсуждение

Ключевым моментом является вызов метода `sort`. При этом сравнивается каждая пара значений времени загрузки и сортируется весь список в порядке убывания времени загрузки. Затем вызов `slice` просто берет первые пять элементов отсортированного массива. Если вместо этого вы хотите получить список из пяти ресурсов с самой быстрой загрузкой, вы можете просто изменить порядок, в котором сравнивается время загрузки (пример 15.4).

Пример 15.4. Поиск пяти самых быстрых ресурсов

```
const fastestResources = window.performance.getEntriesByType('resource')
  .sort((a, b) =>
    (a.responseEnd - a.startTime) - (b.responseEnd - b.startTime))
  .slice(0, 5);
```

Обратное сравнение означает, что массив сортируется по возрастанию, а не по убыванию. Вызов `slice` теперь возвращает пять ресурсов, загружаемых быстрее всего.

15.4. Определение тайминга для конкретного ресурса

Задача

Вы хотите посмотреть время выполнения запросов к определенному ресурсу.

Решение

Используйте метод `window.performance.getEntriesByName` для поиска ресурсов по определенному URL-адресу (пример 15.5).

Пример 15.5. Поиск всех таймингов ресурсов по определенному URL-адресу

```
// Поиск всех запросов к API /api/users
const entries = window.performance.getEntriesByName('https://localhost/api/users',
'resource');
```

Обсуждение

Имя записи ресурса — это его URL. Первым аргументом для `getEntriesByName` является URL. Второй аргумент указывает, что вас интересует тайминг работы ресурса.

Если было несколько запросов по данному URL-адресу, вы получите несколько записей о ресурсах в возвращаемом массиве.

15.5. Профилирование производительности рендеринга

Задача

Вы хотите записать время, необходимое для отображения некоторых данных на странице.

Решение

Создайте временну́ю метку производительности непосредственно перед началом рендеринга. Как только рендеринг завершится, создайте еще одну метку. Затем вы можете измерить промежуток между двумя метками, чтобы записать, сколько времени занял рендеринг.

Представьте, что у вас есть компонент `DataView`, который можно использовать для рендеринга некоторых данных на странице (пример 15.6).

Пример 15.6. Измерение производительности рендеринга

```
// Создайте начальную временную метку непосредственно перед рендерингом.
window.performance.mark('render-start');

// Создайте компонент и визуализируйте данные.
const dataView = new DataView();
dataView.render(data);
```

```
// Когда рендеринг будет завершен, создайте конечную метку производительности.
window.performance.mark('render-end');

// Подсчитайте время между двумя отметками.
const measure = window.performance.measure('render', 'render-start', 'render-end');
```

Объект measure содержит время начала и расчетную продолжительность измерения.

Обсуждение

Всякий раз, когда вы создаете метки и показатели производительности, они добавляются в буфер производительности страницы для последующего просмотра. Например, если вы хотите просмотреть показатель рендеринга позже, можете использовать window.performance.getEntriesByName (пример 15.7).

Пример 15.7. Поиск метрики по названию

```
// Существует только одна метрика 'render', поэтому вы можете использовать
// деструктурирование массива, чтобы получить первую (и единственную) запись.
const [renderMeasure] = window.performance.getEntriesByName('render');
```

Метки и метрики также могут содержать связанные с ними данные, если указать параметр detail. Например, при отображении данных в примере 15.6 вы можете передать сами данные в качестве метаданных при создании метрики.

При создании метрики таким образом вам необходимо включить начальную и конечную метки в объект options (пример 15.8).

Пример 15.8. Измерение производительности рендеринга с использованием данных

```
// Создайте начальную метку производительности непосредственно перед рендерингом.
window.performance.mark('render-start');

// Создайте компонент и визуализируйте данные.
const dataView = new DataView();
dataView.render(data);

// Когда рендеринг будет завершен, создайте конечную метку производительности.
window.performance.mark('render-end');

// Оцените время между двумя метками, передав отображаемые данные
// в качестве сведений о метрике.
const measure = window.performance.measure('render', {
```

```
  start: 'render-start',
  end: 'render-end',
  detail: data
});
```

Позже, когда вы будете искать эту запись о производительности, подробные метаданные будут доступны в свойстве `detail` этой метрики.

15.6. Профилирование многоэтапных задач

Задача

Вы хотите собрать данные о производительности для многоступенчатого процесса. Вы хотите получить тайминг всей последовательности процесса, а также время для отдельных шагов. Например, вы можете захотеть загрузить некоторые данные из API, а затем выполнить некоторую обработку этих данных. В этом случае вам нужно знать время выполнения запроса API, время обработки данных, а также общее время, затраченное на весь процесс.

Решение

Создайте несколько меток и метрик. Вы можете использовать одну и ту же метку для расчета сразу нескольких метрик.

В примере 15.9 есть API, который возвращает некоторые пользовательские транзакции. Как только транзакции будут получены, вы, возможно, захотите проанализировать данные о транзакциях. Затем аналитические данные отправляются в другой API.

Пример 15.9. Профилирование многоступенчатого процесса

```
window.performance.mark('transactions-start');
const transactions = await fetch('/api/users/123/transactions');
window.performance.mark('transactions-end');
window.performance.mark('process-start');
const analytics = processAnalytics(transactions);
window.performance.mark('process-end');
window.performance.mark('upload-start');
await fetch('/api/analytics', {
  method: 'POST',
  body: JSON.stringify(analytics),
  headers: {
```

```
    'Content-Type': 'application/json'
  }
});
window.performance.mark('upload-end');
```

После завершения процесса и определения меток вы можете использовать эти метки для создания нескольких метрик, как показано в примере 15.10.

Пример 15.10. Создание метрик

```
console.log('Загрузка транзакций:',
  window.performance.measure(
    'transactions', 'transactions-start', 'transactions-end'
  ).duration
);

console.log('Аналитика процессов:',
  window.performance.measure(
    'analytics', 'process-start', 'process-end'
  ).duration
);

console.log('Загрузка аналитики:',
  window.performance.measure(
    'upload', 'upload-start', 'upload-end'
  ).duration
);

console.log('Общее время:',
  window.performance.measure(
    'total', 'transactions-start', 'upload-end'
  ).duration
);
```

Обсуждение

В этом примере показано, как можно создать несколько меток и метрик для сбора данных о выполнении ряда задач. Одна и та же метка может использоваться более одного раза в нескольких метриках. В примере 15.10 создается метрика для каждого шага процесса, а затем генерируется итоговая метрика для всей задачи. Для этого берется первая метка задачи загрузки и последняя метка задачи выгрузки и вычисляется разница между ними.

15.7. Прослушивание показателей производительности

Задача

Вы хотите получать уведомления о новых показателях производительности, чтобы можно было сообщать о них в службу аналитики. Например, рассмотрим сценарий, в котором вы хотите получать уведомления о статистике производительности при каждом выполнении запроса API.

Решение

Используйте `PerformanceObserver` для поиска новых записей о производительности нужного типа. Для запросов API типом должен быть `resource` (пример 15.11).

Пример 15.11. Использование PerformanceObserver

```
const analyticsEndpoint = 'https://example.com/api/analytics';

const observer = new PerformanceObserver(entries => {
  for (let entry of entries.getEntries()) {
    // Интересуют только записи 'fetch'. Используйте Beacon API для отправки
    // быстрого запроса, содержащего данные о производительности.
    if (entry.initiatorType === 'fetch') {
      navigator.sendBeacon(analyticsEndpoint, entry);
    }
  }
});

observer.observe({ type: 'resource' });
```

Обсуждение

`PerformanceObserver` запускается при каждом сетевом запросе, включая тот, который вы отправляете в свою службу аналитики. По этой причине в примере 15.11 перед отправкой запроса проверяется, не является ли данная запись конечной точкой аналитики. Без этой проверки вы попадете в бесконечный цикл POST-запросов. Когда выполняется сетевой запрос, запускается `PerformanceObserver`, и вы отправляете POST-запрос. При этом создается новая запись производительности, которая снова вызывает `PerformanceObserver`. Каждый POST-запрос в аналитическом сервисе запускает новый обратный вызов.

Для того чтобы предотвратить лавину запросов к вашей аналитической службе за короткий промежуток времени, для реального приложения, возможно, потребуется сохранить данные о производительности в буфере. Как только буфер достигнет оп-

ределенного размера, вы сможете отправить все записи из буфера одним пакетом (пример 15.12).

Пример 15.12. Отправка записей о производительности пакетами

```
const analyticsEndpoint = 'https://example.com/api/analytics';
// Массив для хранения буферизованных записей. Как только буфер
// достигает желаемого размера, все записи отправляются в одном запросе.
const BUFFER_SIZE = 10;
let buffer = [];

const observer = new PerformanceObserver(entries => {
  for (let entry of entries.getEntries()) {
    if (entry.initiatorType === 'fetch' && entry.name !== analyticsEndpoint) {
      buffer.push(entry);
    }

    // Если буфер достиг своего целевого размера, отправьте запрос на аналитику.
    if (buffer.length === BUFFER_SIZE) {
      fetch(analyticsEndpoint, {
        method: 'POST',
        body: JSON.stringify(buffer),
        headers: {
          'Content-Type': 'application/json'
        }
      });

      // Сбросьте буфер после отправки пакетных записей.
      buffer = [];
    }
  }
});

observer.observe({ type: 'resource' });
```

ГЛАВА 16

Работа с консолью

16.0. Введение

Несмотря на все ваши благие намерения, при выполнении кода могут происходить сбои. В вашем распоряжении есть несколько инструментов для отладки. Современные браузеры оснащены мощными встроенными отладчиками, которые позволяют пошагово просматривать код и проверять значения переменных и выражений. Однако иногда возникает желание упростить задачу и использовать консоль.

В самом простом варианте вы взаимодействуете с консолью, вызывая `console.log` с сообщением. Это сообщение выводится на JavaScript-консоль браузера. Хотя существует более подробная версия отладки на основе точек останова, иногда все же может быть полезно регистрировать и проверять значения во время выполнения.

Помимо простого `console.log`, есть и другие операции, которые вы можете делать с помощью консоли, такие как групповые сообщения, использование счетчиков, отображение таблиц и даже оформление выходных данных с помощью CSS. Существуют также другие уровни ведения журнала (ошибка, предупреждение, отладка), к которым вы можете прибегнуть для классификации и фильтрации сообщений консоли.

16.1. Стилизации вывода консоли

Задача

Вы хотите применить некоторый CSS к выводам журнала вашей консоли. Например, вы хотите увеличить размер шрифта и изменить цвет.

Решение

Используйте *директиву* `%c` в сообщении своего журнала, чтобы указать, какой текст вы хотите стилизовать. Для каждого использования `%c` добавляйте другой аргумент в `console.log`, содержащий стили CSS (пример 16.1).

Пример 16.1. Стилизация вывода консоли

```
console.log('%cПривет, мир!', 'font-size: 2rem; color: red;');
console.log('Это консольное сообщение использует %cкурсивный текст. %cКруто!',
  'font-style: italic;',
  'font-weight: bold;'
);
```

На рис. 16.1 показано, как этот стилизованный текст выглядит в консоли.

Привет, мир!

Это консольное сообщение использует *курсивный текст.* **Круто!**

Рис. 16.1. Стилизованный консольный вывод

Обсуждение

`console.log` принимает переменное количество аргументов. Для каждого использования директивы `%c` должен быть соответствующий дополнительный аргумент, содержащий стили, которые необходимо применить к данному разделу текста.

Обратите внимание, что на рис. 16.1 стили сбрасываются между разделами `%c`. Курсивный шрифт из первого раздела (*курсивный текст*) не переходит в жирный шрифт второго раздела (**Круто!**).

16.2. Использование уровней в журналах сообщений

Задача

Вы хотите различать информационные сообщения, предупреждения и сообщения об ошибках, которые выводятся в консоли.

Решение

Вместо `console.log` используйте `console.info`, `console.warn` и `console.error`, соответственно (пример 16.2). Эти сообщения оформлены по-разному, и большинство браузеров позволяют вам фильтровать сообщения журнала по их уровню.

Пример 16.2. Использование различных уровней в журналах сообщений

```
console.info('Это информационное сообщение');
console.warn('Это предупреждение');
console.error('Это ошибка');
```

Сообщения оформлены по-разному, с помощью значков, как показано на рис. 16.2.

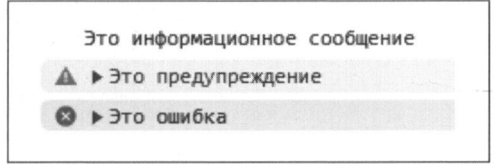


Рис. 16.2. Различные уровни журнальных сообщений (показаны в Chrome)

Обсуждение

Предупреждения и сообщения об ошибках также отображают трассировку стека, которую можно развернуть и просмотреть в консоли. Это позволяет легко отследить место, где произошла ошибка.

16.3. Создание именованных регистраторов

Задача

Вы хотите регистрировать сообщения из разных модулей вашего приложения с названием модуля в качестве префикса, выделенные заданным цветом.

Решение

Используйте функцию `Function.prototype.bind` консоли `console.log`, которая привязывает префикс к имени модуля и меняет цветовой стиль (пример 16.3).

Пример 16.3. Создание именованного регистратора (логгера)

```
function createLogger(name, color) {
  return console.log.bind(console, `%c${name}`, `color: ${color};`);
}
```

Функция `createLogger` возвращает новую функцию ведения журнала, которую вы можете вызвать так же, как `console.log`, но сообщения будут иметь цветной префикс (пример 16.4).

Пример 16.4. Использование именованных регистраторов (логгеров)

```
const rendererLogger = createLogger('renderer', 'blue');
const dataLogger = createLogger('data', 'green');

// Выводит с синим префиксом "renderer".
rendererLogger('Визуализированный компонент');
```

```
// Выводит с зеленым префиксом "data".
dataLogger('Извлеченные данные');
```

При этом сообщения журнала отображаются с цветными префиксами, как показано на рис. 16.3.

```
renderer Визуализированный компонент
data Извлеченные данные
```

Рис. 16.3. Цветные логгеры (показаны в Chrome)

Обсуждение

Вызов `bind` таким образом создает *частично примененную* версию `console.log`, которая автоматически добавляет префикс и цвет. Любые дополнительные аргументы, которые вы передаете ей, добавляются после префикса и цветового стиля.

16.4. Отображение массива объектов в таблице

Задача

У вас есть массив объектов, которые вы хотите вывести в легко читаемом виде.

Решение

Передайте массив в `console.table`, и он отобразит таблицу. Для каждого свойства объекта есть столбец, а для каждого объекта в массиве — строка (пример 16.5).

Пример 16.5. Запись таблицы в журнал

```
const users = [
  { firstName: "Джон", lastName: "Смит", department: "Продажи" },
  { firstName: "Эмили", lastName: "Джонсон", department: "Маркетинг" },
  { firstName: "Майкл", lastName: "Дэвис", department: "Отдел кадров" },
  { firstName: "Сара", lastName: "Томпсон", department: "Финансы" },
  { firstName: "Дэвид", lastName: "Уилсон", department: "Разработки" }
];
console.table(users);
```

На рис. 16.4 показано, как данные регистрируются в табличной форме.

(index)	firstName	lastName	department
0	'Джон'	'Смит'	'Продажи'
1	'Эмили'	'Джонсон'	'Маркетинг'
2	'Майкл'	'Дэвис'	'Отдел кадров'
3	'Сара'	'Томпсон'	'Финансы'
4	'Дэвид'	'Уилсон'	'Разработки'

Рис. 16.4. Таблица объектов (показана в Chrome)

Обсуждение

Вы можете ограничить количество отображаемых свойств объекта, передав второй аргумент в `console.table`. Этот аргумент представляет собой массив имен свойств. Если он задан, в выходных данных таблицы отображаются только эти свойства.

`console.table` также можно использовать с объектом. В примере 16.6 столбец `index` содержит имена свойств, а не индексы массива.

Пример 16.6. Передача объекта в console.table

```
console.table({
  name: 'sysadmin',
  email: 'admin@example.com'
});
```

В примере 16.6 приведена таблица, показанная на рис. 16.5.

(index)	Value
name	'sysadmin'
email	'admin@example.com'

Рис. 16.5. Таблица `console.table` (показана в Chrome)

В примере 16.7 пользователи регистрируются в таблице, но отображаются только столбцы с именем и фамилией (рис. 16.6).

Пример 16.7. Ограничение столбцов таблицы

```
const users = [
  { firstName: "Джон", lastName: "Смит", department: "Продажи" },
  { firstName: "Эмили", lastName: "Джонсон", department: "Маркетинг" },
  { firstName: "Майкл", lastName: "Дэвис", department: "Отдел кадров" },
```

```
  { firstName: "Сара", lastName: "Томпсон", department: "Финансы" },
  { firstName: "Дэвид", lastName: "Уилсон", department: "Разработки" }
];

console.table(users, ['firstName', 'lastName']);
```

(index)	firstName	lastName
0	'Джон'	'Смит'
1	'Эмили'	'Джонсон'
2	'Майкл'	'Дэвис'
3	'Сара'	'Томпсон'
4	'Дэвид'	'Уилсон'

Рис. 16.6. Выведены только столбцы с именем и фамилией (показаны в Chrome)

Отображенную таблицу также можно сортировать. Вы можете щелкнуть по названию столбца, чтобы отсортировать таблицу по этому столбцу (рис. 16.7).

(index)	firstName	lastName ▲
1	'Эмили'	'Джонсон'
2	'Майкл'	'Дэвис'
0	'Джон'	'Смит'
3	'Сара'	'Томпсон'
4	'Дэвид'	'Уилсон'

Рис. 16.7. Сортировка таблицы по фамилии (показано в Chrome)

16.5. Использование консольных таймеров

Задача

Вы хотите для целей отладки рассчитать время, затраченное на выполнение некоторого кода.

Решение

Используйте методы `console.time` и `console.timeEnd` (пример 16.8).

Пример 16.8. Использование console.time и console.timeEnd

```
// Запустите таймер 'loadTransactions'.
console.time('loadTransactions');
```

```
// Загрузите какие-нибудь данные.
const data = await fetch('/api/users/123/transactions');

// Остановите таймер 'loadTransactions'.
// Выводит: "loadTransactions: <elapsed time> ms"
console.timeEnd('loadTransactions');
```

Когда вы вызываете console.time с именем таймера, он запускает указанный таймер. Выполните любую работу, которую вы хотите профилировать, и когда закончите, вызовите console.timeEnd с тем же именем таймера. Прошедшее время вместе с названием таймера выводится на консоль.

Если вы вызываете console.timeEnd с именем таймера, которое не соответствует предыдущему вызову console.time, ошибка не выдается, но на консоль выводится предупреждающее сообщение о том, что таймер не существует.

Обсуждение

Представленное решение отличается от использования window.performance.mark и window.performance.measure, описанных в *главе 15*. console.time применяется для определения времени, обычно во время отладки. Заметное различие заключается в том, что console.time и console.timeEnd не добавляют записи на временну́ю шкалу производительности. Как только вы вызываете console.timeEnd для данного таймера, этот таймер уничтожается. Если вы хотите, чтобы данные о тайминге сохранялись в памяти, вы можете вместо этого использовать Performance API.

16.6. Использование консольных групп

Задача

Вы хотите улучшить организацию групп сообщений журнала.

Решение

Используйте console.group для создания вложенных групп сообщений, которые можно разворачивать и сворачивать (пример 16.9).

Пример 16.9. Использование консольных групп

```
const users = [
  { id: 1, firstName: "Джон", lastName: "Смит", department: "Продажи" },
  { id: 2, firstName: "Эмили", lastName: "Джонсон", department: "Маркетинг" },
  { id: 3, firstName: "Майкл", lastName: "Дэвис", department: "Отдел кадров" },
```

```
  { id: 4, firstName: "Сара", lastName: "Томпсон", department: "Финансы" },
  { id: 5, firstName: "Дэвид", lastName: "Уилсон", department: "Разработки" }
];

console.log('Обновление пользовательских данных');
for (const user of users) {
  console.group(`Пользователь: ${user.firstName} ${user.lastName}`);
  console.log('Загрузка данных о сотрудниках из API');
  const response = await fetch(`/api/users/${user.id}`);
  const userData = await response.json();

  console.log('Обновление профайла');
  userData.lastUpdated = Date.now();

  console.log('Сохранение пользовательских данных');
  await fetch(`/api/users/${user.id}`, {
    method: 'POST',
    body: JSON.stringify(userData),
    headers: {
      'Content-Type': 'application/json'
    }
  });
  console.groupEnd();
}
```

При этом сгруппированные сообщения будут выводиться на консоль. Вы можете разворачивать и сворачивать группы, чтобы сосредоточиться на конкретной группе, которая вас интересует (рис. 16.8).

Рис. 16.8. Сгруппированные сообщения консоли (показаны в Chrome)

Обсуждение

Вы также можете использовать консольные группы для отслеживания сложных алгоритмов. Группы могут быть вложены друг в друга, что значительно облегчает отслеживание сообщений в вашем журнале во время сложных вычислений. Это особенно ценно, когда нужно разобраться с большим количеством сообщений. Если вы хотите, чтобы группа была свернута по умолчанию, вы можете вызвать `console.groupCollapsed` вместо `console.group`.

16.7. Использование счетчиков

Задача

Вы хотите подсчитать, сколько раз вызывается та или иная часть вашего кода.

Решение

Вызовите метод `console.count` с именем счетчика, которое является уникальным для вашего кода. Каждый раз, когда выполняется инструкция `console.count`, она выводит значение счетчика и увеличивает его значение. Это позволяет отслеживать, сколько раз выполнялся вызов `console.count`.

В примере 16.10 выводятся выходные данные, которые представлены в примере 16.11.

Пример 16.10. Использование счетчиков

```
const users = [
  { id: 1, firstName: "Джон", lastName: "Смит", department: "Продажи" },
  { id: 2, firstName: "Эмили", lastName: "Джонсон", department: "Маркетинг" },
  { id: 3, firstName: "Майкл", lastName: "Дэвис", department: "Отдел кадров" },
  { id: 4, firstName: "Сара", lastName: "Томпсон", department: "Финансы" },
  { id: 5, firstName: "Дэвид", lastName: "Уилсон", department: "Разработки" }
];

users.forEach(user => {
  console.count('пользователь');
});
```

Пример 16.11. Выходные данные счетчика

```
пользователь: 1
пользователь: 2
```

```
пользователь: 3
пользователь: 4
пользователь: 5
```

Обсуждение

Метод `console.count` полезен для отслеживания итераций цикла или рекурсивных вызовов функций. Как и другие консольные методы, он в первую очередь предназначен для отладки и не предназначен для сбора метрик использования.

Вы также можете вызвать `console.count` без каких-либо аргументов, и в этом случае он использует счетчик с именем `default`.

16.8. Регистрация переменной и ее значения

Задача

Вы хотите записать имя переменной и ее значение в журнал без необходимости вводить имя дважды.

Решение

Используйте сокращенную запись объекта для регистрации объекта, содержащего переменную (пример 16.12).

Пример 16.12. Запись в журнал переменной и ее значения

```
const username = 'sysadmin';

// Записывает { username: 'sysadmin' }
console.log({ username });
```

При этом создается объект, имя которого — `username`, а значение — значение переменной `username`. Объект регистрируется в консоли, как показано на рис. 16.9.

```
▶ {username: 'sysadmin'}
```

Рис. 16.9. Объект с именованным значением (показан в Chrome)

Обсуждение

До создания сокращенного обозначения объекта вам нужно будет дважды ввести имя переменной (пример 16.13).

Пример 16.13. Запись переменной и ее значения в журнал без сокращения

```
const username = 'sysadmin';

console.log('username', username);
```

Это небольшое изменение, но оно позволяет сократить время.

16.9. Протоколирование трассировки стека

Задача

Вы хотите увидеть трассировку стека, показывающую место выполнения кода в данный момент.

Решение

Используйте `console.trace` для регистрации трассировки текущего стека вызовов (пример 16.14).

Пример 16.14. Использование console.trace

```
function foo() {
  function bar() {
    console.trace();
  }
  bar();
}

foo();
```

Код выводит трассировку стека, показанную на рис. 16.10.

```
▼ console.trace
  bar         @ VM3574:8
  foo         @ VM3574:10
  (anonymous) @ VM3574:13
```

Рис. 16.10. Журнал трассировки стека (показано в Chrome)

Обсуждение

Трассировка стека является полезным инструментом отладки. Она показывает текущее состояние стека вызовов. Первая запись в трассировке стека — это сам вызов `console.trace`. Затем следующая запись — это любая функция, содержащая вызов `console.trace`, и т. д. В большинстве браузеров вы можете щелкнуть по элементу трассировки стека, чтобы перейти к соответствующей строке кода. Вы можете использовать этот трюк для добавления записей в журнал или установки точек останова.

16.10. Проверка ожидаемых значений

Задача

Во время отладки вы желаете убедиться, что выражение имеет ожидаемое значение. Если это не так, вы хотите увидеть сообщение об ошибке в консоли.

Решение

Используйте `console.assert`, чтобы вывести сообщение об ошибке, если выражение не соответствует ожидаемому (пример 16.15).

Пример 16.15. Использование console.assert

```
function updateUser(user) {
  // Зарегистрируйте ошибку, если идентификатор пользователя равен null.
  console.assert(user.id !== null, 'user.id must not be null');

  // Обновите пользователя.
  return fetch(`/api/users/${user.id}`, {
    method: 'PUT',
    body: JSON.stringify(user),
    headers: {
      'Content-Type': 'application/json'
    }
  });
}
```

Если `updateUser` вызывается с использованием объекта `user` без свойства `id`, ошибка регистрируется в журнале.

Обсуждение

Утверждения (assertions) обычно не используются в рабочей среде, поскольку это инструмент отладки, подобный другим консольным методам. Важно отметить, что если утверждение завершается ошибкой, оно выводит сообщение об ошибке, но не выдает ошибку и не останавливает выполнение остальной части функции. В примере 16.15, если проверка идентификатора пользователя завершается неудачей, все равно производится попытка выполнить запрос PUT для обновления пользователя. Это, вероятно, приводит к ошибке 404, поскольку в URL-адресе указан `null`.

16.11. Изучение свойств объекта

Задача

Вы хотите проверить свойства объекта, включая глубоко вложенные свойства и цепочку прототипов.

Решение

Используйте `console.dir` для регистрации объекта.

```
▾ console i
  ▸ assert: ƒ assert()
  ▸ clear: ƒ clear()
  ▸ context: ƒ context()
  ▸ count: ƒ count()
  ▸ countReset: ƒ countReset()
  ▸ createTask: ƒ createTask()
  ▸ debug: ƒ debug()
  ▸ dir: ƒ dir()
  ▸ dirxml: ƒ dirxml()
  ▸ error: ƒ error()
  ▸ group: ƒ group()
  ▸ groupCollapsed: ƒ groupCollapsed()
  ▸ groupEnd: ƒ groupEnd()
  ▸ info: ƒ info()
  ▸ log: ƒ log()
  ▸ memory: MemoryInfo {totalJSHeapSize: 10000000, usedJSHeapSize: 10000000, jsHeapSizeLimit: 3760000000
  ▸ profile: ƒ profile()
  ▸ profileEnd: ƒ profileEnd()
  ▸ table: ƒ table()
  ▸ time: ƒ time()
  ▸ timeEnd: ƒ timeEnd()
  ▸ timeLog: ƒ timeLog()
  ▸ timeStamp: ƒ timeStamp()
  ▸ trace: ƒ trace()
  ▸ warn: ƒ warn()
    Symbol(Symbol.toStringTag): "console"
  ▸ [[Prototype]]: Object
```

Рис. 16.11. Использование `console.dir` в объекте `console` (показано в Chrome)

В примере 16.16 показано, как использовать `console.dir` для проверки самого объекта `console`.

Пример 16.16. Использование console.dir

```
console.dir(console);
```

На рис. 16.11 показана расширяемая древовидная структура, которая регистрируется в консоли. Каждая функция и свойство объекта могут быть расширены. Эта структура включает цепочку прототипов, которую также можно расширять и проверять.

Обсуждение

В некоторых версиях браузеров `console.log` также предоставляет интерактивную структуру для проверки объекта. Хотя это зависит от браузера, `console.dir` всегда проверяет объект, как показано на рис. 16.11.

Для получения дополнительной информации вы можете ознакомиться с официальной спецификацией консоли (**https://oreil.ly/osZhg**).

ГЛАВА 17
CSS

17.0. Введение

В среде современных браузеров CSS (cascading style sheets, каскадные таблицы стилей) не только позволяет создавать правила стиля, но и содержит набор API, которые можно использовать для дальнейшего улучшения вашего приложения.

Объектная модель CSS (CSS Object Model, CSSOM) позволяет программно устанавливать встроенные стили из кода JavaScript. Более того, вы даже можете изменять значения переменных CSS во время выполнения.

В главе 8 вы видели пример использования `window.matchMedia` для программной проверки медиазапроса на соответствие текущей странице.

В этой главе приведено несколько полезных рецептов, в которых используются некоторые из этих API, связанных с CSS. На момент написания книги некоторые из этих API пока еще плохо поддерживались браузерами. Всегда проверяйте совместимость с браузерами перед использованием API.

17.1. Выделение текстовых областей

Задача

Вы хотите применить эффект выделения к некоторому фрагменту текста в документе.

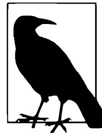
Решение

Создайте объект `Range` вокруг нужного фрагмента текста, затем используйте CSS Custom Highlight API для применения стилей выделения к этой области.

Первым шагом является создание объекта `Range`. Этот объект представляет собой область текста в документе. В примере 17.1 показана универсальная служебная функция для создания области с заданным текстовым узлом и текстом для выделения.

Пример 17.1. Создание объекта Range

```
/**
 * Для заданного текстового узла и подстроки создает объект Range,
 * охватывающий нужный фрагмент текста.
 */
function getRange(textNode, textToHighlight) {
  const startOffset = textNode.textContent.indexOf(textToHighlight);
  const endOffset = startOffset + textToHighlight.length;

  // Создает Range для выделяемого текста
  const range = new Range();
  range.setStart(textNode, startOffset);
  range.setEnd(textNode, endOffset);

  return range;
}
```



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости смотрите в CanIUse (**https://oreil.ly/wDJWH**).

Предположим, у вас есть HTML-элемент, показанный в примере 17.2.

Пример 17.2. Некоторая HTML-разметка

```
<p id="text">
  Это некоторый текст. Мы используем CSS Custom Highlight API для подсвечивания
  некоторого текста.
</p>
```

Если вы хотите выделить текст `"подсвечивания некоторого текста"`, вы можете использовать помощник `getRange` для создания выделения `Range` вокруг этого текста (пример 17.3).

Пример 17.3. Использование помощника getRange

```
const node = document.querySelector('#text');
const range = getRange(node.firstChild, 'подсвечивания
  некоторого текста');
```

Теперь, когда у вас есть диапазон, вам нужно зарегистрировать новую подсветку в реестре подсветки браузера. Для этого создайте новый объект подсветки `Highlight` с диапазоном, а затем передайте эту подсветку функции `CSS.highlights.set` (пример 17.4).

Пример 17.4. Регистрация подсветки

```
const highlight = new Highlight(range);
CSS.highlights.set('highlight-range', highlight);
```

Эти строки кода регистрируют выделение, но по умолчанию оно не имеет визуального эффекта. Далее вам нужно создать несколько стилей CSS, которые вы хотели бы применить к выделению. Это делается с помощью псевдоэлемента `::highlight`. Вы используете этот псевдоэлемент в сочетании с ключом, под которым вы зарегистрировали выделение `Highlight` в примере 17.4 (пример 17.5).

Пример 17.5. Стилизация подсветки

```
::highlight(highlight-range) {
  background-color: #fef3c7;
}
```

При применении этого стиля текст внутри диапазона теперь подсвечивается ярко-желтым цветом.

Обсуждение

Вы также можете выделить содержимое с помощью элемента `<mark>`. В примере 17.6 показано, как выделить некоторый текст с помощью `<mark>`.

Пример 17.6. Выделение с помощью элемента mark

```
<p id="text">
  Это некоторый текст. Мы используем элемент mark для
  <mark>подсвечивания некоторого текста</mark>.
</p>
```

Это дает тот же визуальный эффект, что и использование CSS Custom Highlight API, но основное отличие заключается в том, что использование `<mark>` предполагает вставку нового элемента в DOM. Это может быть сложно в зависимости от того, где вы добавляете новый элемент.

Например, если текст, который вы хотите выделить, состоит из нескольких элементов, вряд ли это можно будет выполнить с помощью элемента <mark> и при этом сохранить допустимый HTML. Рассмотрим HTML в примере 17.7.

Пример 17.7. Некоторая разметка для выделения

```
<p>
  Это абзац, который будет подсвечен.
</p>

<p>
  Выделение распространяется и на этот абзац. Это не выделено.
</p>
```

Если вы хотите выделить "который будет подсвечен. Выделение распространяется и на этот абзац", вы не сможете сделать это с помощью одного элемента <mark> (пример 17.8).

Пример 17.8. Недопустимый HTML-код

```
<p>
  Это абзац, <mark> который будет подсвечен.
</p>

<p>
  Выделение распространяется и на этот абзац</mark>. Это не выделено.
</p>
```

Это недопустимый HTML-код. Правильно было бы использовать два отдельных элемента <mark>, но тогда это не будет единая непрерывная выделенная область.

Применение CSS Custom Highlight API делает возможным такое выделение, создавая диапазон, охватывающий несколько тегов.

17.2. Предотвращение появления текста без стилизации

Задача

При использовании веб-шрифтов требуется избежать появления текста без стилизации.

Решение

Используйте CSS Font Loading API для явной загрузки начертаний шрифтов, которые вы хотите использовать в своем приложении. Отложите рендеринг любого текста до тех пор, пока шрифты не будут загружены.

Для того чтобы загрузить шрифт с помощью этого API, вы сначала создаете объект `FontFace`, содержащий данные о начертании шрифта, который вы хотите загрузить. В примере 17.9 используется шрифт Roboto.

Пример 17.9. Создание начертания шрифта Roboto

```
const roboto = new FontFace(
  'Roboto',
  'url(https://fonts.gstatic.com/s/roboto/v30/KFOmCnqEu92Fr1Mu72xKKTU1Kvnz.woff2)', {
    style: 'normal',
    weight: 400
  });
```

В документе есть глобальное свойство `fonts`, которое представляет собой набор шрифтов `FontFaceSet`, содержащий все начертания шрифтов, используемые в документе. Для того чтобы использовать нужное начертание шрифта, вам необходимо добавить его в набор шрифтов `FontFaceSet` (пример 17.10).

Пример 17.10. Добавление Roboto в глобальный набор шрифтов FontFaceSet

```
document.fonts.add(roboto);
```

Пока что вы определили только шрифт. Но еще ничего не загружено. Вы можете запустить процесс загрузки, вызвав команду `load` для объекта `FontFace` (пример 17.11). Это возвращает промис, который выполняется после загрузки шрифта.

Пример 17.11. Ждем загрузки шрифта

```
roboto.load()
  .then(() => {
    // Шрифт загружен и готов к использованию.
 });
```

Для того чтобы предотвратить появление текста без стилизации, вам нужно скрыть текст, в котором используется этот шрифт, до завершения загрузки. Если в вашем приложении отображается анимация начальной загрузки, вы можете продолжать анимацию до тех пор, пока не будут загружены необходимые шрифты, затем удалить загрузчик и запустить рендеринг приложения.

Если в вашем приложении используется несколько шрифтов, вы можете дождаться разрешения промиса `document.fonts.ready`. Этот промис будет выполнен, как только все шрифты будут загружены и готовы.

Обсуждение

При использовании веб-шрифтов с CSS шрифты объявляются с помощью правила `@font-face`, которое содержит URL-адрес загружаемого файла шрифта. Если текст отображается до завершения загрузки шрифта, используется резервный системный шрифт. Как только шрифт готов, текст перерисовывается правильным шрифтом. Это может привести к нежелательным эффектам, таким как изменение макета, если параметры шрифта отличаются.

Недостатком использования `@font-face` является то, что у вас нет возможности узнать, когда шрифт будет загружен и готов к использованию. Используя CSS Font Loading API, вы можете лучше контролировать загрузку шрифта и точно будете знать, когда можно безопасно начать использовать загружаемый шрифт для отображения текста.

Если при загрузке шрифта возникает ошибка, например, возможно, вы неправильно ввели URL-адрес шрифта, тогда промис, возвращаемый методом загрузки шрифта, отклоняется с ошибкой.

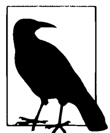
17.3. Анимация переходов DOM

Задача

Вы хотите показать анимированный переход при удалении или добавлении элементов в DOM.

Решение

Используйте View Transitions API, чтобы обеспечить анимированный переход между двумя состояниями.



Возможно, этот API пока поддерживается не всеми браузерами. Последние данные о совместимости см. в CanIUse (**https://oreil.ly/I8RFN**).

Этот API используется для применения эффекта перехода между двумя состояниями DOM. Для того чтобы начать переход, вызовите функцию `document.startViewTransition`. В качестве аргумента она принимает функцию обратного вызова. Вам необходимо внести изменения в DOM при выполнении этой функции обратного вызова.

Рассматривая пример 17.12, представьте, что у вас есть одностраничное приложение. Каждое представление приложения является HTML-элементом верхнего уровня с уникальным идентификатором. Для того чтобы переключиться между представлениями, вы можете удалить текущий вид и добавить новый.

Пример 17.12. Простой переход между представлениями

```
function showAboutPage() {
  document.startViewTransition(() => {
    document.querySelector('#home-page').style.display = 'none';
    document.querySelector('#about-page').style.display = 'block';
  });
}
```

Этот код применяет базовый эффект перекрестного затухания при переходе между двумя представлениями.

Если вы хотите настроить скорость перекрестного затухания, вы можете сделать это с помощью CSS, как показано в примере 17.13.

Пример 17.13. Замедление перехода

```
::view-transition-old(root),
::view-transition-new(root) {
  animation-duration: 2s;
}
```

Обсуждение

Эффект перехода работает путем создания снимка экрана текущего состояния DOM. Как только будут внесены изменения в DOM внутри обратного вызова, будет сделан еще один снимок экрана. Браузер создает на странице несколько псевдоэлементов и применяет анимированный переход между ними.

Созданные псевдоэлементы представляют собой следующее.

`::view-transition`

Наложение верхнего уровня, содержащее все переходы между видами.

`::view-transition-group(<name>)`

Отдельный переход между видами.

`::view-transition-image-pair(<name>)`

Содержит два изображения, между которыми осуществляется переход.

`:: view-transition-old(<name>)`

Изображение старого состояния DOM.

`::view-transition-new(<name>)`

Изображение нового состояния DOM.

Некоторые из этих псевдоэлементов принимают аргумент `name`. В качестве его значения может быть один из следующих вариантов.

`*`

Соответствует всем группам переходов между представлениями.

`root`

Соответствует корневой группе переходов, которая является именем по умолчанию, если пользовательское имя не задано.

`custom identifier`

Пользовательский идентификатор можно указать, установив свойство `view-transition-name` для элемента, который необходимо преобразовать.

Вы можете использовать селекторы CSS, чтобы выбрать эти псевдоэлементы и применить к ним различные анимации. Вы можете сделать это, создав правило `@keyframes` для анимации и применив эту анимацию к псевдоэлементам `::view-transition-old` или `::view-transition-new`.

17.4. Изменение таблиц стилей во время выполнения

Задача

Вы хотите динамически добавить CSS-правило в таблицу стилей страницы.

Решение

Для того чтобы добавить нужное правило, используйте метод `insertRule` из `CSSStyleSheet` (пример 17.14).

Пример 17.14. Добавление правила CSS

```
const [stylesheet] = document.styleSheets;
stylesheet.insertRule(`
  .some-selector {
    background-color: red;
  }
`);
```

Обсуждение

Этот трюк может потребоваться, если у вас есть новый HTML-контент, который динамически добавляется на страницу, например, в одностраничном приложении. Вы можете динамически добавлять правила стиля при добавлении нового контента.

17.5. Условная установка CSS-класса

Задача

Вы хотите применить CSS-класс к элементу только при выполнении определенного условия.

Решение

Используйте метод `toggle` для элемента `classList` (пример 17.15).

Пример 17.15. Условное переключение класса

```
// Предположим, что isExpanded - это переменная с текущим расширенным состоянием.
element.classList.toggle('expanded', isExpanded);
```

Обсуждение

При вызове метода `toggle` без второго аргумента добавляется имя класса, если оно в данный момент не задано, или имя удаляется, если оно уже было задано.

В дополнение к `toggle` вы можете использовать методы `add` и `remove` для управления списком классов, добавляя и удаляя заданное имя класса. Если вы вызываете `add`, когда имя класса уже задано, это не имеет никакого эффекта. Аналогично, если вы вызываете `remove`, когда имя класса не задано, это также не имеет никаких последствий.

17.6. Соответствие медиазапросам

Задача

Вы хотите проверить, удовлетворен ли определенный медиазапрос, с помощью JavaScript. Например, вы можете использовать медиазапрос `prefers-color-scheme`, чтобы определить, установлена ли в операционной системе пользователя темная тема.

Решение

Используйте `window.matchMedia` для оценки медиазапроса или отслеживания изменений (пример 17.16).

Пример 17.16. Проверка наличия темной цветовой схемы

```
const isDarkTheme = window.matchMedia('(prefers-color-scheme: dark)').matches;
```

Обсуждение

`window.matchMedia` возвращает объект `MediaQueryList`, который не только обладает свойством `matches`, но и позволяет отслеживать событие `change`. Это событие срабатывает, если результат медиазапроса изменяется.

Например, если настройки цветовой темы операционной системы пользователя изменяются во время работы вашего приложения, срабатывает событие `change` для запроса `prefers-color-scheme`. Затем вы можете проверить соответствие нового состояния (пример 17.17).

Пример 17.17. Отслеживание изменений медиазапроса

```
const query = window.matchMedia('(prefers-color-scheme: dark)');
query.addEventListener('change', () => {
  if (query.matches) {
    // переключиться на темную тему
  } else {
    // переключиться на светлую тему
  }
});
```

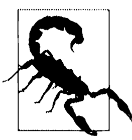
17.7. Получение вычисленного стиля элемента

Задача

Вы хотите найти определенный стиль CSS для элемента, который взят из таблицы стилей (не из встроенного стиля).

Решение

Используйте `window.getComputedStyle` для расчета окончательного стиля для элемента.



Используйте getComputedStyle экономно

Когда вы вызываете `getComputedStyle`, это заставляет браузер повторно вычислять стили и макет страницы, что может стать узким местом в производительности.

Рассмотрим HTML-элемент в примере 17.18 с применением некоторого стиля.

Пример 17.18. Немного HTML-кода со стилем

```
<style>
  #content {
```

```
    background-color: blue;
  }

  .container {
    background-color: red;
    color: white;
  }
</style>

<div id="content" class="container">Какого я цвета?</div>
```

Для того чтобы определить стили, которые будут применены к элементу, передайте элемент в `window.getComputedStyle` (пример 17.19).

Пример 17.19. Получение вычисленного стиля

```
const content = document.querySelector('#content');
const styles = window.getComputedStyle(content);
console.log(styles.backgroundColor);
```

Поскольку селектор ID обладает более высокой специфичностью, чем селектор классов, он выигрывает в конфликте и поэтому `styles.backgroundColor` — синий. В некоторых браузерах это может быть не строка "blue", а цветовое выражение, например `rgb(0, 0, 255)`.

Обсуждение

Свойство `style` элемента работает только для встроенных (inline) стилей.

Рассмотрим пример 17.20.

Пример 17.20. Элемент со встроенными стилями

```
<style>
  #content {
    background-color: blue;
  }
</style>

<div id="content" style="color: white;">Контент</div>
```

В этом примере свойство `color` задано как встроенный стиль, поэтому вы можете получить к нему доступ, обратившись к свойству `style`. Однако цвет фона берется из таблицы стилей и не может быть найден подобным образом (пример 17.21).

Пример 17.21. Проверка встроенных стилей

```
const content = document.querySelector('#content');
console.log(content.style.backgroundColor); // пустая строка
console.log(content.style.color); // 'white'
```

Поскольку getComputedStyle вычисляет окончательный стиль элемента, он содержит как стили таблицы стилей, так и встроенные стили (пример 17.22).

Пример 17.22. Проверка вычисленных стилей

```
const content = document.querySelector('#content');
const styles = window.getComputedStyle(content);
console.log(styles.backgroundColor); // 'rgb(0, 0, 255)'
console.log(styles.color); // 'rgb(255, 255, 255)'
```

ГЛАВА 18

Медиа

18.0. Введение

Современные браузеры имеют широкие возможности API для работы с видео- и аудиопотоками. WebRTC API поддерживает создание этих потоков при помощи таких устройств, как камеры.

Видеопоток может воспроизводиться в режиме реального времени внутри элемента <video>, и оттуда вы можете захватить кадр видео, чтобы сохранить его как изображение или загрузить в API. Элемент <video> также можно использовать для воспроизведения видео, записанного из потока.

До появления этих API для доступа к камере пользователя требовались подключаемые модули (плагины) браузера. Сегодня вы можете использовать API для захвата мультимедиа и потоковой передачи, чтобы начать считывание данных с камеры и микрофона пользователя с помощью всего лишь небольшого количества кода.

18.1. Запись экрана

Задача

Вы хотите захватить видео с экрана пользователя.

Решение

Используйте Screen Capture API для захвата видео с экрана, затем установите его в качестве источника элемента <video> (пример 18.1).

Пример 18.1. Захват видео с экрана

```
async function captureScreen() {
  const stream = await navigator.mediaDevices.getDisplayMedia();
  const mediaRecorder = new MediaRecorder(stream, {
```

```
    mimeType: 'video/webm'
  });

  mediaRecorder.addEventListener('dataavailable', event => {
    const blob = new Blob([event.data], {
      type: 'video/webm',
    });

    const url = URL.createObjectURL(blob);
    video.src = url;
  });

  mediaRecorder.start();
}
```

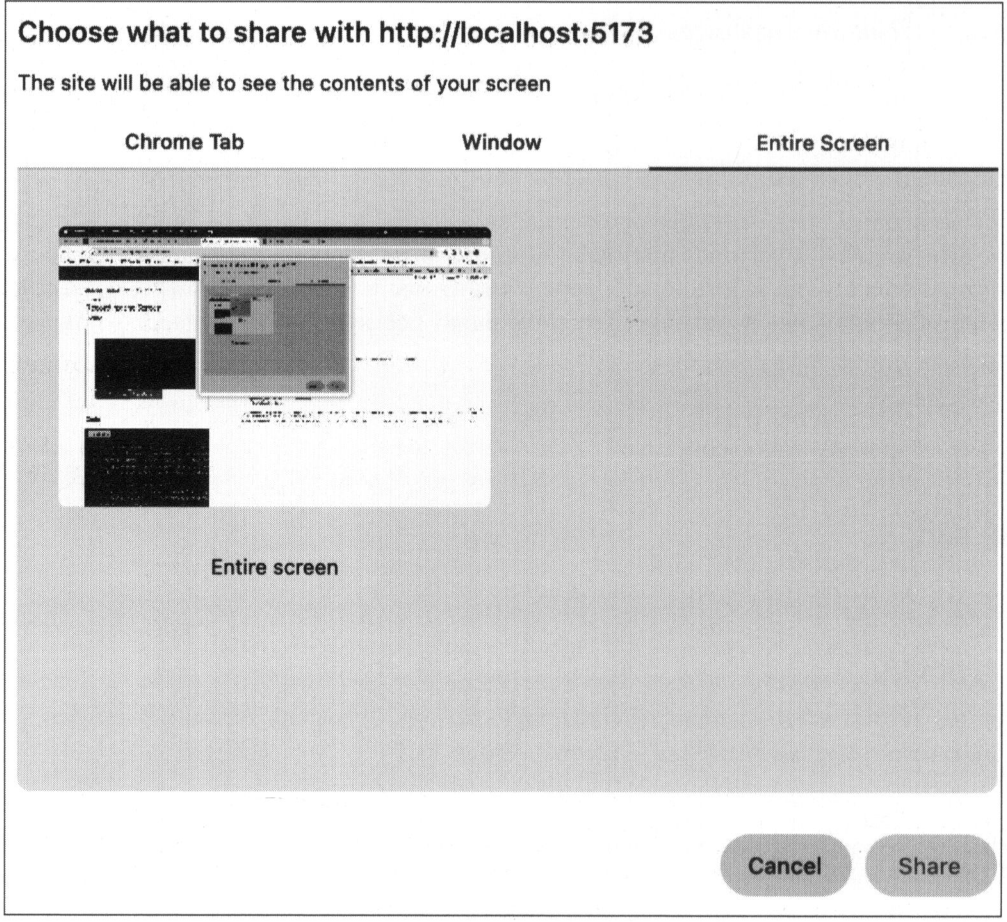
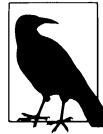


Рис. 18.1. Подсказка о начале записи экрана в Chrome на macOS



Содержимое экрана не передается в режиме реального времени в элемент <video>. Вместо этого данные с экрана сохраняются в памяти. Как только вы завершите захват экрана, записанное видео будет воспроизведено в элементе <video>.

Здесь происходит множество процессов. Для того чтобы запустить захват экрана, сначала вызывается `navigator.mediaDevices.getDisplayMedia()`. В зависимости от браузера и операционной системы вы получите своего рода подсказку о записи экрана (рис. 18.1).

Эта функция возвращает `Promise`, который преобразуется в `MediaStream` экрана пользователя. Как только это `Promise` выполняется, экран записывается, но данные еще никуда не передаются.

Для того чтобы остановить запись, нажмите кнопку, имеющуюся в браузере, чтобы прекратить публикацию, или вызовите функцию `mediaRecorder.stop()`. Это вызовет событие `dataavailable`.

Далее обработчик события формирует `Blob` (большой двоичный объект), содержащий захваченные видеоданные, и создает URL-адрес объекта. Затем вы можете присвоить атрибуту `src` значение URL этого объекта.

Как только это будет сделано, в браузере начнется воспроизведение записи экрана.

Обсуждение

В этом примере используется MIME-тип `video/webm`, который хорошо поддерживается браузерами. WebM — это открытый формат аудио- и видеофайлов, поддерживающий множество кодеков.

Если пользователь не даст разрешения на запись экрана, промис, возвращенный `getDisplayMedia`, будет отклонен с ошибкой.

В этом примере показано, как воспроизвести запись экрана в элементе <video>, но есть и другие действия, которые вы можете выполнить, если у вас есть Blob и его URL-адрес.

Например, вы можете отправить Blob на сервер, используя Fetch API (пример 18.2).

Пример 18.2. Загрузка захваченной записи экрана

```
const form = new FormData();
// Здесь "blob" - это Blob, созданный методом captureScreen.
formData.append('file', blob);

fetch('/api/video/upload', {
  method: 'POST',
  body: formData
});
```

Вы также можете запустить в браузере загрузку захваченного видео (пример 18.3).

Пример 18.3. Запуск загрузки по скрытой ссылке

```
const link = document.createElement('a');

// Здесь "url" - это URL, созданный методом captureScreen.
link.href = url;
link.textContent = 'Download';
link.download = 'screen-recording.webm';
link.click();
```

18.2. Захват изображения с камеры пользователя

Задача

Вы хотите активировать камеру пользователя и сделать снимок.

Решение

Используйте `navigator.mediaDevices.getUserMedia`, чтобы получить изображение с камеры. Сначала вам нужно создать несколько элементов, как показано в примере 18.4.

Пример 18.4. Разметка для захвата изображения с камеры

```
<style>
  #canvas {
    display: none;
  }

  #photo {
    width: 640px;
    height: 480px;
  }
</style>

<canvas id="canvas"></canvas>
<img id="photo">
<video id="preview">
```

Холст (canvas) скрыт, потому что это промежуточный шаг перед созданием изображения.

Общий подход заключается в следующем:

1. Отправьте видеопоток в элемент <video>, чтобы показать предварительный просмотр с камеры в режиме реального времени.
2. Если вы хотите сделать снимок, отобразите текущий видеокадр на холсте.
3. Создайте URL-адрес данных холста для генерации изображения в формате JPEG и задайте его в элементе <img>.

Сначала откройте видеопоток и прикрепите его к элементу <video> (пример 18.5).

Пример 18.5. Получение видеопотока

```
const preview = document.querySelector('#preview');

async function startCamera() {
  const stream = await navigator.mediaDevices.getUserMedia(
    {
      video: true,
      audio: false
    }
  );
  preview.srcObject = stream;
  preview.play();
}
```

Затем выполните захват изображения в ответ на нажатие кнопки или на другое событие (пример 18.6).

Пример 18.6. Захват изображения

```
// Это элемент <video>.
const preview = document.querySelector('#preview');

const photo = document.querySelector('#photo');
const canvas = document.querySelector('#canvas');

function captureImage() {
  // Измените размер холста в зависимости от плотности пикселов устройства.
  // Это поможет предотвратить появление размытого или пикселизированного изображения.
  canvas.width = canvas.width * window.devicePixelRatio;
  canvas.height = canvas.height * window.devicePixelRatio;
```

```
  // Получите 2D-контекст с холста и нарисуйте текущий видеокадр.
  const context = canvas.getContext('2d');
  context.drawImage(preview, 0, 0, canvas.width, canvas.height);

  // Создайте JPEG URL и установите его в качестве источника изображения.
  const dataUrl = canvas.toDataURL('image/jpeg');
  photo.src = dataUrl;
}
```

Обсуждение

Как и следовало ожидать, чтение с камеры вызывает опасения по поводу конфиденциальности. По этой причине при первом открытии камеры пользователя в браузере будет отправлен запрос на разрешение, который пользователь должен принять, чтобы предоставить доступ. Если этот запрос будет отклонен, промис, возвращенный `navigator.mediaDevices.getUserMedia`, будет отклонен с ошибкой.

18.3. Захват видео с камеры пользователя

Задача

Вы хотите записать видео с камеры пользователя и воспроизвести его в браузере.

Решение

Это решение состоит из нескольких шагов:

1. Используйте `getUserMedia`, чтобы открыть трансляцию с камеры.
2. Используйте элемент `<video>` для предварительного просмотра видео.
3. Используйте `MediaRecorder` для записи видео.
4. Воспроизведите записанное видео в элементе `<video>`.

Для этого рецепта вам понадобится элемент `<video>` и кнопки для запуска и остановки записи (пример 18.7).

Пример 18.7. Настройка элемента <video>

```
<video id="preview" muted></video>
<button id="record-button">Запись</button>
<button id="stop-record-button">Остановка записи</button>
```

Затем откройте видеопоток и установите элемент `<video>` для предварительного просмотра (пример 18.8).

Пример 18.8. Открытие аудио- и видеопотока

```
const preview = document.querySelector('#preview');
const stream = await navigator.mediaDevices.getUserMedia({
  video: true,
  audio: true
});
preview.srcObject = stream;
preview.play();
```

Как только поток будет открыт, следующим шагом будет настройка `MediaRecorder` (пример 18.9).

Пример 18.9. Настройка MediaRecorder

```
mediaRecorder = new MediaRecorder(stream, {
  mimeType: 'video/webm'
});

mediaRecorder.addEventListener('dataavailable', event => {
  const blob = new Blob([event.data], {
    type: 'video/webm',
  });

  const url = URL.createObjectURL(blob);

  // Очистите флаг "muted", чтобы воспроизведение было со звуком.
  preview.muted = false;

  // Установите источник видео элемента в только что созданном URL.
  preview.srcObject = null;
  preview.src = url;

  // Начните воспроизведение записи.
  preview.autoplay = true;
  preview.loop = true;
  preview.controls = true;
});
```

Последним шагом является подключение кнопок для запуска и остановки `MediaRecorder` (пример 18.10).

Пример 18.10. Добавление обработчиков событий кнопок

```
document.querySelector('#record-button').addEventListener('click', () => {
  mediaRecorder.start();
});

document.querySelector('#stop-record-button').addEventListener('click', () => {
  mediaRecorder.stop();
});
```

Обсуждение

Возможно, вы заметили, что для элемента <video> изначально был установлен атрибут `muted`. Медиапоток, который вы откроете, будет содержать как видео, так и аудио. Вы хотите предварительно просмотреть видео, но, вероятно, не захотите при этом прослушать и аудио — это приведет к немедленному воспроизведению записанного звука в динамиках, что может повлиять на качество записи или даже вызвать обратную связь с микрофоном. Для того чтобы предотвратить это, вы можете установить атрибут `muted` для элемента <video>.

Позже, когда придет время во всей полноте посмотреть то, что вы записали, вы снимете флажок `muted`, чтобы записанный звук был также воспроизведен.

18.4. Определение возможностей системной поддержки медиа

Задача

Вы хотите знать, поддерживается ли браузером определенный тип медиаконтента.

Решение

Используйте Media Capabilities API, чтобы запросить у браузера данные о поддержке медиаконтента. В результате вы узнаете, поддерживается данный тип или нет (пример 18.11).

Пример 18.11. Проверка медиавозможностей браузера

```
navigator.mediaCapabilities.decodingInfo({
  type: 'file',
  audio: {
    contentType: 'audio/mp3'
  }
```

```
}).then(result => {
  if (result.supported) {
    // MP3-аудио поддерживается!
  }
});

navigator.mediaCapabilities.decodingInfo({
  type: 'file',
  audio: {
    contentType: 'audio/webm;codecs=opus'
  }
}).then(result => {
  if (result.supported) {
    // WebM-аудио поддерживается с помощью opus codec.
  }
});
```

Обсуждение

В примере 18.11 показано несколько способов проверки поддержки аудиокодека. Media Capabilities API также позволяет вам проверять наличие поддержки определенного видеоформата. Вы можете выполнять запросы не только по кодеку, но и по другим атрибутам, таким как частота кадров, битрейт, ширина и высота кадра (пример 18.12).

Пример 18.12. Проверка поддерживаемого видеоформата

```
navigator.mediaCapabilities.decodingInfo({
  type: 'file',
  video: {
    contentType: 'video/webm;codecs=vp8',
    bitrate: 4000000, // 4 MB
    framerate: 30,
    width: 1920,
    height: 1080
  }
  }).then(result => {
  if (result.supported) {
    // Данная конфигурация WebM поддерживается.
  }
});
```

18.5. Применение видеофильтров

Задача

Вы хотите применить к видеопотоку эффект фильтра.

Решение

Выведите видеопоток в элемент <canvas> и примените к холсту CSS-фильтр.

Вы зададите видеопоток в качестве источника элемента <video>, как в рецепте 18.2. Однако в этом случае вы скроете элемент <video>, поскольку это всего лишь промежуточный шаг.

Затем, исходя из желаемой частоты кадров, визуализируйте каждый кадр видео в виде элемента <canvas>. Отсюда вы можете применить CSS-фильтры.

Во-первых, посмотрите на разметку (пример 18.13).

Пример 18.13. Разметка для примера видеофильтра

```
<canvas id="canvas"></canvas>
<video id="preview" style="display: none;"></video>
```

Затем откройте медиапоток и задайте его в элементе <video> (пример 18.14).

Пример 18.14. Настройка видеопотока

```
async function startCamera() {
  const stream = await navigator.mediaDevices.getUserMedia({
    video: true,
    audio: false
  });

  // Прицепите видеоэлемент к потоку.
  preview.srcObject = stream;
  preview.play();

  // Измените размер холста в зависимости от плотности пикселов устройства.
  // Это поможет предотвратить появление размытого или пикселизированного изображения.
  canvas.width = canvas.width * window.devicePixelRatio;
  canvas.height = canvas.height * window.devicePixelRatio;

  const context = canvas.getContext('2d');
  // Целевая частота кадров - 30 fps, отобразите каждый кадр на экране.
```

```
  setInterval(() => {
    context.drawImage(preview, 0, 0, canvas.width, canvas.height);
  }, 30 / 1000);
}
```

Теперь вы можете применить CSS-фильтр к элементу <canvas> (пример 18.15).

Пример 18.15. Применение фильтра

```
#canvas {
  filter: hue-rotate(90deg);
}
```

Обсуждение

Каждые 0,03 секунды на холст будет выводиться текущий кадр видео. По сути, это предварительный просмотр медиапотока с использованием <video> в качестве промежуточного элемента. Это связано с тем, что в настоящее время нет способа "нарисовать" видео из медиапотока непосредственно в элементе <canvas>.

В дополнение к установке фильтров с помощью CSS, вы также можете применить их, используя свойство filter в контексте 2D-холста.

ГЛАВА 19

Заключительные замечания

19.0. Введение

Я надеюсь, что рецепты и API, описанные в этой книге, показались вам полезными и интересными. Я рассчитываю на то, что вы смогли применить знания, почерпнутые из этой книги, для повышения уровня своих приложений на JavaScript.

19.1. В защиту сторонних библиотек

Одной из главных тем этой книги является тот факт, что вы можете многое сделать, не прибегая к услугам сторонних библиотек. Это верно, но не думайте, что вы должны избегать сторонних библиотек любой ценой. Иногда использование встроенных API-интерфейсов браузера избавляет от необходимости устанавливать дополнительные зависимости, но, возможно, вам придется добавить "склеивающий" код, чтобы адаптировать их к тому, чего вы пытаетесь достичь.

С некоторыми API-интерфейсами браузера может быть неудобно работать. Возьмем, к примеру, IndexedDB API. Это мощный уровень сохранения данных и доступа к ним, но его API основан на обратном вызове, что может создать сложности. Существуют библиотеки, которые используют IndexedDB и предоставляют более простой, а в некоторых случаях и более мощный API. Например, Dexie.js оборачивает IndexedDB в API, основанный на Promise.

В конце концов, все зависит от степени компромисса. Если у вас есть свободное место в вашем пакете JavaScript, которое вы можете использовать для упрощения работы разработчика, подумайте о потенциальной пользе сторонних библиотек.

19.2. Определяйте функции, а не версии браузера

Если вам нужно проверить, работает ли пользователь с браузером, поддерживающий нужный вам API, вы можете посмотреть на строку `user agent` и выяснить, какая версия браузера у пользователя. Однако старайтесь избегать этого. Такой метод

заведомо ненадежен, к тому же легко подделать строку пользовательского агента, чтобы выдать один браузер за другой.

Вместо этого определите, доступна ли какая-либо определенная функция. Например, если вы хотите проверить, поддерживает ли браузер IndexedDB, просто проверьте наличие свойства IndexedDB в объекте window (пример 19.1).

Пример 19.1. Проверка поддержки IndexedDB

```
if ('indexedDB' in window) {
  // IndexedDB поддерживается!
}
```

19.3. Полифилы

Если вам нужна поддержка старых браузеров, вы можете по-прежнему использовать некоторые из этих API с полифилом (polyfill). Полифил — это сторонняя библиотека JavaScript, которая добавляет недостающую функциональность. Библиотеки-полифилы могут быть не такими быстродействующими, как встроенные API, но они позволяют вам использовать более новые API в браузерах, которые в противном случае не поддерживали бы их.

Конечно, некоторые API-интерфейсы не могут быть полностью "полифилированы", поскольку они основаны на интеграции с собственными возможностями устройства, такими как акселерометр или геолокация. Если браузер не имеет возможности взаимодействия с этими системными службами, никакое количество стороннего кода не сможет восполнить этот пробел.

19.4. Заглядывая в будущее

На горизонте постоянно появляются более интересные API, которые еще больше расширят возможности браузерных приложений без использования подключаемых модулей или сторонних библиотек. В завершение книги в этом разделе кратко рассматриваются некоторые новые экспериментальные API, которые в ближайшем будущем еще больше обогатят браузерные приложения.

Web Bluetooth API

Вскоре вы сможете взаимодействовать с устройствами Bluetooth непосредственно в браузере, используя Web Bluetooth API. Он предоставляет интерфейс, основанный на промисах, для поиска и считывания информации о подключенных устройствах Bluetooth. Вы можете считывать такие данные, как уровень заряда батареи, или прослушивать уведомления от устройств.

Этот API работает путем взаимодействия с профилем GATT (Generic Attribute) устройства, который определяет поддерживаемые службы и характеристики устройства Bluetooth. Такой подход позволяет сохранить API универсальным, позволяющим гибко работать с любыми устройствами, поддерживающими GATT.

Web NFC API

Технология Near-field communication (NFC) позволяет устройствам обмениваться информацией, когда они находятся в непосредственной близости друг от друга. Web NFC API позволит устройствам обмениваться сообщениями и информацией с помощью аппаратного обеспечения NFC.

Этот API предоставляет возможность обмениваться сообщениями с помощью обмена данными в формате NFC Data Exchange Format (NDEF). Это стандартизированный формат, опубликованный форумом NFC.

EyeDropper API

EyeDropper API позволит вам выбирать цвет из пикселов на экране с помощью инструмента "пипетка" (eye dropper). Этот инструмент будет работать как внутри, так и за пределами окна браузера, обеспечивая возможность выбора цвета из любого места на экране.

Вы можете создать пипетку, вызвав конструктор `EyeDropper`. Eye Dropper предоставляет открытый метод, который отображает интерфейс пипетки на экране и возвращает промис. Как только вы выделяете пиксель с помощью пипетки, изображение меняется в соответствии с цветом выбранного пиксела.

Barcode Detection API

Этот API предоставит вашим приложениям возможность считывать штрих-коды и QR-коды. Он поддерживает множество стандартных типов штрих-кодов. Это будет универсальный API, который сможет считывать штрих-коды из множества различных источников изображений: элементов изображений и видео, больших двоичных объектов (Blobs), холстов (canvas) и многого другого.

Обнаружение штрих-кодов осуществляется путем передачи данных изображения в метод обнаружения `BarcodeDetector`. В результате возвращается промис, который преобразуется в данные о любых обнаруженных штрих-кодах и их значениях.

Cookie Store API

Используемый сегодня механизм работы с файлами cookie в браузере не очень удобен. Свойство `document.cookie` — это отдельная строка, содержащая сопоставления пары "ключ/значение" имен файлов cookie и значений для текущего сайта.

Новый Cookie Store API предоставляет асинхронный, более надежный интерфейс для доступа к информации о файлах cookie. Вы можете просмотреть подробные

сведения об одном файле cookie с помощью метода `CookieStore.get`, который возвращает Promise, преобразующийся в информацию о файле cookie с заданным именем.

API также позволяет вам прослушивать события `change`, которые запускаются при каждом изменении данных cookie.

Платежные API

API запроса платежа предоставляет веб-сайту возможность инициировать платеж в браузере. Затем вы можете использовать Payment Handler API для обработки платежа без необходимости перенаправления на сторонний веб-сайт.

Так можно обеспечить более стабильную работу при использовании внешнего платежного процессора.

Узнайте, что будет дальше

Интернет постоянно меняется. Если вы хотите узнать, какие еще API-интерфейсы веб-браузера появятся в ближайшее время, вот несколько полезных ресурсов.

- В MDN Web Docs (**https://oreil.ly/PqBPh**) есть страница веб-интерфейсов (**https://oreil.ly/YTWkO**), на которой представлен обзор текущих, будущих и экспериментальных API-интерфейсов.
- Страница стандартов и проектов W3C (Standards and drafts W3C; **https://oreil.ly/Xu47E**) содержит доступный для поиска каталог стандартов и черновых спецификаций для всех уровней разработки.

Предметный указатель

A

AnimationTimeline, интерфейс 142
API запроса платежа 295

B

Barcode Detection API 294
Battery Status API 231
Beacon API 65, 71
Bluetooth 293

C

Callback 21
Cascading style sheets (CSS) 269
- CSS-класс, применение к элементу 277
- изменение во время выполнения кода 276

Clipboard API 239
Constraint Validation API 122
- валидация формы 119

Cookie 294
Cookie Store API 294
Cross-site scripting (XSS) attacks 37
CSS Custom Highlight API 269
CSS Font Loading API 273

D

Date API 177
DOMHighResTimeStamp, объект 245

E

Element, интерфейс 131
EventSource API 72
EventTarget, интерфейс 21
EyeDropper API 294

F

Fetch API 65, 112, 175
- загрузка файла 70
- отправка:
  - GET-запроса 67
  - POST-запроса 69

File API 161
FileReader, объект 163
FormData API 71, 110, 111, 113

G

Geocoding API 237
Geolocation API 235, 237
Google Maps API 237
Google Maps Embed API 237

I

IDBKeyRange, интерфейс 89
IDBRequest, интерфейс 79
IndexedDB 77
- запрос с индексом 87
- курсор 90, 92
- обновление существующей БД 85
- поиск значений с помощью курсора 90

- ◊ промисы 94
- ◊ разбивка набора данных на страницы 92
- ◊ создание, чтение и удаление объектов в БД 79

Internationalization API 177
IntersectionObserver 98
- ◊ видеопауза 102
- ◊ обертывание промисом 101
- ◊ отслеживание элемента 107
- ◊ прокрутка бесконечная 108

L

Light DOM 193

M

Math.max, функция 33, 183
Math.min, функция 183
Media Capabilities API 288
MutationObserver 97
- ◊ анимация высоты 103

N

Navigation Timing API 245
Near-field communication (NFC) 294
Network Information API 234

O

OpenStreetMaps 237

P

Payment Handler API 295
Performance API 261
Permissions API 236, 240
Promise 22

Q

QR-код 294

R

ResizeObserver 98
- ◊ изменение элемента в зависимости от размера 106

Resource Timing API 245
RFC 3986 51

S

Screen Capture API 281
Server-sent events (SSE) 65, 72
Shadow DOM 192, 197
SpeechSynthesis API 152
Storage, интерфейс 36

T

Temporal 177

U

URL:
- ◊ маршрутизатор на стороне клиента 60
- ◊ параметр запроса:
  - – добавление 56
  - – удаление 54
  - – чтение 58
- ◊ сопоставление с шаблонами 62

URL Pattern API 63
URL-адрес:
- ◊ данных 167
- ◊ объекта 167

User Timing API 245

V

Vibration API 242
View Transitions API 274

W

Web Bluetooth API 293
Web NFC API 294

Web Share API 241
Web Speech API:
- ◊ Promise-помощник 155
- ◊ голос, доступный для синтеза речи 156
- ◊ приостановка речи автоматическая 160
- ◊ распознавание речи 150
- ◊ синтез речи 151, 157
  - – настройка 159
- ◊ текст продиктованный, добавление в поле 152

Web Storage API 36
- ◊ недостатки 37
- ◊ отслеживание изменений 46
- ◊ поддержка 38
- ◊ поиск ключей 47
- ◊ получение и установка элементов 37
- ◊ сохранение:
  - – простых объектов 40
  - – сложных объектов 41
  - – строковых данных 39
- ◊ удаление данных 49

WebRTC API 281
WebSocket API 65, 74

X

XMLHttpRequest 65, 66

А

Аккумулятор, зарядка 231
Анимация 30
- ◊ загрузки 146
- ◊ запуск и остановка 135
- ◊ изменений высоты 103
- ◊ индикатор прокрутки 141
- ◊ ключевых кадров 130
  - – JavaScript 131
- ◊ множественная 144
- ◊ настройка пользователя 148
- ◊ объект 131
- ◊ перехода при удалении/добавлении элемента 274
- ◊ реверсирование 137
- ◊ сглаживание 134
- ◊ шкала временна́я 142

Атака с использованием кроссайтовых сценариев 37

Б

База данных версионная 83

В

Валидация данных пользователя 110
Веб API 13
Веб-компонент 190
- ◊ дата:
  - – произвольная 194
  - – текущая 193
- ◊ диалога подтверждения 218
- ◊ загрузка отложенная 202
- ◊ карточки профильной 200
- ◊ обратной связи 196
- ◊ отзыва пользователя 196
- ◊ раскрытия информации 204
- ◊ создание 190

Веб-сокет 74
Видео 102
- ◊ загрузка в качестве URL-адреса 166
- ◊ запуск автоматический 103
- ◊ захват с экрана 281

Видеофильтр 290
Выражение регулярное в форме 118
Высота звука 159

Г

Геокодирование 237
- ◊ обратное 237

Д

Дата:
- ◊ разница дат, форматирование 179

◊ форматирование 178
– части 178
Директива %c 255

З

Загрузка изображения отложенная 99
Запрос:
◊ IndexedDB 79
◊ по индексу 87
Значение пороговое 98

И

Изображение:
◊ загрузка:
– из URL-адреса 164
– отложенная 99
– с помощью перетаскивания 167
◊ резервное 24
Индекс хранилища объектов 77
Интернационализация 177
Интерфейс пользователя 211
◊ details 212
◊ подробности 212
Источник страницы 36

К

Кадр ключевой 130
Камера пользователя:
◊ захват видео 286
◊ захват изображения 284
Каскадные таблицы стилей 269
◊ CSS-класс, применение к элементу 277
◊ изменение во время выполнения кода 276
Ключ:
◊ диапазон 89
– страницы 89
◊ дублирующийся 83
◊ объекта хранилища 77
– внешний 78
– встроенный 78
– путь к ключу 78
◊ уникальный 83
Кодирование процентное 57
Консоль:
◊ журнал:
– разделение типов сообщений 256
– сообщений 261
◊ массив объектов в таблице 258
◊ переменная и ее значение 264
◊ проверка ожидаемого значения 266
◊ регистратор именованный 257
◊ свойство объекта 267
◊ стилизация вывода 255
◊ счетчик 263
◊ таймер 260
◊ трассировка стека 265
Контент, совместное использование 241
Коэффициент пересечения 98
Курсор базы данных 90, 92

М

Массив имен, сортировка 188
Маячок 71
Медиа 281
◊ поддержка браузером 288

Н

Наблюдатель 97

О

Объект большой двоичный (Blob) 175
Окно:
◊ всплывающее 212, 222
– позиционирование относительно элемента 224
– управление ручное 223
◊ диалоговое 211
– модальное 214
– немодальное 214
– подтверждения 215, 218
– с предупреждением 212
Ошибка:
◊ аппаратного обеспечения 154
◊ промиса 154
◊ сетевая 154

П

Переход при появлении элемента в поле зрения 107
Плюрализация 184
Подсказка всплывающая 227
Подсчет символов, слов и предложений 186
Полифил 13, 293
Правила множественного числа 184
Приложение одностраничное 51
Производительность:
- время запроса к ресурсу 248
- загрузки веб-страниц 246
- задача многоступенчатая 251
- измерение 245
- отметка 245
- показатель 245
  - прослушивание 253
- рендеринга 249
- ресурс:
  - самый быстрый 248
  - самый медленный 247

Прокрутка бесконечная 108
Промис 22
- обертывание API 33
- параллельное выполнение 29
- связывание с промисом 26
- цепочка 26

Прослушиватель 22

Р

Ресурс:
- время запроса 248
- самый быстрый 248
- самый медленный 247

Речь:
- обработка 151
- приостановка автоматическая 160
- распознавание 150
- синтез 151, 157
  - настройка 159

С

Сеть, состояние 234
Система файловая 161
Скорость произнесения 159
Словоформы 184
Слот 192
- именованный 192

Слушатель события. *См.* Прослушиватель
Событие 21
- отправляемое сервером 65, 72

Список, форматирование 187
Стиль элемента 278

Т

Текст:
- загрузка из файла 161
- запрет без стилизации 272
- копирование и вставка 238
- эффект выделения 269

Транзакция 78

У

Уведомление 212
- отображение 229

Устройство:
- вибрация 242
- доступ общий 241
- зарядка аккумулятора 231
- местоположение 235
  - отображение на карте 237
- ориентация 243

Ф

Файл:
- доступ к файлу в локальной системе 170
- загрузка с помощью перетаскивания 175

Форма 110
- elements, свойство 112
- валидация 119
- выражение регулярное 118
- заполнение полей из локального хранилища 111
- отправка:
  - в формате JSON 114
  - с помощью Fetch и FormData API 112

◊ поле:
  – обязательное, создание 116
  – проверка асинхронная 127
  – числовое, ограничение при вводе 117
◊ флажок, проверка установки 124
◊ шаблон валидации 118

Функция:
◊ обратного вызова 21
◊ фабричная 45

Х

Хранилище:
◊ локальное 36
◊ объектов 77
◊ сессий браузера 36

Ц

Цена, форматирование 183

Ч

Частота произнесения 159

Число:
◊ округление 182
◊ форматирование 181
  – с единицей измерения 184

Ш

Шкала временна́я 142

Штрих-код 294

Э

Экспорт данных 171
◊ со ссылкой для скачивания 173

Элемент:
◊ <canvas> 290
◊ <details> 212
◊ <dialog> 211, 212, 215
◊ <img> 99, 100, 169
◊ <input> 70, 116, 117
◊ <mark> 271
◊ <slot> 192
◊ <summary> 221
◊ <template> 191
◊ <video> 102, 281, 283, 286
◊ пользовательский 190
  – регистрация 191
  – шаблон 191

Об авторе

Джо Аттарди (Joe Attardi) имеет более чем 20-летний опыт создания интерфейсного программного обеспечения и разработал множество браузерных приложений. Он также обладает богатым опытом работы с интерфейсом для Nortel, Dell, Constant Contact, Salesforce и Synopsys. Специализируется в разработке на JavaScript и TypeScript.

Об изображении на обложке

На обложке книги "Web API. Сборник рецептов" изображен златоголовый кетцаль (*pharomachrus auriceps*). Эти птицы обитают во влажных лесах от Панамы до Боливии.

Слово "кетцаль" (quetzal) происходит от слова "кетцалли" (quetzalli), что в переводе с ацтекского языка науатль означает "длинное зеленое оперение". Кетцали известны своим переливчатым зеленым оперением и красным брюшком. Золотоголовый кетцаль назван так из-за своей блестящей золотистой головы. У самок больше коричневых перьев, чем у самцов. Самки весят от 154 до 182 граммов и достигают 33–36 см в длину, самцы крупнее.

Кетцали ведут одиночный образ жизни до периода размножения, когда самцы и самки спариваются и строят гнездо в гниющем стволе дерева. Самки откладывают 1–2 бледно-голубых яйца, а затем обе птицы выводят потомство и разделяют ответственность за кормление птенцов. Птицы питаются в основном фруктами, и поэтому они играют важную роль в распространении семян фруктов в местах своего обитания.

Златоголовый кетцаль распространен в пределах своего ареала и отнесен Международным союзом охраны природы (МСОП — International Union for Conservation of Nature, IUCN) к видам, вызывающим наименьшее беспокойство с точки зрения охраны природы. Многие животные, изображенные на обложках O'Reilly, находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери (Karen Montgomery) по мотивам старинной гравюры Рутледжа (Routledge) "Естественная история" (Picture Natural History).

Джо Аттарди

Web API. Сборник рецептов

Перевод с английского

ТОО "АЛИСТ"
010000, Республика Казахстан,
г. Астана, пр. Сарыарка, д. 17, ВП 30

Подписано в печать 02.09.25.
Формат $70\times100^1/_{16}$. Печать офсетная. Усл. печ. л. 24,51.
Тираж 1000 экз. Заказ № 15378.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, РФ, М.О., г. Чехов, ул. Полиграфистов, д. 1

Web API. Сборник рецептов

JavaScript предоставляет веб-разработчикам богатый арсенал средств для создания динамичных и интерактивных пользовательских интерфейсов прямо в браузере. Сегодня большую часть этой функциональности обеспечивают сами браузеры. Современные Web API позволяют веб-приложениям реализовать то, что ранее требовало установки внешних плагинов. Некоторые из этих интерфейсов еще находятся на стадии разработки и тестирования, но многие из них уже полностью готовы к использованию.

Это практическое руководство предлагает коллекцию прикладных примеров использования новейших возможностей браузерных API. Автор помогает разобраться, какие задачи можно решить с их помощью и как применять эти инструменты на практике. Поскольку речь идет о стандартизированных технологиях, вы всегда можете обратиться к надежной документации, например, на MDN Web Docs. Полученные знания пригодятся вам в проектах любой сложности — от личных до корпоративных.

- **Добавьте веб-приложениям функции, схожие с возможностями современных браузеров**
- **Познакомьтесь с широким набором инструментов, предлагаемых актуальными браузерными API**
- **Изучите перспективные API-интерфейсы, находящиеся в стадии разработки**
- **Используйте новые элементы интерфейса, такие как диалоговые окна, без подключения сторонних библиотек**
- **Создавайте мощные и интерактивные приложения с глубокой интеграцией в возможности пользовательских устройств**
- **Изучите реализуемую браузерами модель разрешений для предоставления доступа к таким функциям, как геолокация и push-уведомления**

«Книга Джо Аттарди охватывает широкий спектр браузерных API и предлагает ценные практические примеры, понятные разработчикам с любым уровнем подготовки. Благодаря ясному стилю изложения и вдумчивым объяснениям обучение превращается в увлекательный процесс. Настоятельно рекомендую всем, кто занимается веб-разработкой!»

— **Сара Шук**, программист, основатель компании Shook

«Эта книга познакомила меня с API и подарила мне новые идеи, о существовании которых я даже не подозревал».

— **Бретт Литтл**, инженер-программист, компания Shopify

Джо Аттарди — разработчик с более чем двадцатилетним опытом создания пользовательских интерфейсов и браузерных приложений. В его портфолио — работа с API для таких компаний, как Nortel, Dell, Constant Contact, Salesforce и Synopsys. Специализируется на разработке с использованием JavaScript и TypeScript.

ISBN 978-601-12-3681-2

9 786011 236812

alist