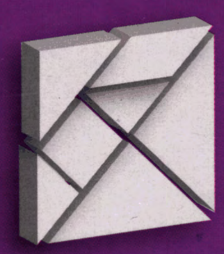
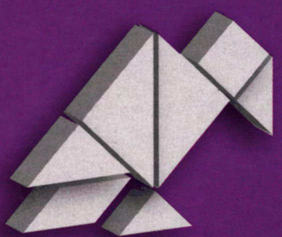
Владимир Дронов

JavaScript

Дополнительные уроки для начинающих





Владимир Дронов

JavaScript

Дополнительные уроки для начинающих

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.438JavaScript
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 JavaScript. Дополнительные уроки для начинающих. — СПб.: БХВ-Петербург, 2022. — 352 с.: ил. — (Для начинающих)

ISBN 978-5-9775-6781-7

В книге 20 иллюстрированных уроков, более 20 практических упражнений на тему программирования веб-сценариев и 18 заданий для самостоятельной работы. Простым языком, кратко, наглядно рассказано о новых программных инструментах, появившихся в последние годы в языке JavaScript. Описаны новый синтаксис объявления классов, промисы, асинхронные функции, итераторы, генераторы и модули. Рассказано о новом загрузчике файлов, встроенной в веб-обозреватель СУБД, HTML-компонентах и шаблонах, средствах для захвата видео, наложении эффектов на звук. Рассмотрены технологии WebSocket для программирования веб-чата и WebRTC для создания службы видеотелефонии. Описаны прогрессивные веб-приложения (PWA), посредники и программируемый кэш, вывод всплывающих оповещений и установка PWA в операционной системе.

Электронный архив на сайте издательства содержит коды всех примеров и результаты выполнения упражнений.

Для начинающих веб-разработчиков

УДК 004.438JavaScript
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Карины Соловьевой*

Подписано в печать 06.07 21.
Формат 70×100[1]/16. Печать офсетная Усл. печ л 28,38
Тираж 1200 экз. Заказ № 1727
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул , 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О , г Чехов, ул Полиграфистов, д 1

ISBN 978-5-9775-6781-7

© ООО "БХВ", 2022
© Оформление ООО "БХВ-Петербург", 2022

Оглавление

Предисловие ... **10**
Зачем нужна еще одна книга по JavaScript? ... 10
Что описывает новая книга? ... 11
Что вам понадобится? ... 12
Типографские соглашения ... 12

ЧАСТЬ I. JAVASCRIPT: НОВЫЕ ИНСТРУМЕНТЫ ... **15**

Урок 1. Классы, функции и деструктурирование ... **17**
1.1. Новый синтаксис объявления классов ... 17
1.1.1. Старый синтаксис объявления классов и его недостатки ... 17
1.1.2. Новый синтаксис объявления классов ... 18
1.1.3. Закрытые свойства и методы ... 21
1.1.4. Статические свойства и методы ... 22
1.1.5. Наследование классов ... 23
1.1.6. Анонимные классы ... 24
1.2. Упражнение. Создаем веб-компонент — спойлер ... 25
1.3. Упражнение. Совершенствуем спойлер ... 30
1.4. Новые инструменты для работы с функциями ... 33
1.4.1. Функции с произвольным количеством параметров. Оператор упаковки-распаковки ... 33
1.4.2. Использование оператора упаковки-распаковки в вызовах функций ... 34
1.5. Упражнение. Реализуем массовую инициализацию спойлеров ... 34
1.6. Деструктурирование ... 36
1.6.1. Деструктурирование массивов ... 36
1.6.2. Деструктурирование объектов ... 37
1.7. Упражнение. Упрощение кода спойлера *Spoiler2* ... 38
1.8. Самостоятельное упражнение ... 39

Урок 2. Промисы и асинхронные функции ... **40**
2.1. Промисы ... 40
2.1.1. Старые приемы выполнения асинхронных операций и их недостатки ... 40
2.1.2. Промисы. Создание промисов ... 41
2.1.3. Обработка промисов ... 43
2.1.4. Массовая обработка промисов ... 45
2.1.5. Дополнительные инструменты для работы с промисами ... 47
2.2. Упражнение. Загрузка файлов посредством AJAX ... 48
2.3. Упражнение. Массовая загрузка файлов ... 51
2.4. Асинхронные функции и оператор ожидания ... 53
2.4.1. Асинхронные методы в классах ... 55
2.5. Самостоятельные упражнения ... 55

Урок 3. Итераторы и генераторы ... 56
3.1. Цикл перебора последовательности ... 56
3.2. Итераторы ... 57
3.3. Упражнение. Вычисляем числа Фибоначчи ... 59
3.4. Генераторы ... 61
3.4.1. Методы-генераторы в классах ... 63
3.5. Упражнение. Вычисление квадратных корней ... 64
3.6. Асинхронные итераторы и генераторы ... 65
3.6.1. Асинхронные итераторы и цикл перебора с ожиданием ... 65
3.6.2. Асинхронные генераторы ... 67
3.6.2.1. Асинхронные методы-генераторы ... 67
3.7. Упражнение. Массовая загрузка файлов, вариант 2 ... 68
3.8. Самостоятельные упражнения ... 69

Урок 4. Модули ... 70
4.1. Старый способ скрытия деталей реализации и его недостатки ... 70
4.2. Модули ... 70
4.2.1. Экспорт ... 72
4.2.2. Импорт ... 74
4.2.3. Прогон ... 76
4.3. Упражнение. Реорганизация кода спойлера ... 76
4.4. Самостоятельное упражнение ... 78

ЧАСТЬ II. HTML API: НОВЫЕ ПОЛЕЗНЫЕ МЕЛОЧИ ... 79

Урок 5. Детектор видимости ... 81
5.1. Использование детектора видимости ... 82
5.1.1. Создание детектора видимости ... 82
5.1.2. Указание отслеживаемых элементов ... 84
5.1.3. Отслеживание попадания элементов в область видимости ... 85
5.2. Упражнение. Делаем подсвечивающиеся изображения ... 86
5.3. Самостоятельное упражнение ... 88

Урок 6. Загрузчик данных AJAX ... 89
6.1. Отправка запроса на загрузку данных ... 89
6.1.1. Задание заголовков HTTP-запроса ... 90
6.1.2. Управление кэшированием загруженных данных ... 92
6.1.3. Использование объектов запросов ... 93
6.2. Получение загруженных данных ... 94
6.3. Упражнение. Загружаем и выводим список языков программирования ... 96
6.4. Самостоятельные упражнения ... 98

Урок 7. Встроенная СУБД ... 99
7.1. Введение во встроенную СУБД ... 99
7.2. Создание баз данных ... 101
7.2.1. Создание и удаление самих баз данных ... 101
7.2.2. Создание и удаление хранилищ ... 104
7.2.3. Создание индексов ... 106
7.2.4. Создание и удаление индексов в существующих хранилищах ... 107

7.3. Запись и выборка данных 107
7.3.1. Открытие базы данных 107
7.3.2. Добавление документов. Работа с транзакциями 108
7.3.2.1. Этап 1: запуск транзакции 108
7.3.2.2. Этап 2: получение хранилища 109
7.3.2.3. Этап 3: собственно добавление документов 109
7.3.2.4. Подтверждение и откат транзакций 110
7.2.3.5. Добавление начальных документов 112
7.3.3. Поиск документов 113
7.3.3.1. Поиск документа по ключу 113
7.3.3.2. Поиск документа по значению индексированного свойства 113
7.3.3.3. Поиск документа в диапазоне значений 114
7.3.4. Выборка и фильтрация документов 116
7.3.4.1. Последовательная выборка документов. Курсоры 116
7.3.4.2. Одновременная выборка всех документов 119
7.3.5. Получение количества документов 120
7.3.6. Правка документов 120
7.3.6.1. Правка произвольного документа 120
7.3.6.2. Правка документов в процессе их выборки 121
7.3.7. Удаление документов 122
7.3.7.1. Удаление произвольных документов 122
7.3.7.2. Удаление документов в процессе их выборки 122
7.3.8. Закрытие базы данных 123
7.4. Упражнение. Пишем веб-приложение — телефонную книгу 123
7.5. Отладочные инструменты для работы с базами данных 131
7.6. Самостоятельное упражнение 134

Урок 8. Фоновые задачи 135
8.1. Фоновые задачи 135
8.1.1. Регистрация и выполнение фоновых задач 136
8.1.2. Получение сведений о времени, отводимом на выполнение фоновой задачи 137
8.1.3. Отмена фоновых задач 137
8.2. Синхронный вывод на экран 138
8.3. Упражнение. Вычисление чисел Фибоначчи, вариант 2 139
8.4. Самостоятельное упражнение 141

ЧАСТЬ III. HTML-КОМПОНЕНТЫ И ШАБЛОНЫ 143

Урок 9. HTML-компоненты 145
9.1. Создание HTML-компонентов 145
9.1.1. Объявление класса HTML-компонента 146
9.1.1.1. Создание скрытой DOM 146
9.1.1.2. Создание внутренней структуры HTML-компонента 147
9.1.1.3. Передача параметров HTML-компонентам 148
9.1.1.4. Получение содержимого тега HTML-компонента 149
9.1.2. Регистрация HTML-компонентов 150
9.2. Размещение HTML-компонентов на веб-странице 151
9.3. Упражнение. Пишем HTML-компонент — спойлер 152
9.4. Дополнительные инструменты для работы с HTML-компонентами 155
9.4.1. Волшебные методы HTML-компонентов 155

9.4.2. Средства CSS для работы с HTML-компонентами 156
9.4.3. Проверка регистрации HTML-компонентов 158
9.5. Самостоятельные упражнения 158

Урок 10. Шаблоны и слоты 159
10.1. Шаблоны 159
10.1.1. Написание шаблонов 159
10.1.2. Использование шаблонов 160
10.2. Слоты 161
10.2.1. Создание слотов 162
10.2.2. Использование слотов 163
10.3. Упражнение. Пишем HTML-компонент — спойлер, вариант 2 164
10.4. Дополнительные инструменты для работы с шаблонами и слотами 166
10.5. Самостоятельное упражнение 166

ЧАСТЬ IV. МУЛЬТИМЕДИА 169

Урок 11. Работа со встроенной камерой, часть 1 171
11.1. Захват видео с камеры 171
11.2. Запись видео 174
11.3. Упражнение. Пишем веб-приложение для записи видео 178
11.4. Упражнение. Реализуем выгрузку записанного видео на сервер 183
11.5. Захват и запись звука 186
11.6. Захват видео с экрана 187
11.7. Захват статичных изображений 190
11.8. Самостоятельные упражнения 191

Урок 12. Работа со встроенной камерой, часть 2 192
12.1. Задание параметров видео и звука 192
12.1.1. Параметры, доступные для указания 192
12.1.1.1. Параметры видео 193
12.1.1.2. Параметры звука 195
12.1.2. Получение списка поддерживаемых параметров 197
12.1.3. Смена параметров видео и звука во время захвата 197
12.2. Упражнение. Реализуем переключение между фронтальной и тыловой камерами 198
12.3. Задание параметров статичных изображений 201
12.3.1. Параметры, доступные для указания 201
12.3.2. Получение допустимых значений параметров 202
12.4. Получение текущих параметров 203
12.4.1. Получение параметров видео и звука 203
12.4.2. Получение параметров статичных изображений 204
12.5. Самостоятельные упражнения 205

Урок 13. Обработка звука 206
13.1. Как выполняется обработка звука 206
13.2. Реализация обработки звука 207
13.2.1. Создание звукового контекста 207
13.2.2. Создание источника звука 208
13.2.3. Создание и присоединение обработчиков звука 209
13.2.4. Присоединение получателя звука 209
13.2.5. Решение проблемы с приостановленным звуковым контекстом 210

13.3. Обработчики звука 211
13.3.1. Эффект панорамирования 211
13.3.2. Управление усилением 212
13.3.3. Биквадратный фильтр 212
13.4. Упражнение. Пишем аудиопроигрыватель с усилением басов 215
13.5. Дополнительные инструменты для обработки звука 217
13.6. Самостоятельное упражнение 218

Урок 14. Визуализация звука 219
14.1. Анализатор звука 219
14.1.1. Создание анализатора звука 219
14.1.2. Вывод осциллограммы 220
14.1.3. Вывод спектра 221
14.2. Упражнение. Пишем аудиопроигрыватель, выводящий осциллограмму 222
14.3. Упражнение. Пишем аудиопроигрыватель, выводящий спектр 225
14.4. Самостоятельные упражнения 228

ЧАСТЬ V. РАБОТА С СЕТЬЮ 229

Урок 15. WebSocket 231
15.1. Клиент WebSocket 232
15.1.1. Установление соединения с сервером 232
15.1.2. Отправка данных серверу 233
15.1.2.1. Отправка файлов серверу 234
15.1.3. Получение данных от сервера 235
15.1.3.1. Получение файлов от сервера 235
15.1.4. Разрыв соединения 236
15.1.5. Обработка ошибок 236
15.2. Сервер WebSocket 236
15.2.1. Библиотека Workerman 237
15.2.2. Класс слушателя 237
15.2.2.1. Настройка слушателя для работы по защищенной редакции WebSocket 239
15.2.3. Класс соединения 241
15.2.4. Запуск и остановка сервера 242
15.3. Упражнение. Пишем веб-чат 243
15.3.1. Веб-чат: технические детали 243
15.3.2. Веб-чат: сервер 245
15.3.3. Веб-чат: клиент 248
15.4. Самостоятельное упражнение 257

Урок 16. WebRTC 258
16.1. Организация вещания посредством WebRTC 259
16.1.1. Создание соединения WebRTC 260
16.1.2. Указание вещаемого медиапотока 261
16.1.3. Обмен приглашением и согласием 262
16.1.3.1. Вызывающий: создание приглашения 262
16.1.3.2. Вызываемый: получение приглашения и создание согласия 262
16.1.3.3. Вызывающий: получение согласия 263

16.1.4. Обмен претендентами 264
16.1.4.1. Отправка претендента....264
16.1.4.2. Получение и регистрация претендента 265
16.1.5. Присоединение к вещанию 265
16.1.6. Завершение вещания 266
16.2. Упражнение. Пишем службу видеотелефона 267
16.2.1. Видеотелефон: технические детали 267
16.2.2. Видеотелефон: сервер 272
16.2.3. Видеотелефон: клиент....275
16.3. Обмен произвольными данными посредством WebRTC....289
16.3.1. Создание канала данных. Равноправный режим 290
16.3.2. Отправка данных 291
16.3.3. Получение данных....291
16.3.4. Закрытие канала данных 292
16.3.5. Режим «главный — подчиненный» 292
16.4. Самостоятельное упражнение 293

ЧАСТЬ VI. ПРОГРЕССИВНЫЕ ВЕБ-ПРИЛОЖЕНИЯ (PWA) 295

Урок 17. Введение в прогрессивные веб-приложения 297
17.1. Что такое прогрессивное веб-приложение (PWA)?....298
17.2. Основные принципы разработки PWA....299
17.3. Отладка PWA 300

Урок 18. Посредники и программируемый кэш 301
18.1. Посредники 301
18.1.1. Регистрация посредника 302
18.1.2. Перехват запроса и формирование ответа....304
18.1.2.1. Формирование произвольных ответов 304
18.1.2.2. Получение параметров запроса....305
18.1.3. Удаление посредника 305
18.2. Программируемый кэш....306
18.2.1. Получение доступа к области кэша 306
18.2.2. Сохранение файлов в области кэша....307
18.2.3. Извлечение файлов 309
18.2.3.1. Извлечение файлов из области кэша....309
18.2.3.2. Извлечение файлов из кэша 311
18.2.4. Удаление файлов из области кэша....312
18.2.5. Удаление области кэша....312
18.3. Упражнение. Добавляем посредник в клиент веб-чата....313
18.4. Задачи посредника....314
18.4.1. Регистрация задач посредника 315
18.4.2. Выполнение задач посредника 316
18.4.3. Удаление задач посредника 316
18.5. Отладочные инструменты для работы с посредниками и кэшем....316
18.5.1. Инструменты для работы с посредниками....317
18.5.2. Инструменты для работы с программируемым кэшем 318
18.5.3. Инструменты для работы с задачами посредника 320
18.5.4. Удаление данных PWA 322
18.6. Самостоятельное упражнение 323

Урок 19. Всплывающие оповещения....................324
19.1. Работа со всплывающими оповещениями....................324
19.1.1. Запрос разрешения на вывод всплывающих оповещений....................324
19.1.2. Вывод всплывающих оповещений....................325
19.1.2.1. Вывод всплывающих оповещений фронтендом....................325
19.1.2.2. Вывод всплывающих оповещений посредником....................326
19.1.3. Управление всплывающими оповещениями....................327
19.1.3.1. Управление всплывающими сообщениями во фронтендах....................327
19.1.3.2. Управление всплывающими сообщениями в посредниках....................328
19.1.4. Активация клиента....................329
19.1.4.1. Активация клиента в коде фронтенда....................329
19.1.4.2. Активация клиента в коде посредника....................329
19.1.5. Удаление всплывающих оповещений....................330
19.2. Упражнение. Реализуем вывод всплывающих оповещений у клиента веб-чата....................331
19.3. Самостоятельное упражнение....................333

Урок 20. Манифест PWA. Установка веб-приложений....................334
20.1. Манифест PWA....................334
20.1.1. Формат манифеста PWA....................334
20.1.2. Привязка манифеста к веб-странице PWA....................336
20.2. Установка PWA....................337
20.2.1. Установка PWA на мобильных платформах....................337
20.2.2. Установка PWA на настольных платформах....................337
20.3. Упражнение. Веб-чат: пишем манифест и реализуем установку....................339
20.4. Отладочные инструменты для работы с манифестом PWA....................342

Заключение....................345

Приложение. Описание электронного архива....................347

Предметный указатель....................348

Предисловие

Зачем нужна еще одна книга по JavaScript?

Что описывает новая книга?

Что вам понадобится в процессе чтения?

Типографские соглашения

В 2019 году автор написал книгу «JavaScript. 20 уроков для начинающих», рассказывающую о языке программирования JavaScript и его использовании для написания интерактивных веб-страниц. Судя по всему, книга была благосклонно принята читателями.

Сейчас, спустя 2 года, автор выпускает еще одну книгу по той же теме...

Зачем нужна еще одна книга по JavaScript?

Если вы, уважаемый читатель, хотите знать больше, программировать лучше — и получить конкурентное преимущество на рынке труда, — внимательно изучите эту книгу.

- Она содержит материал:
  - не вошедший в предыдущую книгу по причине ее ограниченного объема;
  - описывающий нововведения в JavaScript, которые появились уже после окончания работы над предыдущей книгой.
- Она рассказывает о новых впечатляющих программных инструментах, позволяющих:
  - наделить страницы, сайты и веб-приложения не доступными ранее возможностями (наподобие записи видео, съемки фото, чата или даже видеотелефона);
  - существенно упростить программирование и, соответственно, повысить продуктивность труда программиста.

Как и предыдущая книга, она:

- содержит 20 коротких, емких, наглядных уроков, дающих необходимые теоретические знания;
- включает:
  - более 20 практических упражнений, выполняемых под руководством автора. Делая их, вы закрепите полученные знания и приобретете программистский опыт, что безусловно оценит ваш будущий работодатель;

- 18 упражнений, рассчитанных на самостоятельное выполнение. Чтобы справиться с ними, достаточно внимательно изучать приведенный в уроках теоретический материал.

Что описывает новая книга?

- Нововведения в сам JavaScript: новый синтаксис объявления классов и функций, деструктурирование, промисы, асинхронные функции, итераторы, генераторы и модули.

  Все это существенно упрощает программирование и делает код более наглядным.

- HTML-компоненты.

  Отличное средство для создания с минимумом трудозатрат сложных элементов страниц, предназначенных для повторного использования: спойлеров, слайдеров, фотогалерей и т. п.

- СУБД, встроенную в веб-обозреватель.

  Она позволяет хранить практически любые данные в виде документов произвольной структуры, организуя их в коллекции. Может выполнять поиск нужного документа в коллекции по заданному критерию, выборку документов с возможностью фильтрации и сортировки.

- Инструменты для захвата видео со встроенной камеры...

  ...и записи его в видеофайл, который можно сохранить на локальном диске или отправить на сервер. Они также позволяют захватывать видео с экрана, записывать звук с микрофона и делать фото.

- Инструменты для наложения звуковых эффектов (например, панорамирования).
- Средства визуализации звука в виде осциллограммы и спектра.
- Сетевой протокол WebSocket.

  Реализующий двусторонний обмен данными с сервером с минимальными задержками. Хорошо подходит для создания чатов.

- Семейство технологий WebRTC.

  Позволяющих реализовать интенсивный обмен данными между клиентами без участия сервера. Применяются, в частности, при программировании служб интернет-телефонии.

- Прогрессивные веб-приложения (PWA).

  Дальнейшее развитие одностраничных сайтов — теперь они могут устанавливаться в операционной системе подобно обычным приложениям.

Книга содержит описание программных инструментов, поддерживаемых наиболее популярными веб-обозревателями из «большой тройки»: Google Chrome, Microsoft Edge Chromium и Mozilla Firefox (остальные веб-обозреватели имеют слишком незначительную долю рынка, чтобы принимать их в расчет).

> **Внимание!**
>
> В книге описываются также некоторые программные инструменты, поддерживаемые не всеми веб-обозревателями из «большой тройки» (обычно таких инструментов лишен Mozilla Firefox).

Сделано это по двум причинам. Во-первых, разработчику может понадобится задействовать подобного рода инструмент, по крайней мере, на поддерживающих его веб-обозревателях (в таком случае в качестве примера приводится фрагмент кода, корректно работающий на не поддерживающих его веб-обозревателях). Во-вторых, поддержка инструментов подобного рода может появиться в неподдерживающих веб-обозревателях в будущем.

Книгу сопровождает электронный архив (см. *приложение*), содержащий результаты выполнения всех упражнений в виде файлов с готовым программным кодом. Архив выложен на FTP-сервер издательства «БХВ» по адресу: **ftp://ftp.bhv.ru/9785977567817.zip**. Ссылка доступна и со страницы книги на сайте издательства: **https://bhv.ru/**.

Что вам понадобится?

Для изучения этой книги вам, уважаемый читатель, нужны следующие программы:

- *веб-обозреватель*, разумеется. Автор тестировал подготовленные им примеры в Google Chrome и Mozilla Firefox (Microsoft Edge Chromium полностью аналогичен Chrome);
- *текстовый редактор* — лучше какой-либо «программистский» — например, Visual Studio Code (**https://code.visualstudio.com/**);
- *пакет веб-хостинга XAMPP* (**https://www.apachefriends.org/ru/index.html**). Автор применял версию 8.0.2, включающую веб-сервер Apache HTTP Server 2.4.46 и PHP 8.0.2.

Еще от вас требуется:

- владеть языками HTML, CSS, безусловно JavaScript и PHP.

  Описание этих языков можно найти в книгах Владимира Дронова «HTML и CSS. 25 уроков для начинающих», «JavaScript. 20 уроков для начинающих», «PHP. 25 уроков для начинающих» или каких-либо других;
- иметь минимальные навыки веб-верстки;
- знать в общих чертах, как работает веб-сервер.

Типографские соглашения

В книге будут часто приводиться различные языковые конструкции, применяемые в программировании. Для наглядности при их написании использованы следующие типографские соглашения (в реальном коде они недействительны):

♦ HTML-, CSS-, JavaScript- и PHP-код набран моноширинным шрифтом:

```
<script type="text/javascript">
    const spls = Spoiler3.init({shown: true}, 'spl1', 'spl2');
</script>
```

♦ в угловые скобки <> заключены наименования различных значений, которые дополнительно выделены *курсивом*. В реальный код, разумеется, должны быть подставлены реальные значения. Например:

```
cancelAnimationFrame(<идентификатор задачи синхронного вывода>)
```

Здесь вместо подстроки *идентификатор задачи синхронного вывода* должен быть подставлен реальный идентификатор;

♦ в квадратные скобки `[]` заключены необязательные фрагменты кода:

```
fetch(<интернет-адрес>[, <параметры>])
```

Здесь *параметры* с предшествующей им запятой могут указываться, а могут и не указываться;

♦ у необязательных параметров функций и методов после знака равенства указано значение по умолчанию:

```
GainNode(<звуковой контекст>[, <параметры>=undefined])
```

Здесь у второго параметра конструктора класса `GainNode` значение по умолчанию — `undefined`;

♦ вертикальной чертой `|` разделены доступные для выбора значения, из которых в код можно подставить лишь одно:

```
respondWith(<ответ>|<файл>|<промис>)
```

Здесь можно подставить либо *ответ*, либо *файл*, либо *промис*;

♦ слишком длинные, не помещающиеся на одной строке книги фрагменты, разделены на несколько строк и в местах разрывов поставлены знаки ↳:

```
[<оператор объявления переменных>] ↳
[ <переменная 1>, <переменная 2> . . . <переменная n> ] = <массив>
```

Приведенный код здесь разбит на две строки, но должен быть набран в одну. Символ ↳ при этом нужно удалить;

♦ троеточием `. . .` помечены фрагменты кода, пропущенные ради сокращения объема текста книги:

```
class SomeClass {
    . . .
    get #privateProperty()      { . . . }
    set #privateProperty(value) { . . . }
}
```

Здесь весь код между первой и третьей строками пропущен.

Обычно такое можно встретить в исправленных впоследствии фрагментах кода — приведены лишь собственно исправленные строки, а оставшиеся неизмененными пропущены. Также троеточие используется, чтобы показать, в какое место должен быть вставлен вновь написанный код: в начало исходного фрагмента, в его конец или в середину, между уже присутствующими в нем строками;

- **полужирным шрифтом** выделен добавленный код:

```
const spl1 = document.getElementById('spl1');
const cSpl1 = new Spoiler(spl1);
cSpl1.shown = true;
```

Здесь в код было добавлено третье по счету выражение;

- ~~зачеркиванием~~ помечен удаленный код:

```
constructor(element, {shown = false, emphasized = false} = {}) {
    super(element);
    const shown = options.shown || false;
    const emphasized = options.emphasized || false;
    . . .
}
```

Здесь из кода были удалены третья и четвертая строка.

Полужирный шрифт и зачеркивание применяются только в коде практических упражнений.

> ***Внимание!***
>
> Все приведенные здесь типографские соглашения имеют смысл только в примерах написания языковых конструкций JavaScript и PHP.

ЧАСТЬ I
JavaScript: новые инструменты

- Новые инструменты для объявления классов и функций
- Деструктурирование
- Промисы
- Асинхронные функции
- Итераторы
- Генераторы
- Модули. Экспорт и импорт

Урок 1
Классы, функции и деструктурирование

Новый синтаксис объявления классов

Новые инструменты для объявления функций

Деструктурирование объектов и массивов

1.1. Новый синтаксис объявления классов

1.1.1. Старый синтаксис объявления классов и его недостатки

Ранее, чтобы объявить в JavaScript новый класс, было необходимо:

- объявить функцию-конструктор;
- указать у конструктора прототип, в котором создать свойства и методы, поддерживаемые объявляемым классом;
- создать динамические свойства класса, воспользовавшись методами `defineProperty()` и `defineProperties()` класса `Object`.

> *Динамическое свойство* — имитация обычного свойства, представляющее собой комбинацию геттера и сеттера.
>
> *Геттер* — метод, выполняющийся при попытке получить значение динамического свойства и возвращающий значение, которое и станет значением этого свойства. Геттер может брать возвращаемое значение из какого-либо обычного свойства или вычислять его на основе других значений.
>
> *Сеттер* — метод, выполняющийся при попытке занести новое значение в динамическое свойство и обычно сохраняющий это значение в каком-либо обычном свойстве. Сеттер также может выполнять какие-либо дополнительные действия.

В листинге 1.1 показан код класса `User`, объявленного в «старом» стиле. Класс поддерживает свойства `name`, `password`, `sendNotifications`, метод `isMature()` и динамическое свойство `age`.

Листинг 1.1. Код класса User, объявленного в «старом» стиле

```
// Функция-конструктор
function User(name, password, age) {
    this.name = name;
    this.password = password;
    this.__age = age;
}

// Объявление свойства sendNotifications и метода isMature() в прототипе
// создаваемого класса
User.prototype.sendNotifications = true;
User.prototype.isMature = function() {
    return this.age >= 18;
};

// Функции, реализующие геттер и сеттер для динамического свойства age
function User__getAge() {
    return this.__age;
}
function User__setAge(value) {
    this.__age = value;
}

// Объявление динамического свойства age
Object.defineProperty(User.prototype, 'age', {
    get: User__getAge,
    set: User__setAge
});
```

Такое объявление класса выглядит как набор не связанных друг с другом фрагментов кода. Вдобавок трудно понять, где начинается объявление класса и где оно заканчивается.

1.1.2. Новый синтаксис объявления классов

Всех этих недостатков лишен новый синтаксис объявления классов. Вот его формат:

```
class <имя объявляемого класса> {
    <объявления свойств>
    <объявление конструктора>
    <объявления методов>
    <объявления динамических свойств>
}
```

Отметим, что после любого из *объявлений* не ставятся ни запятая, ни точка с запятой.

- `Объявление свойства` — записывается в формате:

  ```
  <имя свойства>[ = <изначальное значение>]
  ```

  Если `изначальное значение` не указано, объявляемое свойство получит значение `undefined`.

  Присваивание `изначальных значений` объявляемым свойствам производится до выполнения конструктора. Так что в теле конструктора можно дать свойству другое значение (обычно так поступают со свойствами, чьи `изначальные значения` не указаны в их объявлениях).

- `Объявление конструктора` — записывается в формате:

  ```
  constructor([<перечень параметров через запятую>]) {
      <тело конструктора>
  }
  ```

- `Объявление метода` имеет похожий формат:

  ```
  <имя метода>([<перечень параметров через запятую>]) {
      <тело метода>
  }
  ```

- `Объявление динамического свойства` включает объявления его геттера и (или) сеттера.

  - Геттер объявляется в формате:

    ```
    get <имя динамического свойства>() {
        <тело геттера>
    }
    ```

    Геттер не должен принимать параметров и обязан возвращать результат (который и станет значением динамического свойства).

  - Сеттер объявляется в формате:

    ```
    set <имя динамического свойства>(<параметр>) {
        <тело сеттера>
    }
    ```

    Сеттер должен принимать `параметр` — новое значение динамического свойства. Этот `параметр` может иметь произвольное имя, но чаще всего ему дают имя `value`. Возвращать результат сеттер не должен.

  Динамическое свойство с геттером и сеттером доступно и для чтения, и для записи. Если в динамическом свойстве объявлен только геттер, свойство доступно лишь для чтения, если объявлен лишь сеттер — только для записи.

Порядок следования `объявлений свойств`, `конструктора`, `методов` и `динамических свойств` может быть произвольным. Однако традиционно сначала помещают `объявления свойств`, потом — `конструктора`, за ним — `методов` и `динамических свойств`.

В листинге 1.2 представлен код класса `User`, объявленного с применением нового синтаксиса.

Листинг 1.2. Код класса `User`, объявленного с помощью нового синтаксиса

```
class User {
    name
    password
    __age
    sendNotifications = true

    constructor(name, password, age) {
        this.name = name;
        this.password = password;
        this.__age = age;
    }

    isMature() {
        return this.age >= 18;
    }

    get age() {
        return this.__age;
    }
    set age(value) {
        this.__age = value;
    }
}
```

Такой код несравнимо нагляднее и выглядит более целостным, нежели представленный в листинге 1.1.

Если значения каких-либо свойств задаются в конструкторе, объявлять их непосредственно в классе необязательно. Например, в классе `User` значения свойств `name`, `password` и `__age` задаются в конструкторе, поэтому их объявления можно убрать (в приведенном далее листинге они зачеркнуты):

```
class User {
    name
    password
    __age
    sendNotifications = true
    constructor(name, password, age) {
        this.name = name;
        this.password = password;
        this.__age = age;
    }
   . . .
}
```

Тем не менее в классе рекомендуется указывать объявления *всех* свойств — для наглядности.

1.1.3. Закрытые свойства и методы

Ранее все свойства и методы, создаваемые в объявляемом классе, являлись общедоступными.

> *Общедоступными* называются свойства и методы, доступные везде: в текущем классе, производных классах (о них — чуть позже) и извне текущего класса.

Вследствие этого любой внешний по отношению к объекту код мог обратиться к любому свойству или вызвать любой метод этого объекта. Что было совершенно неприемлемо в случае, например, если какое-либо свойство хранило значение, изменение которого «извне» могло нарушить правильное функционирование объекта.

В новых версиях JavaScript можно пометить свойства и методы, доступ к которым извне нежелателен, как закрытые.

> *Закрытыми* называются свойства и методы, доступные *только в текущем классе*. При попытке получить доступ к закрытому свойству или методу извне возникнет ошибка `SyntaxError`.

Чтобы сделать свойство или метод закрытым, достаточно предварить его имя символом «решетки» (`#`). Этот символ необходимо указывать не только при объявлении свойства (метода) но и при обращении к нему (т. е. «решетка» становится неотъемлемой частью имени свойства или метода).

> **Внимание!**
>
> Закрытые свойства и методы в настоящее время не поддерживаются Mozilla Firefox[1].

Пример объявления в классе `User2` закрытых свойства `password` и метода `getPassword()`:

```
class User2 {
    name
    #password
    . . .
    #getPassword() {
        return this.#password;
    }
    about() {
        return this.name + ', ' + this.#password;
    }
}
```

Проверим, сможем ли мы получить доступ к закрытым свойству и методу:

```
// Создаем объект класса User2
let u = new User2('someuser', '123456');
```

[1] Описание программных инструментов, не поддерживаемых какими-либо веб-обозревателями, в книге приводится для справки и на будущее, когда эта поддержка появится (см. *предисловие*).

```
// Обращаемся к общедоступному свойству name — успешно
console.log(u.name);                        // Результат: someuser
// Обращаемся к закрытым свойству password и методу getPassword() —
// безуспешно
console.log(u.#password);                   // Ошибка SyntaxError
console.log(u.#getPassword());              // Ошибка SyntaxError
// Однако мы можем вызвать общедоступный метод about(), который
// без проблем обращается к закрытому свойству password,
// поскольку объявлен в том же классе
console.log(u.about());                     // Результат: someuser, 123456
```

Возможно создание закрытых динамических свойств. Для этого достаточно пометить геттер и сеттер такого свойства как закрытые, предварив их имена символом «решетки». Пример объявления закрытого динамического свойства `privateProperty`:

```
class SomeClass {
    . . .
    get #privateProperty()      { . . . }
    set #privateProperty(value) { . . . }
}
```

1.1.4. Статические свойства и методы

В старых версиях JavaScript статические свойства и методы приходилось создавать непосредственно у функции-конструктора, что выглядело весьма странно.

> *Статическими* называются свойства и методы, принадлежащие не объекту какого-либо класса, а самому этому классу.

В новых версиях JavaScript для создания статического свойства или метода достаточно при его объявлении поставить перед именем `static`.

Доступ к статическому свойству или методу осуществляется:

- внутри класса — через переменную `this`;
- вне класса — через имя класса.

Пример объявления класса `User2` со статическими свойством `url` и методом `getFullURL()`:

```
class User2 {
    . . .
    static url = 'www.somesite.ru/account/login/';
    . . .
    static getFullURL() {
        return 'https://' + this.url;
    }
}
```

Попробуем обратиться к статическим свойству и методу:

```
console.log(User2.url);      // Результат: www.somesite.ru/account/login/
console.log(User2.getFullURL());
                             // Результат: https://www.somesite.ru/account/login/
```

Можно объявлять закрытые статические свойства и методы описанным ранее способом: предварив имя символом «решетки». Пример:

```
class User2 {
    . . .
    static #url = 'www.somesite.ru/account/login/';
    . . .
    static getFullURL() {
        return 'https://' + this.#url;
    }
}
```

1.1.5. Наследование классов

В старых версиях JavaScript наследование классов реализовывалось путем манипуляций с прототипами, что выглядело странно и не отличалось наглядностью.

> *Наследование* — создание одного класса на основе другого. Наследующий класс (*производный*, или *подкласс*) получает все *общедоступные* свойства, методы и динамические свойства того класса, от которого он наследован (*базового класса*, или *суперкласса*). Закрытые свойства, методы и динамические свойства не наследуются. Также производный класс может объявлять свои собственные свойства, методы и динамические свойства.

В новых версиях JavaScript объявление производного класса выполняется гораздо проще и нагляднее — с помощью следующего синтаксиса:

```
class <имя производного класса> extends <имя базового класса> {
    <объявления свойств производного класса>
    <объявление конструктора производного класса>
    <объявления методов производного класса>
    <объявления динамических свойств производного класса>
}
```

В производном классе можно объявлять свойства, методы и динамические свойства с именами, совпадающими с аналогичными сущностями базового класса. В этом случае производный класс получит то же свойство (метод, динамическое свойство), что и базовый класс, но имеющее другое значение (другую функциональность).

> *Перекрытие* — замена в производном классе свойства (метода, динамического свойства), унаследованного от базового класса, путем его повторного объявления.

Однако в случае методов гораздо чаще производится не перекрытие, а переопределение.

> *Переопределение* — дополнение и (или) изменение функциональности метода, унаследованного от базового класса, в производном классе.

При переопределении в объявлении метода производного класса записывается код, реализующий дополнительную функциональность, а в нужном месте его вставляется вызов метода базового класса. Для этого применяется следующий синтаксис:

- для вызова конструктора базового класса:

  ```
  super([<перечень параметров через запятую>])
  ```

- для вызова метода, принадлежащего базовому классу, с указанным *именем*:

  ```
  super.<имя вызываемого метода>([<перечень параметров через запятую>])
  ```

В листинге 1.3 приведен код класса `User3`, производного от класса `User` (см. листинг 1.2). В новом классе добавлено свойство `gender`, перекрыто свойство `sendNotifications`, переопределены конструктор и метод `isMature()`.

Листинг 1.3. Код класса `User3`

```
class User3 extends User {
    gender
    sendNotifications = false

    constructor(name, password, age, gender) {
        super(name, password, age);
        this.gender = gender;
    }

    isMature() {
        return super.isMature() ? 'Совершеннолетний(яя)' :
                                  'Несовершеннолетний(яя)';
    }
}
```

Проверим новый класс:

```
let u1 = new User('Вася Пупкин', '123456', 50);
console.log(u1.sendNotifications);   // Результат: true
console.log(u1.isMature());          // Результат: true
let u2 = new User3('Вера Машкина', '123456', 17, 'Ж');
console.log(u2.sendNotifications);   // Результат: false
console.log(u2.isMature());          // Результат: Несовершеннолетний(яя)
```

1.1.6. Анонимные классы

Новые версии JavaScript позволяют объявлять *анонимные классы*, не имеющие имен. Анонимные классы объявляются так же, как обычные, только без указания имени.

Объявление такого класса следует присвоить какой-либо переменной (или свойству объекта). В противном случае объявление класса будет тотчас удалено из памяти и, таким образом, утеряно.

Пример объявления двух анонимных классов, причем второй является производным от первого:

```
const User = class { . . . }
const User3 = class extends User { . . . }
```

Анонимные классы можно использовать так же, как и обычные, *именованные*. Для создания объекта анонимного класса следует использовать оператор `new` с именем переменной, которой было присвоено объявление класса. Пример:

```
let u1 = new User('Вася Пупкин', '654321', 50);
console.log(u1.isMature());          // Результат: true
let u2 = new User3('Вера Машкина', '111111', 17, 'Ж');
console.log(u2.isMature());          // Результат: Несовершеннолетний(яя)
```

> **ПОЛЕЗНО ЗНАТЬ...**
>
> Объявление класса, выполненное в новом синтаксисе, при выполнении кода веб-обозревателем неявно преобразуется в объявление старого синтаксиса. В результате получается обычная функция-конструктор, представленная объектом класса `Function`. Кстати, именно этот объект присваивается переменной при объявлении анонимного класса (см. *разд. 1.1.6*).

Это сделано для совместимости с ранее написанными веб-сценариями, которые зачем-либо манипулируют ранее объявленными классами.

1.2. Упражнение. Создаем веб-компонент — спойлер

Напишем спойлер — вероятно, один из наиболее часто используемых веб-компонентов. Применим для этого новый синтаксис объявления классов.

> *Спойлер* — панель, свертывающаяся и развертывающаяся при последовательных щелчках на ее заголовке.
>
> *Веб-компонент* — класс, реализующий функциональность сложного элемента страницы (наподобие спойлера), предназначенный для повторного использования (в том числе и в других проектах) и при необходимости конструирующий необходимые вспомогательные элементы самостоятельно.

Класс спойлера будет иметь имя `Spoiler`, поддерживать общедоступное динамическое свойство `shown` (`true`, если спойлер развернут, `false` — если свернут) и общедоступный метод `toggle()` (изменяет состояние спойлера с развернутое на свернутое и наоборот). Конструктор класса в качестве единственного параметра будет принимать ссылку на базовый элемент.

> *Базовый элемент веб-страницы* — элемент, на основе которого создается экземпляр компонента.

В качестве базового может выступать блочный элемент, созданный с применением любого тега. Внутри него должны находиться следующие элементы:

- гиперссылка (тег `<a>`) со стилевым классом `header` — заголовок спойлера;
- блочный элемент (тег не имеет значения) — разворачиваемое и сворачиваемое содержимое спойлера.

1. Найдем в папке 1\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы index.html (веб-страница с двумя заготовками для спойлеров), styles.css (таблица стилей с оформлением самой страницы) и spoiler.css (таблица стилей с оформлением самого спойлера). Скопируем их куда-либо на локальный диск.
2. Откроем в текстовом редакторе копию страницы index.html и добавим в секцию заголовка (тег `<head>`) тег `<script>`, привязывающий к странице веб-сценарий spoiler.js, в котором будет записан код класса спойлера и который мы создадим чуть позже:

```
<html>
    <head>
        . . .
        <link href="spoiler.css" rel="stylesheet" type="text/css">
        <script src="spoiler.js" type="text/javascript"></script>
    </head>
    . . .
</html>
```

3. Создадим файл spoiler.js в той же папке, в которой хранится файл index.html и все остальные файлы примера, и откроем его в текстовом редакторе.

   Нужно объявить класс спойлера `Spoiler` и в его конструкторе сделать необходимые приготовления: привязку стилевого класса `spoiler` (все необходимые стили записаны в таблице стилей spoiler.css) к базовому элементу и привязку обработчика события `click` к гиперссылке заголовка спойлера. Также в классе необходимо объявить закрытое свойство `element`, которое будет хранить ссылку на базовый элемент.
4. Запишем в файл spoiler.js объявление класса спойлера `Spoiler` со свойством `element` и конструктором:

```
class Spoiler {
    element

    constructor(element) {
        this.element = element;
        element.classList.add('spoiler');
        const a = element.querySelector('a.header');
        a.compSpoiler = this;
        a.addEventListener('click', Spoiler__headerClick);
    }
}
```

Здесь мы сохраняем полученный с параметром базовый элемент в свойстве element. Затем добавляем к базовому элементу стилевой класс spoiler. Все классы, представляющие элементы страницы, поддерживают свойство classList, хранящее коллекцию привязанных к элементу стилевых классов. Метод add() этой коллекции привязывает указанный в его параметре стилевой класс к элементу.

Далее ищем находящуюся в полученном элементе гиперссылку со стилевым классом header, создаем в представляющим ее объекте свойство compSpoiler, заносим в это свойство ссылку на текущий объект компонента (он понадобится позже) и привязываем к гиперссылке обработчик события click — функцию Spoiler__headerClick(), которую сейчас напишем.

5. Объявим функцию Spoiler__headerClick() — обработчик события click гиперссылки-заголовка спойлера. Поскольку это будет обычная функция, запишем ее код вне объявления класса Spoiler, перед ним:

```
function Spoiler__headerClick(evt) {
    evt.preventDefault();
    evt.target.compSpoiler.toggle();
}

class Spoiler {
    . . .
}
```

Обязательно отменяем обработчик события по умолчанию (чтобы не вызвать несанкционированный переход по гиперссылке), для чего вызываем у объекта события метод preventDefault(). После чего из свойства target объекта события получаем источник события (элемент, в котором возникло событие), т. е. гиперссылку, обращаемся к ранее созданному в ней свойству compSpoiler, чтобы получить объект спойлера, и вызываем у этого объекта метод toggle(), который переключит состояние спойлера и который мы сейчас напишем.

6. Объявим в классе Spoiler метод toggle(), переключающий состояние спойлера (с развернутое на свернутое и наоборот):

```
class Spoiler {
    . . .
    toggle() {
        this.element.classList.toggle('shown');
    }
}
```

Метод toggle() коллекции стилевых классов привязывает к элементу страницы заданный стилевой класс, если он еще не привязан, и убирает — в противном случае.

7. Объявим в классе Spoiler геттер динамического свойства shown:

```
class Spoiler {
    . . .
```

```
    get shown() {
        return this.element.classList.contains('shown');
    }
}
```

Метод `contains()` коллекции стилевых классов возвращает `true`, если указанный стилевой класс привязан к элементу, и `false` — в противном случае.

8. Объявим в классе `Spoiler` сеттер динамического свойства `shown`:

```
class Spoiler {
    . . .
    set shown(value) {
        if (value) {
            if (!this.shown)
                this.toggle();
        } else {
            if (this.shown)
                this.toggle();
        }
    }
}
```

Если выполняется развертывание спойлера (свойству `shown` присваивается значение `true`), который сейчас свернут (текущее значение этого свойства — `false`), вызываем метод `toggle()` спойлера, чтобы развернуть его. Если же выполняется свертывание спойлера (свойству `shown` присваивается значение `false`), который сейчас развернут (текущее значение этого свойства — `true`), также вызываем метод `toggle()` спойлера, что свернет его.

Осталось проверить только что написанный компонент в работе. Страница index.html содержит две «заготовки» для спойлеров, имеющих якоря `spl1` и `spl2`.

> *Якорь* — уникальная пометка элемента страницы, задаваемая у его тега в атрибуте `id`. По этой пометке в веб-сценарии можно получить объект, представляющий помеченный элемент, для манипулирования им.

9. Откроем в текстовом редакторе копию страницы index.html и добавим в самый конец HTML-кода тег `<script>` с веб-сценарием, который инициализирует оба спойлера:

```
<html>
    . . .
</html>
<script type="text/javascript">
    const spl1 = document.getElementById('spl1');
    const cSpl1 = new Spoiler(spl1);
    const spl2 = document.getElementById('spl2');
    const cSpl2 = new Spoiler(spl2);
</script>
```

ВНИМАНИЕ!

Не забываем сохранять все созданные и исправленные файлы перед их открытием в веб-обозревателе! Автор более не будет напоминать об этом.

Откроем копию страницы index.html в каком-либо веб-обозревателе (автор использовал Google Chrome). Щелкнем на заголовке первого спойлера, чтобы развернуть его (рис. 1.1).

[-] JavaScript

Мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262).

Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

[+] TypeScript

Рис. 1.1. Веб-страница с двумя спойлерами, левый развернут

Теперь сделаем так, чтобы первый спойлер развертывался сразу после открытия страницы.

10. Добавим в сценарий, вставленный в страницу index.html, выражение, которое развернет первый спойлер:

```
<script type="text/javascript">
    const spl1 = document.getElementById('spl1');
    const cSpl1 = new Spoiler(spl1);
    cSpl1.shown = true;
    . . .
</script>
```

Для развертывания спойлера достаточно присвоить свойству `shown` представляющего его объекта значение `true`.

Обновим открытую в веб-обозревателе страницу и убедимся, что левый спойлер развертывается изначально.

1.3. Упражнение. Совершенствуем спойлер

Усовершенствуем созданный в *упражнении 1.2* спойлер.

- Добавим возможность указания у спойлера двух настроек:
  - `shown` — если `true`, спойлер будет изначально развернут, если `false` (значение по умолчанию) — свернут;
  - `emphasized` — если `true`, заголовок спойлера будет отображаться белым текстом на черном фоне (станет выделенным), если `false` (значение по умолчанию) — в обычной цветовой гамме.

  Настройки будут передаваться конструктору класса компонента со вторым, необязательным для указания параметром. Они будут задаваться в виде служебного объекта, содержащего свойства, имена которых совпадают с именами параметров.
- Добавим событие `ontoggle`, возникающее после переключения состояния спойлера.

Класс усовершенствованного спойлера назовем `Spoiler2` и сделаем его производным от класса `Spoiler`. Чтобы сделать заголовок спойлера выделенным, привяжем к базовому элементу стилевой класс `em` (соответствующий стиль уже хранится в таблице стилей spoiler.css). Событие `ontoggle` реализуем в виде одноименного свойства, которому будет присваиваться функция-обработчик события.

1. Найдем в папке 1\ex1.2 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css, spoiler.css, spoiler.js и скопируем их куда-либо на локальный диск.
2. Откроем в текстовом редакторе копию страницы index.html и добавим в секцию заголовка (тег `<head>`) тег `<script>`, привязывающий к странице веб-сценарий spoiler2.js, в котором будет записан код усовершенствованного спойлера и который мы создадим чуть позже:

```
<html>
    <head>
        . . .
        <script src="spoiler.js" type="text/javascript"></script>
        <script src="spoiler2.js" type="text/javascript"></script>
    </head>
    . . .
</html>
```

3. Создадим файл spoiler2.js в той же папке, в которой хранится файл index.html и все остальные файлы примера, и откроем его в текстовом редакторе.
4. Запишем в файл spoiler2.js объявление класса `Spoiler2`, производного от `Spoiler`, и сразу же объявим в нем свойство `ontoggle`:

```
class Spoiler2 extends Spoiler {
    ontoggle
}
```

Теперь нужно переопределить конструктор класса, записав в него код, который прочитает значения настроек, переданные в служебном объекте вторым параметром, и соответствующим образом настроит спойлер.

5. Добавим в объявление класса код переопределенного конструктора:

```
class Spoiler2 extends Spoiler {
    . . .
    constructor(element, options = {}) {
        super(element);
        const shown = options.shown || false;
        const emphasized = options.emphasized || false;
        this.shown = shown;
        if (emphasized)
            element.classList.add('em');
    }
}
```

Сначала вызываем конструктор базового класса, использовав языковую конструкцию `super()`, иначе спойлер не будет работать.

Для чтения настроек из служебного объекта воспользуемся особенностью оператора `||` (логическое ИЛИ) JavaScript. Операнды, не относящиеся к логическому типу, он преобразуют в логический тип. Если в результате преобразования первого операнда получается `true`, оператор `||` возвращает значение первого операнда в изначальном, не преобразованным в логическую величину виде. Если же получается `false`, возвращается непреобразованное значение второго операнда.

В нашем случае, если в служебном объекте присутствует свойство `shown`, в одноименную константу будет занесено значение этого свойства. Если же в служебном объекте такого свойства нет, константа `shown` получит значение `false` (значение одноименного параметра по умолчанию). Точно так же получается значение параметра `emphasized`.

Далее присваиваем значение параметра `shown` одноименному свойству спойлера — если этот параметр имеет значение `true`, спойлер будет развернут. Если параметр `emphasized` хранит значение `true`, привязываем к базовому элементу стилевой класс `em`.

Осталось переопределить метод `toggle()`, добавив в него функциональность вызова обработчика события `ontoggle`.

6. Добавим в объявление класса переопределенный метод `toggle()`:

```
class Spoiler2 extends Spoiler {
    . . .
    toggle() {
        super.toggle();
        if (this.ontoggle instanceof Function)
            this.ontoggle();
    }
}
```

Сначала вызываем метод `toggle()`, унаследованный от базового класса, далее проверяем, является ли значение свойства `ontoggle` объектом класса `Function` (т. е. функцией), и, если это так, вызываем эту функцию.

Проверим усовершенствованный спойлер в действии. Сделаем так, чтобы первый спойлер на странице index.html был изначально развернутым, а второй имел выделенный заголовок. Кроме этого, сделаем так, чтобы при переключении состояния второго спойлера в консоли веб-обозревателя выводилось сообщение о разворачивании или сворачивании спойлера.

7. Исправим уже присутствующий в файле index.html код, инициализирующий оба спойлера, следующим образом:

```
<script type="text/javascript">
    const spl1 = document.getElementById('spl1');
    const cSpl1 = new Spoiler2(spl1, {shown: true});
    const spl2 = document.getElementById('spl2');
    const cSpl2 = new Spoiler2(spl2, {emphasized: true});
    cSpl2.ontoggle = () => {
        if (cSpl2.shown)
            console.log('Спойлер развернут.');
        else
            console.log('Спойлер свернут.');
    };
</script>
```

[-] JavaScript

Мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262).

Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

[+] TypeScript

Рис. 1.2. Веб-страница с двумя усовершенствованными спойлерами

Откроем страницу index.html в веб-обозревателе (рис. 1.2). Развернем и свернем второй спойлер. В консоли должны появиться одна за другой строки «Спойлер развернут.» и «Спойлер свернут.».

1.4. Новые инструменты для работы с функциями

1.4.1. Функции с произвольным количеством параметров. Оператор упаковки-распаковки

Начиная с самых первых версий JavaScript, в функциях доступна коллекция `arguments`, хранящая значения всех переданных функции параметров. Благодаря этому можно создавать функции, принимающие произвольное количество параметров. Пример такой функции:

```
function processValues() {
    let par;
    for (let i = 0; i < arguments.length; i++) {
        // Получаем очередной параметр, переданный функции
        par = arguments[i];
        // Обрабатываем его
        . . .
    }
}
```

Однако, взглянув на такое объявление, невозможно сразу понять, сколько параметров принимает функция: произвольное количество или вообще ни одного.

В новых версиях языка достаточно записать все обязательные параметры, принимаемые функцией, после них поставить еще один, последний параметр и указать перед ним оператор упаковки-распаковки.

> *Оператор упаковки-распаковки* `...<параметр>` предписывает объединить («упаковать») все оставшиеся параметры, переданные функции при вызове, в массив и присвоить этот массив указанному *параметру*.

ВНИМАНИЕ!

Оператор упаковки-распаковки должен указываться у самого последнего параметра в объявлении функции.

Будучи указанным *в объявлении функции*, этот оператор выполняет *упаковку* значений параметров (также он может выполнить распаковку, о которой мы поговорим позже).

Пример функции с двумя обязательными параметрами и произвольным количеством необязательных:

```
function processValues(p1, p2, ...pars) {
    console.log(p1, p2, pars);
}
```

```
// Параметр p1 получит значение 1, параметр p2 — значение 2,
// а параметр pars — массив [3, 4, 5]
processValues(1, 2, 3, 4, 5);                // Результат: 1 2 [3, 4, 5]
// Параметр pars получит массив из одного элемента — [3]
processValues(1, 2, 3);                      // Результат: 1 2 [3]
// Параметр pars получит «пустой» массив
processValues(1, 2);                         // Результат: 1 2 []
```

1.4.2. Использование оператора упаковки-распаковки в вызовах функций

Оператор упаковки-распаковки можно использовать и в вызовах функций — с применением формата `...<массив>`. В таком случае он извлекает («распаковывает») из заданного *массива* элементы и подставляет их в вызов функции в качестве параметров в том порядке, в котором эти элементы присутствуют в *массиве*.

Будучи указанным *в вызове функции*, этот оператор выполняет *распаковку* параметров (поэтому и носит название оператора упаковки-распаковки).

Пример:

```
function someFunc(x, y, z) {
    . . .
}

let arr = [1, 2, 3];
someFunc(...arr);
// Того же результата можно добиться, записав:
// someFunc(arr[0], arr[1], arr[2]);
```

1.5. Упражнение. Реализуем массовую инициализацию спойлеров

Если на странице присутствуют несколько спойлеров, достаточно утомительно для инициализации каждого из них писать выражения вида:

```
const spln = document.getElementById('spln');
const cSpln = new Spoiler2(spln);
```

Объявим еще один класс спойлера, производный от класса `Spoiler2`, назвав его `Spoiler3`, и добавим в него статический метод `init()`. Этот метод будет принимать первым параметром служебный объект с настройками всех спойлеров, а остальными параметрами (которым может быть произвольное количество) — якоря базовых элементов. В качестве результата метод будет возвращать массив созданных спойлеров.

1. Найдем в папке 1\ex1.3 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css, spoiler.css, spoiler.js, spoiler2.js и скопируем их куда-либо на локальный диск.

2. Откроем в текстовом редакторе копию страницы index.html и добавим в секцию заголовка (тег `<head>`) тег `<script>`, привязывающий к странице веб-сценарий spoiler3.js, в котором будет записан код нового спойлера и который мы создадим чуть позже:

```
<html>
    <head>
        . . .
        <script src="spoiler2.js" type="text/javascript"></script>
        <script src="spoiler3.js" type="text/javascript"></script>
    </head>
    . . .
</html>
```

3. Создадим файл spoiler3.js в той же папке, в которой хранится файл index.html и все остальные файлы примера, и откроем его в текстовом редакторе.

4. Запишем в файл spoiler3.js объявление класса `Spoiler3`:

```
class Spoiler3 extends Spoiler2 {
    static init(options, ...anchors) {
        let el, spoilers = [], sp;
        for (let i = 0; i < anchors.length; i++) {
            el = document.getElementById(anchors[i]);
            if (el) {
                sp = new Spoiler3(el, options);
                spoilers.push(sp);
            }
        }
        return spoilers;
    }
}
```

   В объявлении метода `init()` мы указали у последнего параметра `anchors` оператор упаковки-распаковки. В результате все якоря базовых элементов, перечисленные в вызове этого метода вторым и последующими параметрами, будут «упакованы» в массив `anchors`, из которого мы сможем извлечь их для обработки.

   Проверим новый компонент, сделав так, чтобы все спойлеры на странице выводились изначально развернутыми.

5. Исправим уже присутствующий в файле index.html код, инициализирующий спойлеры, следующим образом:

```
<script type="text/javascript">
    const spls = Spoiler3.init({shown: true}, 'spl1', 'spl2');
</script>
```

   Откроем страницу index.html в веб-обозревателе и посмотрим на результат. Как видим, мы инициализировали все спойлеры и задали у них одинаковые параметры, записав всего одно выражение.

1.6. Деструктурирование

Ранее, чтобы присвоить значения разных элементов массива разным переменным, приходилось записывать набор выражений, каждое из которых выполняло присваивание очередного элемента одной из переменных. То же самое касалось и свойств объектов.

В новых версиях JavaScript такие операции можно выполнить в одном выражении — посредством деструктурирования.

> *Деструктурирование* (или *деструктурирующее присваивание*) — присваивание значений отдельных элементов массива (*деструктурирование массива*) или отдельных свойств объекта (*деструктурирование объекта*) разным переменным в одном выражении.

1.6.1. Деструктурирование массивов

Выражение деструктурирования массива записывается в формате:

```
[<оператор объявления переменных>] ↳
[ <переменная 1>, <переменная 2> . . . <переменная n> ] = <массив>
```

Переменной 1 будет присвоен первый элемент указанного *массива*, *переменной 2* — второй элемент и т. д. Если *переменные* не были объявлены ранее, следует указать какой-либо *оператор объявления переменных*: `let`, `const` или `var`.

В качестве *переменной* можно указать:

- имя переменной:

  ```
  const arr = [1, 2, 3, 4, 5];
  let [a, b, c, d, e] = arr;
  console.log(a, b, c, d, e);            // Результат: 1 2 3 4 5
  ```

  В выражении деструктурирования можно указать переменных меньше, чем элементов в массиве, — тогда «лишние» элементы останутся неприсвоенными:

  ```
  let [x, y, z] = arr;
  console.log(x, y, z);                  // Результат: 1 2 3
  ```

  Можно не указывать какую-либо переменную внутри перечня, поставив, однако, запятую, — тогда соответствующий элемент массива останется неприсвоенным:

  ```
  [a, , b, , c] = arr;
  console.log(a, b, c);                  // Результат: 1 3 5
  ```

  Еще можно указать переменных больше, чем элементов в массиве, — тогда лишние переменные получат значение `undefined`:

  ```
  [a, b, c, d, e, x, y] = arr;
  console.log(a, b, c, d, e, x, y);
                              // Результат: 1 2 3 4 5 undefined undefined
  ```

- конструкцию формата:

  ```
  <имя переменной> = <значение по умолчанию>
  ```

  В этом случае, если переменной с указанным *именем* не «хватит» элемента массива, она получит заданное *значение по умолчанию*:

  ```
  [a, b, c, d, e, x = 10, y = 100] = arr;
  console.log(a, b, c, d, e, x, y);       // Результат: 1 2 3 4 5 10 100
  ```

- конструкцию формата `...<переменная>` — тогда заданной *переменной* будет присвоен массив со всеми «лишними» элементами деструктурируемого массива. Такая конструкция ставится в самом конце перечня переменных. Пример:

  ```
  let [x, y, ...z] = arr;
  console.log(x, y, z);                   // Результат: 1 2 [3, 4, 5]
  ```

Деструктурирование массивов часто применяются для перестановки местами значений двух переменных:

```
a = 'HTML';
b = 'CSS';
[b, a] = [a, b];
console.log(a, b);                        // Результат: CSS HTML
```

1.6.2. Деструктурирование объектов

Выражение деструктурирования объекта записывается в формате:

```
[<оператор объявления переменных>] ↳
{ <переменная 1>, <переменная 2> . . . <переменная n> } = <объект>
```

Если *переменные* не были объявлены ранее, следует указать какой-либо *оператор объявления переменных*: `let`, `const` или `var`. Если же все *переменные* были объявлены ранее, все выражение следует взять в круглые скобки.

В качестве *переменной* можно указать:

- имя переменной — тогда этой переменной будет присвоено значение одноименного свойства указанного *объекта*:

  ```
  const obj = {name1: 'Vasya', name2: 'Pupkin', age: 50};
  let {name1, name2} = obj;
  console.log(name1, name2);              // Результат: Vasya Pupkin
  ```

  Переменная, чье имя не совпадает с именами свойств объекта, получит значение `undefined`:

  ```
  let gender;
  // Поскольку все три переменные были объявлены ранее, выражение
  // деструктурирования объекта нужно взять в круглые скобки
  ({name1, name2, gender} = obj);
  console.log(name1, name2, gender);// Результат: Vasya Pupkin undefined
  ```

- конструкцию формата:

  ```
  <имя свойства объекта>: <имя переменной>
  ```

  В таком случае значение свойства с указанным *именем* будет присвоено переменной с заданным *именем*. Это позволяет присваивать значения свойств переменным с произвольно выбранными именами. Пример:

  ```
  let {name1: firstName, name2: lastName} = obj;
  console.log(firstName, lastName);       // Результат: Vasya Pupkin
  ```

- конструкцию формата:

  ```
  <имя переменной> = <значение по умолчанию>
  ```

  Если в заданном объекте отсутствует свойство с именем, совпадающим с заданным *именем переменной*, переменная получит указанное *значение по умолчанию*:

  ```
  ({name1, name2, gender = 'M'} = obj);
  console.log(name1, name2, gender);      // Результат: Vasya Pupkin M
  ```

- конструкцию формата:

  ```
  <имя свойства объекта>: <имя переменной> = <значение по умолчанию>
  ```

  Она является комбинацией двух описанных ранее конструкций. Пример:

  ```
  ({name1: firstName, name2: lastName, gender: gnd = 'M'} = obj);
  console.log(firstName, lastName, gnd);  // Результат: Vasya Pupkin M
  ```

- конструкцию формата `...<переменная>` — тогда заданной *переменной* будет присвоен служебный объект, содержащий все «лишние» свойства деструктурируемого объекта. Такая конструкция ставится в самом конце перечня переменных. Пример:

  ```
  let o;
  ({name2, ...o} = obj);
  console.log(name2, o);  // Результат: Pupkin {name1: "Vasya", age: 50}
  ```

Деструктурирование объекта часто применяется при объявлении функций и методов, принимающих в качестве одного из параметров конфигурационный объект с какими-либо настройками. Пример будет рассмотрен в следующем упражнении.

1.7. Упражнение. Упрощение кода спойлера *Spoiler2*

Конструктор класса `Spoiler2` включает код, извлекающий из полученного конфигурационного объекта два поддерживаемых компонентом параметра и, если какой-либо параметр не указан, дающий ему значение по умолчанию. Этот код можно исключить, использовав в объявлении конструктора деструктурирование конфигурационного объекта.

1. Найдем в папке 1\ex1.5 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css, spoiler.css, spoiler.js, spoiler2.js, spoiler3.js и скопируем их куда-либо на локальный диск.

2. Откроем в текстовом редакторе копию файла spoiler2.js, в котором хранится код класса `Spoiler2`, и заменим в объявлении конструктора второй параметр следующим выражение деструктурирования объекта:

   ```
   class Spoiler2 extends Spoiler {
       . . .
       constructor(element, {shown = false, emphasized = false} = {}) {
           . . .
       }
       . . .
   }
   ```

   Это выражение деструктурирует полученный со вторым параметром конфигурационный объект, разнося значения его свойств по разным локальным переменным.

3. Удалим из кода конструктора более не нужные выражения, извлекающие настройки из конфигурационного объекта:

   ```
   class Spoiler2 extends Spoiler {
       . . .
       constructor(element, {shown = false, emphasized = false} = {}) {
           super(element);
           const shown = options.shown || false;
           const emphasized = options.emphasized || false;
           . . .
       }
       . . .
   }
   ```

   Напоследок проверим компонент в работе.

1.8. Самостоятельное упражнение

Еще раз усовершенствуйте компонент спойлера. Объявите класс `Spoiler4`, производный от класса `Spoiler3`, сохранив его в файле spoiler4.js, и добавьте в него статический метод `init2()`. Этот метод должен принимать произвольное количество якорей базовых элементов и создавать на их основе спойлеры, причем первый спойлер должен быть изначально развернутым, а остальные — свернутыми. Возвращать метод должен массив созданных спойлеров.

Урок 2
Промисы и асинхронные функции

Промисы
Асинхронные функции
Оператор ожидания

2.1. Промисы

Почти все операции, выполняемые веб-сценариями, являются *синхронными* — приостанавливающими выполнение сценария до тех пор, пока выполнение операции не завершится. К синхронным операциям относятся создание переменной, объявление функции, класса, привязка обработчика к событию и др.

Загрузка данных с сервера с применением технологии AJAX — типичный пример *асинхронной* операции, при выполнении которой исполнение сценария не приостанавливается. После загрузки файла в объекте загрузчика возникает событие `readystatechange`, в обработчике которого можно получить содержимое загруженного файла.

Загрузка данных может занять неопределенное, и иногда весьма продолжительное время. Чтобы на время загрузки веб-обозреватель не «повисал», эта операция и реализована в виде асинхронной.

2.1.1. Старые приемы выполнения асинхронных операций и их недостатки

В предыдущих версиях JavaScript код, асинхронно загружающий данные, не отличался наглядностью. Рассмотрим пример:

```
// 1) Выполняем какие-либо подготовительные операции
. . .
const ajax = new XMLHttpRequest();
ajax.addEventListener('readystatechange', function () {
    if (this.readyState == 4 && this.status == 200) {
        const resp = this.responseText;
        // 3) Обрабатываем загруженные данные
        . . .
    }
});
```

```
// 2) Запускаем загрузку данных
ajax.open('GET', '/programs/getdata.php', true);
ajax.send();
```

По идее, сначала следует выполнить подготовительные действия (пункт **1**), потом загрузить данные (пункт **2**) и, наконец, обработать их (пункт **3**). Но, согласно приведенному примеру, данные, еще не загруженные, сначала обрабатываются, а уже потом загружаются. Код выглядит непоследовательным и может обескуражить неопытного программиста.

Можно попытаться решить проблему, вынеся код, загружающий и обрабатывающий данные, в отдельную функцию. Например:

```
// Первым параметром передается HTTP-метод, вторым — интернет-адрес,
// третьим — функция, обрабатывающая полученные данные
function loadAndProcessData(method, url, dataHandler) {
    const ajax = new XMLHttpRequest();
    ajax.addEventListener('readystatechange', function () {
        if (this.readyState == 4 && this.status == 200)
            dataHandler(this.responseText);
    });
    ajax.open(method, url, true);
    ajax.send();
}
. . .
loadAndProcessData('GET', '/programs/getdata.php', (resp) => {
    // Обрабатываем полученные данные
});
```

Однако подобного рода код, выполняющий сразу несколько действий, тоже не очень нагляден, и трудно понять, в каком месте данные загружаются, а в каком — обрабатываются. Сопровождать такой код нелегко.

2.1.2. Промисы. Создание промисов

Новые версии JavaScript предоставляют хорошее средство сделать код, выполняющий асинхронную операцию, легче для восприятия — промисы.

> *Промис* — сущность, представляющая произвольное значение, которое будет получено в результате выполнения асинхронной операции (например, загружено с сервера спустя неопределенный промежуток времени).
>
> *Нагрузка* — значение, представляемое промисом. Может быть произвольного типа.

Промис может находиться в одном из трех состояний:

- *ожидания* — когда нагрузка еще не получена (например, асинхронная загрузка данных с сервера запущена, но данные еще не получены);
- *подтвержденном* — когда нагрузка получена, может быть извлечена и использована в вычислениях (данные с сервера успешно загружены);

♦ *отклоненном* — когда в процессе получения нагрузки возникла нештатная ситуация (данные получить не удалось из-за проблем с сетью или сервером).

Алгоритм выполнения асинхронной операции, применяющий промисы, в общих чертах таков:

1. Запускается выполнение асинхронной операции (например, загрузки файла) и выдается промис, представляющий будущий результат выполнения этой операции и пока что находящийся в состоянии ожидания.
2. Когда выполнение асинхронной операции успешно завершается, выданный промис подтверждается, и в него в качестве нагрузки заносится результат выполнения операции (например, содержимое загруженного файла).
3. Если в процессе выполнения операции возникла ошибка, промис отклоняется.

Промис представляется объектом класса `Promise`. Конструктор этого класса имеет следующий формат вызова:

`Promise(<функция, реализующая логику промиса>)`

`Функция, реализующая логику промиса,` которая передается конструктору, собственно, и реализует асинхронную операцию, результатом выполнения которой будет нагрузка промиса. `Функция` не должна возвращать результат и обязана принимать два параметра: подтверждающую и отклоняющую функции.

> *Подтверждающая функция* — вызывается функцией, реализующей логику промиса, *при успешном получении нагрузки*. Вызов этой функции подтверждает промис. Подтверждающей функции при вызове передается один параметр — сама нагрузка.
>
> *Отклоняющая функция* — вызывается функцией, реализующей логику промиса, если при получении нагрузки возникла какая-либо проблема. Вызов этой функции отклоняет промис. Отклоняющей функции при вызове передается один параметр — объект отклоняющего исключения.
>
> *Отклоняющее исключение* — исключение, генерируемое в теле функции, которая реализует логику промиса. Представляет собой объект одного из классов исключений, поддерживаемых JavaScript, обычно класса `Error`.

При создании объекта отклоняющего исключения конструктору его класса передается какое-либо значение, описывающее возникшую проблему (например, интернет-адрес загружаемого файла).

Пример функции, загружающей данные с сервера, которая в качестве результата возвращает промис:

```
function loadAndProcessData(method, url) {
    return new Promise((resolve, reject) => {
        const ajax = new XMLHttpRequest();
        ajax.addEventListener('readystatechange', function () {
            if (this.readyState == 4 && this.status == 200)
```

```
                // Если данные были загружены, вызываем подтверждающую
                // функцию, передав ей загруженные данные
                resolve(this.responseText);
            else
                // В противном случае вызываем отклоняющую функцию,
                // передав ей объект исключения с интернет-адресом
                reject(new Error(url));
        });
        ajax.open(method, url, true);
        ajax.send();
    });
}
```

Вместо готового объекта исключения отклоняющей функции можно передать любое другое значение. В этом случае сам промис создаст исключение в виде объекта класса `Error`, передав конструктору класса значение, полученное отклоняющей функцией. Пример:

```
function loadAndProcessData(method, url) {
    return new Promise((resolve, reject) => {
        . . .
        // Передаем отклоняющей функции непосредственно интернет-адрес.
        // Объект исключения будет создан автоматически, самим промисом.
        reject(url);
        . . .
    });
}
```

НА ЗАМЕТКУ

Автор рассматривает работу с промисами на примере загрузки файлов с помощью класса `XMPHttpRequest`. По мнению автора, это лучший способ изучать промисы.

Однако современные версии JavaScript предоставляют новый, более удобный инструмент для загрузки файлов с применением AJAX (кстати, тоже использующий промисы). Он будет рассмотрен в *главе 6*.

2.1.3. Обработка промисов

Для обработки полученного промиса применяются следующие три метода, поддерживаемые классом `Promise`:

- `then(<функция then>[, <функция catch>])` — задает у текущего промиса *функции then* и *catch*.

    Функция then — выполняется при подтверждении текущего промиса. Должна принимать в единственном параметре нагрузку. В теле функции then выполняется обработка полученной нагрузки.

    Функция catch — выполняется при отклонении текущего промиса. Должна принимать в единственном параметре объект отклоняющего исключения.

> В теле функции catch каким-либо образом обрабатывается нештатная ситуация, возникшая при получении нагрузки промиса (например, выводится соответствующее сообщение).

Любая из заданных *функций* может возвращать результат одного из следующих типов:

- другой промис, созданный в теле *функции*, — который будет возвращен методом `then()` без изменений (и может быть обработан в последующем вызове того же метода);
- любое значение, не являющееся промисом, — метод `then()` заключит его в другой, вновь созданный промис, сразу же пометит его как подтвержденный и вернет в качестве результата.
- Если ни одна из *функций* не вернет результат, — метод `then()` вернет вновь созданный промис, хранящий значение `undefined` и уже помеченный как подтвержденный.
- Если в теле какой-либо из *функций* возникнет ошибка, — метод `then()` вернет промис, хранящий объект исключения и уже помеченный как отклоненный.

Если *функция catch* не задана, отклоняющее исключение будет передано обработчику по умолчанию, встроенному в веб-обозреватель. Обработчик по умолчанию просто выводит сообщение о возникшей ошибке в консоли;

- `catch(<функция catch>)` — задает у текущего промиса *функцию catch*.

Внимание!

Для указания функции catch удобнее использовать метод `catch()` (а не задавать ее вторым параметром в методе `then()`). Благодаря этому код станет нагляднее.

- `finally(<функция finally>)` — задает у текущего промиса *функцию finally*.

> *Функция finally* — выполняется после исполнения функции then или функции catch. Не должна принимать параметров и возвращать результат. В ее теле обычно производятся какие-либо завершающие действия (например, скрытие индикатора загрузки данных).

Пример, иллюстрирующий применение приведенных ранее методов:

```
const prm = loadAndProcessData('GET', '/programs/getdata.php');
prm.then((data) => {
        // Промис подтвержден, значит, данные успешно приняты.
        // В параметре получаем нагрузку — принятые данные.
        const decodedData = JSON.parse(data);
        // Обрабатываем полученные данные
    })
   .catch((exception) => {
        // Промис отклонен, значит, данные загрузить не удалось.
        // В параметре получаем объект отклоняющего исключения.
        const errMsg = exception.message;
```

```
        // Выводим сообщение об ошибке
    })
    .finally(() => {
        // Промис подтвержден или отклонен.
        // Функции then и catch выполнены.
        // Производим завершающие действия.
    });
```

Такой код более нагляден, чем представленный в *разд. 2.1.1*. Сразу видно, где выполняется загрузка данных, где обрабатываются загруженные данные, а где — возникшая нештатная ситуация.

Поскольку методы `then()` и `catch()` могут возвращать результаты в виде промисов, есть возможность разделить обработку полученных данных или возникших исключений на части, записав несколько вызовов этих методов друг за другом. Так программный код станет еще нагляднее и проще в сопровождении. Пример:

```
pmm.then((data) => {
        // Декодируем полученные данные и возвращаем их
        return JSON.parse(data);
        // Метод then() «обернет» возвращаемые данные в уже
        // подтвержденный промис, который будет обработан следующим
        // вызовом того же метода
    })
    .then((decodedData) => {
        // Получаем JSON-данные, декодированные в предыдущем вызове
        // метода then(), и пускаем их в обработку
    })
    .catch((exception) => {
        // Генерируем другое исключение — с развернутым сообщением
        // об ошибке
        throw new Error('Не удалось загрузить файл ' +
                        exception.message;)
        // Метод catch() вернет промис, содержащий сгенерированное
        // исключение и уже отклоненный, который будет обработан в
        // следующем вызове того же метода
    })
    .catch((error2) => {
        // Получаем объект исключения, сгенерированного в предыдущем
        // вызове метода catch(), и выводим сообщение об ошибке
        console.log(error2.message);
    });
```

2.1.4. Массовая обработка промисов

Часто требуется обработать в один и тот же момент сразу несколько промисов (что может возникнуть, например, при одновременной загрузке нескольких файлов). Причем здесь могут реализовываться разные стратегии: «каждый пришел к фини-

шу» (дождаться, когда каждый из обрабатываемых промисов будет подтвержден), «каждый пришел к финишу или сошел с дистанции» (дождаться, когда каждый из обрабатываемых промисов будет либо подтвержден, либо отклонен) и др.

В этом могут помочь статические методы класса `Promise`, рассмотренные далее:

- `all(<`*`массив промисов`*`>)` — реализует стратегию «каждый пришел к финишу».

  Возвращает промис, который:

  - подтверждается — когда все промисы из заданного *массива* «придут к финишу» (будут подтверждены). Функции then возвращаемого промиса передается в параметре массив нагрузок промисов;
  - отклоняется — если хотя бы один промис «сойдет с дистанции» (будет отклонен). Функции catch возвращаемого промиса передается в параметре отклоняющее исключение из «сошедшего с дистанции» промиса.

  Пример использования этого метода будет рассмотрен в *упражнении 2.3*;

- `allSettled(<`*`массив промисов`*`>)` — реализует стратегию «каждый пришел к финишу или сошел с дистанции».

  Возвращает промис, который подтверждается, когда каждый промис из указанного *массива* будет либо подтвержден, либо отклонен. Функции then возвращаемого промиса передается в параметре массив служебных объектов, содержащих следующие свойства:

  - `status` — если `'fulfilled'`, соответствующий промис подтвержден, если `'rejected'` — отклонен;
  - `value` — присутствует, только если соответствующий промис подтвержден, и хранит представляемое им значение;
  - `reason` — присутствует, только если промис отклонен, и хранит отклоняющее исключение.

  Пример:

```
const prm1 = loadAndProcessData('GET', '/files/1.html');
const prm2 = loadAndProcessData('GET', '/files/2.html');
const prm3 = loadAndProcessData('GET', '/files/3.html');
Promise.allSettled([prm1, prm2, prm3])
       .then((arr) => {
            let obj, data;
            for (let i = 0; i < arr.length; i++) {
                obj = arr[i];
                if (obj.status == 'fulfilled') {
                    data = obj.value;
                    // Обрабатываем загруженный файл
                } else {
                    // Выводим сообщение об ошибке загрузки
                }
            }
        });
```

- any(<*массив промисов*>) — реализует стратегию «один пришел к финишу».

  Возвращает промис, который:

  - подтверждается — как только какой-либо промис из заданного *массива* «придет к финишу» (подтвердится). Функции then возвращаемого промиса передается в параметре нагрузка подтвержденного промиса.

    Все промисы, «пришедшие к финишу» позже, игнорируются;

  - отклоняется — если все заданные промисы «сойдут с дистанции» (отклонятся). Самостоятельно генерирует отклоняющее исключение AggregateError, производное от класса Error. Объект этого исключения имеет свойство errors, хранящее массив с отклоняющими исключениями из отдельных промисов.

  Пример:

```
Promise.any([prm1, prm2, prm3])
       .then((data) => {
            // Обрабатываем файл, который загрузился раньше всех
        })
       .catch((exception) => {
            if (exception.name == 'AggregateError')
                console.log('Ни один файл не был загружен.');
        });
```

- race(<*массив промисов*>) — реализует стратегию «один пришел к финишу или сошел с дистанции».

  Возвращает промис, который:

  - подтверждается — как только какой-либо промис из заданного *массива* будет подтвержден. Функции then возвращаемого промиса передается в параметре нагрузка подтвержденного промиса;
  - отклоняется — если какой-либо промис из заданного *массива* отклонится. Функции catch возвращаемого промиса передается в параметре отклоняющее исключение.

2.1.5. Дополнительные инструменты для работы с промисами

При работе с промисами могут пригодиться два следующих статических метода, поддерживаемые классом Promise:

- resolve(<*нагрузка*>) — возвращает уже подтвержденный промис, хранящий заданную *нагрузку*.

  Этот метод может пригодиться, если какие-либо данные в зависимости от некоторого условия либо загружаются с сервера, либо вычисляются самим веб-сценарием, и для обработки этих данных желательно использовать один и тот же код. Вот пример:

```
let prm;
if (loadDataFromServer)
    prm = loadAndProcessData('GET', '/programs/getdata.php');
else
    prm = Promise.resolve(computedData);
prm.then( . . . ).catch( . . . );
```

- `reject(<отклоняющее исключение>)` — возвращает уже отклоненный промис с указанным *отклоняющим исключением*.

Класс `Window` получил поддержку двух новых событий:

- `rejectionhandled` — возникает после успешной обработки отклоненного промиса заданной у него функцией catch. Невсплывающее и не имеет обработчика по умолчанию.

  Сведения об этом событии представляются объектом класса `PromiseRejectionEvent` (производного от класса `Event`). Этот класс поддерживает два свойства (помимо унаследованных):

  - `reason` — отклоняющее исключение;
  - `promise` — объект отклоненного промиса.

  Пример использования этого события при отладке:

```
window.addEventListener('rejectionhandled', (evt) => {
    console.log(evt.reason);
});
```

- `unhandledrejection` — возникает при отклонении промиса, если у него не была указана функция catch. Имеет обработчик по умолчанию, выводящий сообщение об отклоняющем исключении в консоли веб-обозревателя. Если отменить обработчик по умолчанию, сообщение выводиться не будет:

```
window.addEventListener('unhandledrejection', (evt) => {
    evt.preventDefault();
});
```

  В остальном аналогично событию `rejectionhandled`.

2.2. Упражнение. Загрузка файлов посредством AJAX

Реализуем загрузку двух HTML-файлов и вывод их содержимого на странице, применив промисы.

Внимание!

Для тестирования результата выполнения этого упражнения понадобится какой-либо веб-сервер. Автор использовал популярный пакет хостинга XAMPP (**https://www.**

apachefriends.org/ru/index.html). Описание установки и использования этого пакета можно найти в книге Владимира Дронова «JavaScript. 20 уроков для начинающих»[1].

1. Найдем в папке 2\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы 2.2.html (веб-страница с двумя элементами, в которых будет выводиться содержимое файлов), 2.2_1.html, 2.2_2.html (загружаемые HTML-файлы) и 2.2.css (таблица стилей с оформлением страницы). Скопируем их куда-либо на локальный диск.
2. Создадим в папке, где хранятся копии этих файлов, файл веб-сценария dataloader.js.

   В этом файле будет храниться код функции `loadData()`, загружающей указанный файл. Она будет принимать в качестве параметра интернет-адрес загружаемого файла и возвращать промис, подтверждаемый при успешной загрузке и отклоняемый при возникновении ошибки.
3. Откроем в текстовом редакторе только что созданный файл dataloader.js и запишем в него код объявления функции `loadData()`:

```
function loadData(url) {
    return new Promise((resolve, reject) => {
        const ajax = new XMLHttpRequest();
        ajax.addEventListener('readystatechange', function () {
            if (this.readyState == 4)
                if (this.status == 200)
                    resolve(this.responseText);
                else
                    reject(new Error(url + ': ошибка ' + this.status));
        });
        ajax.open('get', url, true);
        ajax.send();
    });
}
```

4. Откроем в текстовом редакторе копию файла 2.2.html и добавим в секцию заголовка тег `<script>`, привязывающий веб-сценарий dataloader.js:

```
<html>
    <head>
        . . .
        <link href="2.2.css" rel="stylesheet" type="text/css">
        <script src="dataloader.js" type="text/javascript"></script>
    </head>
    . . .
</html>
```

5. Добавим в конец кода страницы 2.2.html тег `<script>` со сценарием, получающим доступ к элементам `output1` и `output2`, в которых будет выводиться содержимое загруженных файлов 2.2_1.html и 2.2_2.html соответственно:

---

[1] См. **https://bhv.ru/product/javascript-20-urokov-dlya-nachinayushhih/**.

```
<html>
    . . .
</html>
<script type="text/javascript">
    const output1 = document.getElementById('output1');
    const output2 = document.getElementById('output2');
</script>
```

6. Добавим в только что созданный тег `<script>` код, который, используя функцию `dataLoad()`, загрузит файл 2.2_1.html, в случае успеха выведет его содержимое в элементе `output1`, а в случае неудачи — отобразит там же сообщение об ошибке:

```
<script type="text/javascript">
    const output1 = document.getElementById('output1');
    const output2 = document.getElementById('output2');
    loadData('2.2_1.html')
        .then((data) => {
            output1.innerHTML = data;
        })
        .catch((exception) => {
            output1.innerHTML = exception.message;
        });
</script>
```

7. Добавим самостоятельно аналогичный код, загружающий и выводящий файл 2.2_2.html.

Опробуем готовый код в работе. Скопируем файлы 2.2.html, 2.2_1.html, 2.2_2.html, 2.2.css и dataloader.js в корневую папку веб-сервера, запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/2.2.html**. Мы увидим то, что показано на рис. 2.1.

JavaScript	TypeScript
Мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262). Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.	Язык программирования, представленный Microsoft в 2012 году и позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript. Разработчиком языка TypeScript является Андерс Хейлсберг (англ. Anders Hejlsberg), создавший ранее Turbo Pascal, Delphi и C#.

Рис. 2.1. Веб-страница с содержимым двух загруженных HTML-файлов

Проверим, как работает обработка ошибок загрузки. Переименуем файл 2.2_2.html в 2.2_3.html и обновим страницу. Мы должны увидеть сообщение об ошибке (рис. 2.2).

2.2_2.html: ошибка 404

JavaScript

Мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262).

Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

Рис. 2.2. Загрузка одного из HTML-файлов завершилась неудачей

2.3. Упражнение. Массовая загрузка файлов

Создадим страницу, которая будет выводить содержимое трех HTML-файлов, загружаемых с сервера. Причем файлы должны загрузиться все (стратегия «каждый пришел к финишу») — в противном случае на странице ничего не появится. Для реализации применим массовую обработку промисов.

1. Найдем в папке 2\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы 2.3.html (веб-страница с тремя элементами, в которых будет выводиться содержимое файлов), 2.2_1.html, 2.2_2.html, 2.3_1.html (загружаемые HTML-файлы) и 2.3.css (таблица стилей с оформлением страницы). Скопируем их куда-либо на локальный диск.
2. Найдем в папке 2\ex.2.2 файл dataloader.js (написанная в *упражнении 2.2* функция для загрузки файлов) и скопируем его туда же, где хранятся копии упомянутых ранее файлов.
3. Откроем в текстовом редакторе копию страницы 2.3.html и добавим в секцию заголовка тег `<script>`, привязывающий веб-сценарий dataloader.js:

```
<html>
    <head>
        . . .
```

```
        <link href="2.3.css" rel="stylesheet" type="text/css">
        <script src="dataloader.js" type="text/javascript"></script>
    </head>
    . . .
</html>
```

4. Добавим в конец кода страницы 2.3.html тег <script> со сценарием, получающим доступ к элементам со стилевыми классами output, в которых будет выводиться содержимое загруженных файлов:

```
<html>
    . . .
</html>
<script type="text/javascript">
    const cOutput = document.body.querySelectorAll('.output');
</script>
```

5. Добавим в только что созданный тег <script> код, создающий массив промисов, которые представляют содержимое загружаемых файлов:

```
<script type="text/javascript">
    const cOutput = document.body.querySelectorAll('.output');
    const prms = [
        loadData('2.2_1.html'),
        loadData('2.2_2.html'),
        loadData('2.3_1.html')
    ];
</script>
```

6. Добавим в тег <script> код, выводящий содержимое загруженных файлов в элементы со стилевым классом output, а если загрузка какого-либо из файлов не увенчалась успехом, — сообщение об ошибке в первом элементе:

```
<script type="text/javascript">
    . . .
    Promise.all(prms)
            .then((arr) => {
                for (let i = 0; i < arr.length; i++)
                    cOutput[i].innerHTML = arr[i];
             })
            .catch((exception) => {
                cOutput[0].innerHTML = exception.message;
            });
</script>
```

Скопируем файлы 2.3.html, 2.2_1.html, 2.2_2.html, 2.3_1.html, 2.3.css и dataloader.js в корневую папку веб-сервера, запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/2.3.html**. Мы увидим то, что показано на рис. 2.3.

JavaScript	TypeScript	Python
Мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262). Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.	Язык программирования, представленный Microsoft в 2012 году и позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript. Разработчиком языка TypeScript является Андерс Хейлсберг (англ. Anders Hejlsberg), создавший ранее Turbo Pascal, Delphi и C#.	Высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой набор полезных функций.

Рис. 2.3. Веб-страница с содержимым трех загруженных HTML-файлов

2.4. Асинхронные функции и оператор ожидания

Код функции, выполняющей асинхронную операцию и возвращающей промис, весьма сложен (см. пример из *разд. 2.1.2*). Поэтому в более поздних версиях JavaScript была введена поддержка асинхронных функций, имеющих более простой и привычный по обыкновенным функциям синтаксис.

> *Асинхронная функция* — функция, выполняющая асинхронную операцию и использующая для выдачи результата обычный оператор `return`.

Фактически асинхронная функция сразу после запуска в качестве результата возвращает автоматически созданный промис. Явный возврат в теле этой функции оператором `return` результата вызывает подтверждение этого промиса с занесением в него этого результата в качестве нагрузки, а генерирование любого исключения — отклонение промиса. Можно сказать, что асинхронная функция — более наглядный аналог обычной функции, возвращающей промис.

Формат объявления асинхронной функции:

```
async function <имя функции>([<перечень параметров через запятую>]) {
    <тело функции>
}
```

Рассмотрим в сравнении обычную функцию, возвращающую промис, и полностью эквивалентную ей асинхронную функцию:

Обычная функция, возвращающая промис	Асинхронная функция

`function someFunc() {`
`    return new Promise((res, rej) => {`
`            . . .`
`            if (success)`
`                res(value);`
`            else`
`                rej(new Error());`
`        });`
`}`
	`async function someFunc() {`
	`    . . .`
	`    if (success)`
	`        return value;`
	`    else`
	`        throw new Error();`
	`}`

Видно, что код асинхронной функции компактнее и намного нагляднее.

> ***Внимание!***
>
> Результат, выдаваемый асинхронной функцией, должен вычисляться непосредственно в теле этой функции.

Поэтому переделать функцию `dataLoad()` (см. *упражнение 2.2*), в которой результат (загружаемый файл) получается в другой функции — обработчике события `readystatechange` объекта загрузчика, не выйдет (а даже если так и сделать, результат будет потерян).

Промис, неявно возвращенный асинхронной функцией, может быть обработан описанным в *разд. 2.1* способом — посредством указания функций then и catch. Однако удобнее применять для этого оператор ожидания `await`, введенный в JavaScript одновременно с асинхронными функциями.

> *Оператор ожидания* `await <промис>` — приостанавливает исполнение кода и ждет того момента, когда указанный *промис* будет подтвержден или отклонен. Если промис подтвержден, оператор возвращает его нагрузку. Если промис отклонен, оператор возбуждает исключение, вызвавшее отклонение промиса.

> ***Внимание!***
>
> Оператор ожидания `await` допускается применять лишь в асинхронных функциях.

Рассмотрим два способа обработки промисов: посредством функций `then` и `catch` и с помощью оператора ожидания — в сравнении:

Функции `then` и `catch`	Оператор ожидания `await`

`someFunc()`
`    .then((data) => {`
`        output.textContent = data;`
`    })`
`    .catch((exc) => {`
`        console.log(exc.message);`
`    });`
	`async function func() {`
	`    try {`
	`        output.textContent = await someFunc();`
	`    }`
	`    catch(exception) {`
	`        console.log(exc.message);`
	`    }`
	`}`
	`func();`

Код, использующий оператор ожидания, чуть больше, но заметно нагляднее. К тому же, во многих случаях его можно значительно сократить, использовав асинхронное замыкание и убрав обработку исключений:

```
(async function () {
    output.textContent = await someFunc();
})();
```

Разумеется, посредством оператора ожидания можно обрабатывать и промисы, возвращаемые обычными функциями.

К сожалению, массовую обработку промисов (описывалась в *разд. 2.1.4*) выполнить посредством операторов ожидания нельзя.

2.4.1. Асинхронные методы в классах

В классах JavaScript можно объявлять асинхронные методы. Для этого достаточно записать перед именем метода через пробел языковую конструкцию `async`. В объявлении статического асинхронного метода конструкция `async` ставится сразу после конструкцией `static`. Пример:

```
class SomeClass {
    . . .
    // Общедоступный асинхронный метод
    async method1() { . . . }
    // Общедоступный статический асинхронный метод
    static async staticMethod1() { . . . }
    // Закрытый асинхронный метод
    async #method2() { . . . }
    // Закрытый статический асинхронный метод
    static async #staticMethod2() { . . . }
}
```

Асинхронные методы вызываются так же, как и асинхронные функции:

```
const sc = new SomeClass();
(async function () {
    await sc.method1();
    await SomeClass.staticMethod1();
}){};
```

2.5. Самостоятельные упражнения

- Переделайте результат выполнения *упражнения 2.3* таким образом, чтобы он реализовывал стратегию «каждый пришел к финишу или сошел с дистанции». Сообщения об ошибках загрузки отдельных файлов пусть выводятся в соответствующих элементах со стилевыми классами `output`.
- Переделайте результат выполнения *упражнения 2.2* с использованием оператора ожидания.

Урок 3
Итераторы и генераторы

Цикл перебора последовательности

Итераторы

Генераторы

Асинхронные итераторы и генераторы

Цикл перебора с ожиданием

3.1. Цикл перебора последовательности

В новых версиях JavaScript появилась новая разновидность цикла — цикл перебора последовательности.

> *Цикл перебора последовательности* — перебирает один за другим элементы из заданной последовательности, присваивая очередной элемент заданной переменной, которая будет доступна в теле цикла.
>
> *Последовательность* — объект JavaScript, способный выдавать одно за другим какие-либо значения, хранящиеся в нем или вычисленные в процессе работы, и пригодный для применения в цикле перебора последовательности.

В JavaScript последовательностями являются массивы, строки, коллекции `arguments` и `NodeList` (однако коллекция `HTMLCollection` не является последовательностью).

Цикл перебора последовательности записывается в следующем формате:

```
for ([<оператор объявления>] <переменная> of <последовательность>) {
    <тело цикла>
}
```

Очередной элемент указанной *последовательности* будет присваиваться заданной *переменной*. Если *переменная* ранее не была объявлена, следует поставить какой-либо оператор объявления переменной: `let`, `const` или `var`.

Пример перебора массива:

```
const arr = ['HTML', 'CSS', 'JavaScript'];
for (let el of arr)
    console.log(el);                        // Результат: HTML
                                            //            CSS
                                            //            JavaScript
```

Пример перебора строки:

```
const str = 'Google Chrome';
let output = '';
for (let chr of str)
    output = chr + output;
console.log(output);                        // Результат: emorhC elgooG
```

Цикл перебора последовательности имеет более компактный и наглядный синтаксис, чем применяемый для этой же цели цикл со счетчиком. К тому же он может применяться для обработки объектов, оснащенных итераторами.

3.2. Итераторы

Любой класс можно наделить функциональностью последовательности. В результате объект такого класса будет выдавать одно за другим вычисленные в нем значения, которые могут быть обработаны в цикле перебора последовательности (см. *разд. 3.1*).

Чтобы превратить класс в последовательность, достаточно объявить в нем итератор.

> *Итератор* — часть функциональности класса, наделяющая объекты этого класса функциональностью последовательности.

Итератор представляет собой обычный метод, который:

- не должен принимать параметров;
- должен иметь имя, совпадающее со значением статического свойства `iterator` класса `Symbol`. При объявлении этого метода его имя следует взять в квадратные скобки;
- должен возвращать объект итератора, представляющий собой обычный служебный объект с не принимающим параметров методом `next()`. Метод `next()` должен возвращать служебный объект со следующими свойствами:
    - `done` — `false`, если в текущий момент выдается одно из вычисленных значений (даже если это значение — последнее), `true` — если значений больше нет;
    - `value` — очередное вычисленное значение. Если значений больше нет, не указывается.

Пример класса с итератором:

```
class SomeClass {
    . . .
    [Symbol.iterator]() {
        return {
            next: function() {
```

```
                // Вычисляем очередное значение для выдачи или выясняем,
                // не пора ли прекратить выдачу значений.
                . . .
                // Есть ли очередное значение для выдачи?
                if (isValueExists)
                    // Если есть, выдаем его
                    return { value: computedValue, done: false };
                else
                    // Если нет, сообщаем, что значения закончились
                    return { done: true };
            }
        };
    }
}
```

Пример использования объектов этого класса:

```
const obj = new SomeClass();
for (let val of obj) {
    // Обрабатываем выдаваемые объектом значения
}
```

Можно объявить функцию, возвращающую итератор. Пример такой функции:

```
function someFunc() {
    return {
        next: function() {
            . . .
            if (isValueExists)
                return { value: computedValue, done: false };
            else
                return { done: true };
        }
    };
}
```

Использование такой функции:

```
for (val of fomeFunc()) {
    // Обрабатываем выдаваемые функцией значения
}
```

Полезно знать...

- **Класс `Symbol` служит для создания так называемых *символов* JavaScript — уникальных значений особого типа. Они обычно используются для объявления в объектах свойств с гарантированно уникальными именами. Пример:**

  ```
  const obj = {};
  // Создаем два уникальных символа
  const sym1 = new Symbol();
  const sym2 = new Symbol();
  ```

```
// Пользуясь символами, создаем в объекте два свойства с уникальными
// именами
obj[sym1] = 123;
obj[sym2] = '321';
. . .
// Пользуемся символами для обращения к этим свойствам
let n = obj[sym1];
let s = obj[sym2];
```

Сфера применения символов очень узка, поэтому в настоящей книге они не описываются.

- Статическое свойство `iterator` класса `Symbol` хранит предопределенный символ, используемый для объявления итераторов в классах.

3.3. Упражнение. Вычисляем числа Фибоначчи

Напишем веб-приложение, вычисляющее и выводящее на экран последовательность из заданного количества чисел Фибоначчи.

> *Числа Фибоначчи* — элементы числовой последовательности, в которой первые два числа равны, соответственно, 0 и 1, а каждое из последующих — сумме двух предыдущих чисел.

Для вычисления чисел объявим класс `Fibonacci`, содержащий итератор.

1. Найдем в папке 3\!sources сопровождающего книгу электронного архива (см. *приложение*) файл template.html (страница, на основе которой будет создано веб-приложение), скопируем его куда-либо на локальный диск и дадим копии имя 3.3.html.

   Страница включает абзац с якорем `output`, в котором будет выведен результат. А в конце ее HTML-кода находится пустой тег `<script>`, в котором будет записан программный код.

2. Откроем в текстовом редакторе только что созданный файл 3.3.html и запишем в теге `<script>` первую часть кода, объявляющего класс `Fibonacci`:

```
<script type="text/javascript">
    class Fibonacci {
        count

        constructor(count) {
            this.count = count;
        }
    }
</script>
```

   В классе `Fibonacci` мы объявили свойство `count`, хранящее длину вычисляемой последовательности чисел, и конструктор, который получит в параметре требуемую длину последовательности и сохранит ее в свойстве `count`.

Теперь объявим метод-итератор, вычисляющий числа. В нем создадим следующие переменные: `i` (счетчик уже вычисленных чисел), `current` (текущее число, выдаваемое в настоящий момент), `prev` (предыдущее вычисленное число) и `pprev` («пред-предыдущее» число).

Еще необходимо объявить переменную `cnt` для хранения количества вычисляемых чисел. Ведь внутри объекта, возвращаемого итератором, мы не сможем обратиться к этому свойству, поскольку переменная `this` в возвращаемом из итератора объекте указывает на сам возвращаемый объект, а не на «внешний» по отношению к нему объект класса `Fibonacci`.

3. Добавим в объявление класса `Fibonacci` код итератора с объявлением необходимых переменных и выражение, возвращающее служебный объект с пока что «пустым» методом `next()`:

```
class Fibonacci {
    . . .
    [Symbol.iterator]() {
        const cnt = this.count;
        let i = 0, current, pprev, prev;
        return {
            next: function() {
            }
        };
    }
}
```

4. Напишем тело метода `next()` возвращаемого итератором объекта:

```
[Symbol.iterator]() {
    const cnt = this.#count;
    let i = 0, current, pprev, prev;
    return {
        next: function() {
            if (i < cnt) {
                if (i <= 1)
                    current = i;
                else
                    current = pprev + prev;
                [pprev, prev] = [prev, current];
                i++;
                return { value: current, done: false}
            } else
                return { done: true };
        }
    };
}
```

Сначала проверяем, не превысило ли значение переменной-счетчика `i` количество требуемых чисел. Если это так, то:

- если значение счетчика не превышает 1 (т. е. вычисляется первое или второе число) — используем значение счетчика в качестве очередного вычисленного числа Фибоначчи, присвоив его переменной `current`;
- в противном случае — вычисляем очередное число как сумму «пред-предыдущего» и предыдущего чисел, беря их из переменных `pprev` и `prev`. Результат также заносим в переменную `current`.

Далее заносим значение переменной `prev` в переменную `pprev`, а значение переменной `current` — в переменную `prev` (для чего применяем деструктурирование массива, описанное в *разд. 1.6.1*). Наконец, инкрементируем счетчик и выдаем вычисленное число.

Если же значение счетчика превысило количество требуемых чисел, выдаем признак того, что числа закончились.

5. Добавим после объявления класса код, запускающий вычисление 20 чисел (этого пока достаточно) и выводящий результат, в котором отдельные числа отделяются друг от друга вертикальными черточками:

```
class Fibonacci {
    . . .
}

const output = document.getElementById('output');
const oFib = new Fibonacci(20);
let s = '';
for (let v of oFib) {
    if (s)
        s += ' | ';
    s += v;
}
output.textContent = s;
```

Откроем страницу 3.3.html в веб-обозревателе и посмотрим на результат. Должно получиться вот что:

```
0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 |
610 | 987 | 1597 | 2584 | 4181
```

Вы можете указать в вызове конструктора другое количество чисел и посмотреть, что выдаст в ответ это веб-приложение.

3.4. Генераторы

Итераторы — инструмент, достаточно сложный в использовании, а код, написанный с их применением, не отличается наглядностью. Поэтому итераторы применяются нечасто и лишь в специфических ситуациях. Для более простых случаев удобнее применять генераторы, введенные в более поздние версии JavaScript.

> *Генератор* — функция, при каждом вызове возвращающая очередное вычисленное значение.

Можно сказать, что генератор — это функция, возвращающая итератор (наподобие рассмотренной в *разд. 3.2*), только записанная в более наглядном виде.

Формат объявления генератора:

```
function* <имя генератора>([<перечень параметров через запятую>]) {
    <тело генератора>
}
```

Для возврата очередного вычисленного значения из генератора применяется оператор возврата с приостановкой `yield`.

> Оператор возврата с приостановкой `yield <результат>` — возвращает заданный *результат* и приостанавливает (не прерывает!) исполнение генератора на том месте, где находится этот оператор. При последующем вызове генератора его выполнение продолжится с того места, на котором оно было приостановлено.

Пример простого генератора, последовательно возвращающего названия трех языков, которые применяются в веб-разработке:

```
function* languages() {
    yield 'HTML';
    yield 'CSS';
    yield 'JavaScript';
}
. . .
for (let el of languages())
    console.log(el);                                // Результат: HTML
                                                    //            CSS
                                                    //            JavaScript
```

Пример более сложного генератора, возвращающего последовательность целых чисел, находящихся в заданном диапазоне, с указанным шагом:

```
function* integers(min, max, step = 1) {
    for (let i = min; i <= max; i += step)
        yield i;
}
. . .
for (el of integers(10, 30, 7))
    console.log(el);                                // Результат: 10
                                                    //            17
                                                    //            24
```

В теле генератора можно вызвать другой генератор, применив оператор вызова генератора `yield*`.

> *Оператор вызова генератора* `yield* <генератор>`, — будучи указанным в теле генератора, вызывает указанный другой *генератор* и побуждает вызывающий генератор последовательно вернуть все значения, выданные вызываемым *генератором*.

Пример:

```
function* languages2() {
    yield 'PHP';
    yield* languages();
    yield 'MySQL';
}
. . .
for (let el of languages2())
    console.log(el);                    // Результат: PHP
                                        //            HTML
                                        //            CSS
                                        //            JavaScript
                                        //            MySQL
```

3.4.1. Методы-генераторы в классах

В классах можно объявлять методы-генераторы, поставив в начале имени метода символ «звездочки» (`*`). В объявлении закрытого метода символ `*` должен стоять перед символом `#`. Пример:

```
class SomeClass {
    . . .
    // Общедоступный метод-генератор
    *method1() { . . . }
    // Общедоступный статический метод-генератор
    static *staticMethod1() { . . . }
    // Закрытый метод-генератор
    *#method2() { . . . }
    // Закрытый статический метод-генератор
    static *#staticMethod2() { . . . }
}
```

Вызов методов-генераторов выполняется так же, как и генераторов, созданных на основе функций, только без указания «звездочки» в начале имени метода:

```
const sc = new SomeClass();
for (let el of sc.method1()) {
    . . .
}
for (let el of SomeClass.staticMethod1()) {
    . . .
}
```

Полезно знать...

Генераторы пришли в JavaScript из языка программирования Python.

3.5. Упражнение. Вычисление квадратных корней

Напишем веб-приложение, вычисляющее квадратные корни от последовательности чисел из указанного диапазона.

Для этого создадим генератор `sqrts()`, в качестве параметров принимающий начальное и конечное значения диапазона. Возвращать он будет служебный объект со свойствами `source` (исходное число) и `result` (квадратный корень из него).

1. Найдем в папке 3\!sources сопровождающего книгу электронного архива (см. *приложение*) файл template.html (страница, на основе которой будет создано веб-приложение), скопируем его куда-либо на локальный диск и дадим копии имя 3.5.html.

   Эта страница включает абзац с якорем `output`, в котором будет выведен результат. А в конце ее HTML-кода находится пустой тег `<script>`, в котором будет записан программный код.

2. Откроем в текстовом редакторе только что созданный файл 3.5.html и запишем в теге `<script>` код, объявляющий генератор `sqrts()`:

```
<script type="text/javascript">
    function* sqrts(min, max) {
        for (let i = min; i <= max; i++)
            yield { source: i, result: Math.sqrt(i) };
    }
</script>
```

3. Добавим после объявления генератора код, вычисляющий и выводящий квадратные корни чисел от 20 до 25 (этого достаточно для тестирования):

```
<script type="text/javascript">
    function* sqrts(min, max) {
        . . .
    }

    const output = document.getElementById('output');
    let s = '';
    for (let v of sqrts(20, 25)) {
        if (s)
            s += '<br>';
        s += v.source + ': ' + v.result;
    }
    output.innerHTML = s;
</script>
```

   Откроем страницу 3.5.html в веб-обозревателе и посмотрим на результат. Должно получиться вот что:

```
20: 4.47213595499958
21: 4.58257569495584
22: 4.69041575982343
23: 4.795831523312719
24: 4.898979485566356
25: 5
```

Вы можете указать в вызове генератора другие начальное и конечное значения диапазона чисел и посмотреть, что получится.

3.6. Асинхронные итераторы и генераторы

Если итератор или генератор должен выдавать значения, получаемые в результате выполнения какой-либо асинхронной операции (например, последовательность файлов, загружаемых посредством AJAX), этот итератор (генератор) можно превратить в асинхронный.

3.6.1. Асинхронные итераторы и цикл перебора с ожиданием

У асинхронного итератора метод `next()` объекта итератора должен возвращать промис. Как только очередное предназначенное к выдаче значение будет готово для выдачи, следует подтвердить этот промис с занесением в него нагрузки — объекта, хранящего это значение (он был описан в *разд. 3.2*). Если значений больше нет, в промис при подтверждении следует занести служебный объект с признаком исчерпания значений.

Вот пример фрагмент кода класса, реализующего асинхронный итератор:

```
[Symbol.iterator]() {
    return {
        next: function() {
            return new Promise((resolve, reject) => {
                // Получаем очередное значение для выдачи
                . . .
                // Значение получено?
                if (isValueExists)
                    // Если да, подтверждаем промис и заносим в него
                    // объект с полученным значением
                    resolve({ value: computedValue, done: false });
                else if (isValuesEnded)
                    // Если значений больше нет, подтверждаем промис
                    // и заносим в него объект с признаком исчерпания
                    // значений
                    resolve({ done: true });
                else if (isError)
                    // Если возникла ошибка, отклоняем промис
```

```
                    reject(new Error('Что-то пошло не так...'));
            });
        }
    };
}
```

Для обработки объектов такого класса следует применять цикл перебора с ожиданием, совмещающий в себе цикл перебора (см. *разд. 3.1*) и оператор ожидания (см. *разд. 2.4*).

> *Цикл перебора с ожиданием* — перебирает один за другим промисы из заданной последовательности. На каждой итерации цикла исполнение кода приостанавливается до тех пор, пока очередной промис не будет подтвержден.

Формат записи цикла перебора с ожиданием похож на формат записи обычного цикла перебора:

```
for await ([<оператор объявления>] <переменная> of <последовательность>)
{
    <тело цикла>
}
```

ВНИМАНИЕ!

Цикл перебора с ожиданием может быть использован только в асинхронной функции.

Пример использования такого цикла:

```
const obj = new SomeClass();
(async function () {
    for await (let val of obj) {
        // Обрабатываем выдаваемые объектом значения
        . . .
    }
})();
```

ВНИМАНИЕ!

Исключение, генерируемое асинхронным генератором, возникает *в самом выражении цикла перебора с ожиданием*, а не в его теле. При этом выполнение цикла прерывается.

Пример обработки исключений в цикле перебора с ожиданием:

```
(async function () {
    try {
        for await (let val of obj) {
            . . .
        }
    }
    catch(exception) {
        // Обрабатываем исключение
    }
})();
```

3.6.2. Асинхронные генераторы

Также можно объявить асинхронный генератор, использовав формат:

```
async function* <имя генератора>([<перечень параметров через запятую>]) {
    <тело генератора>
}
```

Пример:

```
async function* someFunc() {
    // Получаем очередное значение для выдачи
    . . .
    // Значение получено?
    if (isValueExists)
        // Если да, выполняем возврат значения с приостановкой
        yield computedValue;
    else if (isError)
        // Если возникла ошибка, генерируем исключение
        throw new Error('Что-то пошло не так...');
    // Если значений больше нет, ничего не делаем, и генератор завершает работу
}
```

3.6.2.1. Асинхронные методы-генераторы

Можно объявлять асинхронные методы-генераторы, поставив в начале имени метода символ * и предварив имя языковой конструкций `async` через пробел. Пример:

```
class SomeClass {
    . . .
    // Общедоступный асинхронный метод-генератор
    async *method1() { . . . }
    // Общедоступный статический асинхронный метод-генератор
    static async *staticMethod1() { . . . }
    // Закрытый асинхронный метод-генератор
    async *#method2() { . . . }
    // Закрытый статический асинхронный метод-генератор
    static async *#staticMethod2() { . . . }
}
```

Асинхронные методы-генераторы вызываются так же, как и асинхронные генераторы:

```
const sc = new SomeClass();
(async function () {
    for await (let el of sc.method1()) {
        . . .
    }
    for await (let el of SomeClass.staticMethod1()) {
        . . .
    }
})();
```

3.7. Упражнение. Массовая загрузка файлов, вариант 2

Переделаем результат выполнения *упражнения 2.3*, применив в новом варианте кода асинхронный генератор и цикл перебора с ожиданием. Это позволит сделать код нагляднее.

1. Найдем в папке 2\ex.2.3 сопровождающего книгу электронного архива (см. *приложение*) файлы 2.3.html (веб-страница с элементами, в которых будет выводиться содержимое файлов), 2.2_1.html, 2.2_2.html, 2.3_1.html (загружаемые HTML-файлы), 2.3.css (таблица стилей с оформлением страницы) и dataloader.js (веб-сценарий с кодом функции `loadData()`, загружающей файлы). Скопируем их куда-либо на локальный диск.
2. Откроем в текстовом редакторе копию страницы 2.3.html и удалим из тега `<script>` весь код, кроме самого первого выражения:

   ```
   <script type="text/javascript">
       const cOutput = document.body.querySelectorAll('.output');
       . . .
   </script>
   ```

   Это выражение получает доступ ко всем элементам со стилевым классом `output` — именно в них будут выводиться загруженные файлы.
3. Добавим в код сценария объявление массива с интернет-адресами загружаемых файлов:

   ```
   const cOutput = document.body.querySelectorAll('.output');
   const urls = ['2.2_1.html', '2.2_2.html', '2.3_1.html'];
   ```

4. Добавим код, объявляющий асинхронный генератор, который будет один за другим загружать файлы из только что объявленного массива:

   ```
   . . .
   const urls = ['2.2_1.html', '2.2_2.html', '2.3_1.html'];
   async function* loadFiles() {
       for (let el of urls)
           yield loadData(el);
   }
   ```

   Функция загрузки файла `loadData()`, написанная при выполнении *упражнения 2.2*, возвращает промис. Этот промис мы можем просто вернуть из асинхронного генератора — он будет выдан без всякого преобразования.
5. Добавим код, загружающий и выводящий файлы:

   ```
   async function* loadFiles() {
       . . .
   }
   (async function () {
       let i = 0;
   ```

```
    try {
        for await (let f of loadFiles())
            cOutput[i++].innerHTML = f;
    }
    catch(exception) {
        cOutput[i++].innerHTML = exception.message;
    }
})();
```

Нам необходимо последовательно перебирать все элементы со стилевым классом `output`. Для этого мы объявили переменную-счетчик `i`. После занесения содержимого очередного загруженного файла в очередной элемент инкрементируем значение этой переменной.

Скопируем файлы 2.3.html, 2.2_1.html, 2.2_2.html, 2.3_1.html, 2.3.css и dataloader.js в корневую папку веб-сервера, запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/2.3.html**. Мы должны увидеть то, что показано на рис. 2.3.

3.8. Самостоятельные упражнения

- Переделайте результат выполнения *упражнения 3.3*, используя метод-генератор вместо итератора. Дайте методу-генератору имя `getNumbers()`.
- Усовершенствуйте результат выполнения переделанного *упражнения 3.3*, заставив класс `Fibonacci` выдавать последовательность чисел, находящихся в указанном диапазоне. Результат сохраните под именем 3.8.html.

Урок 4
Модули

Модули JavaScript

Экспорт и импорт сущностей

Прогон модуля

4.1. Старый способ скрытия деталей реализации и его недостатки

Некоторые сущности (переменные, функции, классы), объявленные в коде, лучше скрыть от несанкционированного доступа. Например, если какая-либо переменная хранит важную для выполнения кода величину, следует предотвратить изменение ее значения посторонним кодом, в противном случае это может нарушить работу сценария. Особенно это касается библиотек, предназначенных для широкого распространения.

Ранее для скрытия деталей реализации применялись *замыкания*. Все сущности, которые требовалось скрыть, помещались в замыкание, а сущности, открываемые для общего доступа, возвращались из замыкания в качестве результата. Пример:

```
const Spoiler = (function () {
    function Spoiler__headerClick(evt) {
        . . .
    }

    return class {
        . . .
    }
})();
```

Здесь в качестве результата возвращается объявление класса компонента `Spoiler` (в новом синтаксисе), а объявление функции `Spoiler_headerClick()`, не предназначенной для непосредственного вызова сторонним кодом, скрывается.

4.2. Модули

В новых версиях JavaScript для скрытия деталей реализации предлагается более удобное и наглядное средство — модули.

> *Модуль* — веб-сценарий, способный выполнять экспорт объявленных в нем сущностей (переменных, функций, классов и пр.) для использования в других модулях и импорт необходимых для работы сущностей, объявленных в других модулях.
>
> *Экспорт* — пометка какой-либо сущности (переменной, функции, класса и др.) как предназначенной для использования в других модулях.
> *Импорт* — использование модулем для работы сущности, объявленной в другом модуле.

ВНИМАНИЕ!

Импортировать можно лишь те сущности, что были явно экспортированы в объявляющих их модулях. Все прочие сущности недоступны для импорта и, соответственно, для использования.

Вот основные принципы и правила применения модулей:

- Модули, как обычные веб-сценарии, могут храниться в отдельных файлах (*внешние модули*) и непосредственно в коде страниц (*встроенные модули*).
- Встроенный модуль помещается в HTML-код веб-страницы посредством парного тега `<script>`.
- Привязка внешнего модуля к странице также производится тегом `<script>`.

ВНИМАНИЕ!

В теге `<script>`, содержащем встроенный модуль или привязывающем к странице внешний модуль, необходимо указать атрибут `type` со значением `module`:

```
<script type="module">
    // Код встроенного модуля
</script>
. . .
<script type="module" src="/library/module1.js"></script>
```

Если этот атрибут тега не указать, веб-обозреватель *не посчитает веб-сценарий модулем* и *запретит выполнение операций импорта и экспорта*, что приведет к ошибке.

- Экспортировать из модулей следует лишь те сущности, которые рассчитаны на использование в стороннем коде.

  Сущности, не рассчитанные на использование в стороннем коде, экспортировать не нужно — тем самым они будут скрыты от несанкционированного доступа.
- Выражение, выполняющее импорт сущности из другого модуля, в любом случае содержит интернет-адрес этого модуля. Веб-обозреватель считывает его и выполняет загрузку модуля.
- Если модуль содержит код, *не рассчитанный на исполнение непосредственно в процессе загрузки страницы*, этот модуль *не нужно привязывать к странице* тегом `<script>`.

  Даже в таком случае веб-обозреватель при первой же попытке импортировать любую сущность из этого модуля успешно загрузит его.

Эта особенность существенно упрощает использование модулей, хранящих библиотеки (код, используемый в разных сайтах, в том числе и разными разработчиками), — не придется следить, чтобы все библиотечные модули были привязаны к странице.

Внимание!

Модули могут загружаться только с веб-сервера. Загрузка модулей с локальной файловой системы заблокирована в целях безопасности.

4.2.1. Экспорт

Экспорт сущностей из модуля можно выполнить двумя способами:

- *экспорт по умолчанию* — какой-либо одной сущности, с применением выражения следующего формата:

```
export default <экспортируемая сущность>
```

Внимание!

Экспортировать по умолчанию в текущем модуле можно лишь одну сущность.

Примеры:

```
// Экспорт по умолчанию либо класса...
export default class Spoiler { . . . }

// ...либо функции...
export default function someFunc() { . . . }

// ...либо переменной...
export default let someVar = 123;
// ...но не всего вместе в одном модуле!
```

Можно сначала объявить сущность, а потом экспортировать:

```
class Spoiler {}
export default Spoiler;

let someVar = 123;
export default someVar;
```

Для экспорта по умолчанию ранее объявленной сущности также можно использовать выражение формата:

```
export {<экспортируемая сущность> as default}
```

Пример:

```
let someVar = 123;
export {someVar as default};
```

- *именованный экспорт* — произвольного количества сущностей, с применением выражения формата:

```
export <экспортируемая сущность>
```

Примеры:

```
// Именованный экспорт класса, функции и переменной в одном модуле
export class Spoiler { . . . }
export function someFunc() { . . . }
export let someVar = 123;
```

Можно сначала объявить сущность, а потом экспортировать, применив выражение формата:

```
export {<перечень экспортируемых сущностей через запятую>}
```

В качестве *экспортируемой сущности* можно указать:

- имя сущности:

  ```
  class Spoiler {}
  let someVar = 123;
  export (Spoiler, someVar};
  ```

- конструкцию формата:

  ```
  <экспортируемая сущность> as <новое имя>
  ```

  В этом случае заданная *сущность* будет экспортирована под указанным *новым именем*. В последующем для импорта этой сущности в других модулях потребуется указать *новое имя*.

  Пример:

  ```
  export (Spoiler, someVar as sv};
  ```

Можно комбинировать оба этих типа экспорта в одном модуле:

```
// Экспорт по умолчанию класса
export default class Spoiler { . . . }
// Именованный экспорт функции и переменной
export function someFunc() { . . . }
export let someVar = 123;

// Другой способ экспорта
class Spoiler { . . . }
function someFunc() { . . . }
let someVar = 123;
export {Spoiler as default, someFunc, someVar};
```

Какой способ экспорта выбрать для каждой из сущности — дело вкуса. Обычно экспорт по умолчанию выполняют в случае какой-либо ключевой сущности (класса компонента, функции, выполняющей основную часть работы, и т. п.), а именованный — в случае вспомогательных сущностей (например, констант, хранящих предопределенные значения каких-либо параметров конструктора или функции, вспомогательных функций и др.).

4.2.2. Импорт

Импорт сущностей из другого модуля также можно выполнить двумя способами:

- *импорт по умолчанию* — сущности, экспортированной по умолчанию, с помощью выражения формата:

  ```
  import <имя сущности> from <интернет-адрес модуля>
  ```

  Импортированная сущность будет доступна в текущем модуле под указанным *именем* (которое не обязательно должно совпадать с именем, указанным у сущности в модуле, в котором она объявлена).

  Примеры:

  ```
  // Модуль /modules/lib.js
  export default class Spoiler { . . . }
  export function someFunc() { . . . }
  export let someVar = 123;
  export const someConst = 'JavaScript';

  // Другой модуль
  // Импортированный класс Spoiler будет доступен в коде под своим
  // изначальным именем
  import Spoiler from '/modules/lib.js';
  const oSpl = new Spoiler('spl1');

  // Можно указать у импортируемого класса другое имя
  import Spl from '/modules/lib.js';
  const oSpl = new Spl('spl1');
  ```

  В выражении импорта можно записать полный (содержащий и обозначение протокола, и адрес хоста, и путь к файлу), абсолютный *интернет-адрес модуля* (отсчитываемый от корневой папки сайта), а также относительный (отсчитываемый от текущей папки, в которой хранится импортирующий модуль). Относительный *интернет-адрес* записывается согласно следующим правилам:

  - чтобы отсчитать путь импортируемого модуля от текущей папки — *интернет-адрес* импортируемого модуля предваряется комбинацией символов `./`:

    ```
    // Пусть текущий модуль хранится в файле /modules/libs/mod.js
    // Импорт из модуля /modules/libs/lib.js
    import Spoiler from './lib.js';

    // Импорт из модуля /modules/libs/components/menu.js
    import MainMenu from './components/lib.js';
    ```

  - чтобы отсчитать путь импортируемого модуля от папки предыдущего уровня вложенности — *интернет-адрес* импортируемого модуля предваряется комбинацией символов `../`:

    ```
    // Импорт из модуля /modules/consts.js
    import Spoiler from '../consts.js';
    ```

```
// Импорт из модуля /modules/other/helpers.js
import MainMenu from '../other/helpers.js';
```

- чтобы отсчитать путь импортируемого модуля от папки второго или большего уровня вложенности — *интернет-адрес* импортируемого модуля предваряется комбинацией символов `../` нужное количество раз:

```
// Импорт из модуля /main.js
import Spoiler from '../../main.js';
```

♦ *именованный импорт* — только указанных сущностей, в отношении которых был применен именной экспорт, с помощью выражения формата:

```
import {<перечень импортируемых сущностей через запятую>} ↳
from <интернет-адрес модуля>
```

В качестве *импортируемой сущности* можно указать:

- имя сущности — указанное в объявляющем ее модуле:

```
// Импортируем функцию someFunc() и переменную someVar
import {someFunc, someVar} from '/modules/lib.js';
let res = someFunc() + someVar;
```

- конструкцию формата:

```
<изначальное имя сущности> as <новое имя сущности>
```

В этом случае импортированная сущность будет доступна под указанным *новым именем*.

Примеры:

```
// Импортируем переменную someVar и константу someConst,
// давая последней новое имя — scst
import {someVar, someConst as scst} from '/modules/lib.js';
let s = 'Язык ' + scst;
```

♦ именованный импорт — сразу всех сущностей, в отношении которых был применен именной экспорт, с помощью выражения формата:

```
import * as <имя переменной> from <интернет-адрес модуля>
```

В этом случае исполняющая среда JavaScript создаст служебный объект, в нем сформирует свойства, одноименные импортируемым сущностям, присвоит сущности этим свойствам, а сам созданный объект занесет в переменную с указанным *именем*.

Пример:

```
import * as modLib from '/modules/lib.js';
let res = modLib.someFunc() + modLib.someVar;
```

Можно комбинировать эти типы импорта в одном выражении. Примеры:

```
// Импорт:
//   * класса Spoiler, экспортированного по умолчанию;
```

```
//   * функции someFunc() и константы someConst,
//     подвергшихся именованному экспорту.
import Spoiler, {someFunc, someConst} from '/modules/lib.js';

// Импорт:
//   * класса Spoiler, экспортированного по умолчанию;
//   * все остальных сущностей, подвергшихся именованному экспорту.
import Spoiler, * as modLib from '/modules/lib.js';
```

4.2.3. Прогон

При импорте первой сущности из какого-либо модуля веб-обозреватель загружает этот модуль и выполняет весь содержащийся в нем код. Однако иногда не требуется импортировать сущности из модуля, а произвести его прогон.

> *Прогон* — выполнение программного кода модуля без импорта из него каких бы то ни было сущностей.

Например, если требуется привязать много обработчиков к разным событиям, имеет смысл вынести соответствующий код в отдельный модуль, а потом выполнить его прогон.

Прогон модуля производится посредством выражения формата:

```
import <интернет-адрес модуля>
```

Пример:

```
import '/modules/initialization.js';
```

4.3. Упражнение. Реорганизация кода спойлера

Реорганизуем код веб-компонента спойлера, написанного при выполнении *упражнения 1.7*, с применением модулей. Это позволит сделать код более наглядным и немного сократить его объем за счет удаления большей части тегов `<script>`.

1. Найдем в папке 1\ex1.7 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css, spoiler.css, spoiler.js, spoiler2.js, spoiler3.js и скопируем их куда-либо на локальный диск.

2. Откроем в текстовом редакторе копию файла spoiler.js и вставим в содержащийся в нем код фрагмент, выполняющий экспорт по умолчанию класса `Spoiler`:

```
function Spoiler__headerClick(evt) {
    . . .
}

export default class Spoiler {
    . . .
}
```

Функцию `Spoiler_headerClick()` экспортировать не будем, т. к. она является деталью реализации спойлера, и ее следует предохранить от несанкционированного доступа.

3. Откроем в текстовом редакторе копию файла spoiler2.js и вставим в самое начало кода выражение, импортирующее класс `Spoiler` из модуля spoiler.js:

```
import Spoiler from './spoiler.js';

class Spoiler2 extends Spoiler {
    . . .
}
```

4. Добавим в код фрагмент, выполняющий экспорт по умолчанию класса `Spoiler2`:

```
import Spoiler from './spoiler.js';

export default class Spoiler2 extends Spoiler {
    . . .
}
```

5. Откроем в текстовом редакторе копию файла spoiler3.js и вставим в него аналогичные фрагменты кода, импортирующие класс `Spoiler2` из модуля spoiler2.js и экспортирующие по умолчанию класс `Spoiler3`.

   Поскольку мы превратили файлы веб-сценариев spoiler.js, spoiler2.js и spoiler3.js в модули, привязывать их к странице посредством тегов <script> более не нужно.

6. Откроем файл index.html и удалим все теги <script> из тега <head>:

```
<html>
    <head>
        . . .
        <link href="styles.css" rel="stylesheet" type="text/css">
        <link href="spoiler.css" rel="stylesheet" type="text/css">
        <script src="spoiler.js" type="text/javascript"></script>
        <script src="spoiler2.js" type="text/javascript"></script>
        <script src="spoiler3.js" type="text/javascript"></script>
    </head>
    . . .
</html>
```

7. Заменим в теге <script>, находящемся в конце HTML-кода и хранящем встроенный сценарий, значение атрибута `type` на `module`, чтобы превратить этот сценарий в модуль:

```
<html>
    . . .
</html>
<script type="module">
    const spls = Spoiler3.init({shown: true}, 'spl1', 'spl2');
</script>
```

8. Добавим в начало встроенного модуля выражение, импортирующее класс `Spoiler3` из модуля spoiler3.js:

```
<script type="module">
    import Spoiler3 from './spoiler3.js';
    const spls = Spoiler3.init({shown: true}, 'spl1', 'spl2');
</script>
```

   Скопируем файлы index.html, styles.css, spoiler.css, spoiler.js, spoiler2.js и spoiler3.js в корневую папку веб-сервера, запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/**. Если мы не допустили ошибок, увидим два развернутых спойлера.

4.4. Самостоятельное упражнение

Переделайте результат выполнения *упражнения 3.7* с использованием модулей следующим образом:

- подвергните функцию `loadData()` из модуля dataloader.js экспорту по умолчанию;
- вынесите объявление константы `urls` (хранит массив интернет-адресов загружаемых файлов) из страницы 2.3.html во вновь созданный модуль fileurls.js и подвергните эту константу именованному экспорту под изначальным именем;
- вынесите асинхронную функцию `loadFiles()` из страницы 2.3.html во вновь созданный модуль fileloader.js и подвергните ее экспорту по умолчанию. Не забудьте в коде этого модуля импортировать из модуля dataloader.js функцию `loadData()`, иначе ничего не заработает;
- перепишите встроенный в страницу 2.3.html сценарий так, чтобы он использовал массив `urls` и функцию `loadFiles()` из ранее созданных модулей.

ЧАСТЬ II
HTML API: новые полезные мелочи

- Детектор видимости
- Загрузчик файлов AJAX
- Встроенная СУБД

Урок 5
Детектор видимости

Область видимости

Детектор видимости

Отслеживание попадания

Часто требуется отслеживать, присутствует ли какой-либо элемент страницы в области видимости.

> *Область видимости* — область в окне веб-обозревателя или в элементе страницы с прокруткой, в которой отображается содержимое страницы (или элемента с прокруткой).

Элементы страницы, частично или полностью попадающие в область видимости, видимы посетителю также частично или полностью. Элементы, выходящие за пределы области видимости, невидимы посетителю (рис. 5.1).

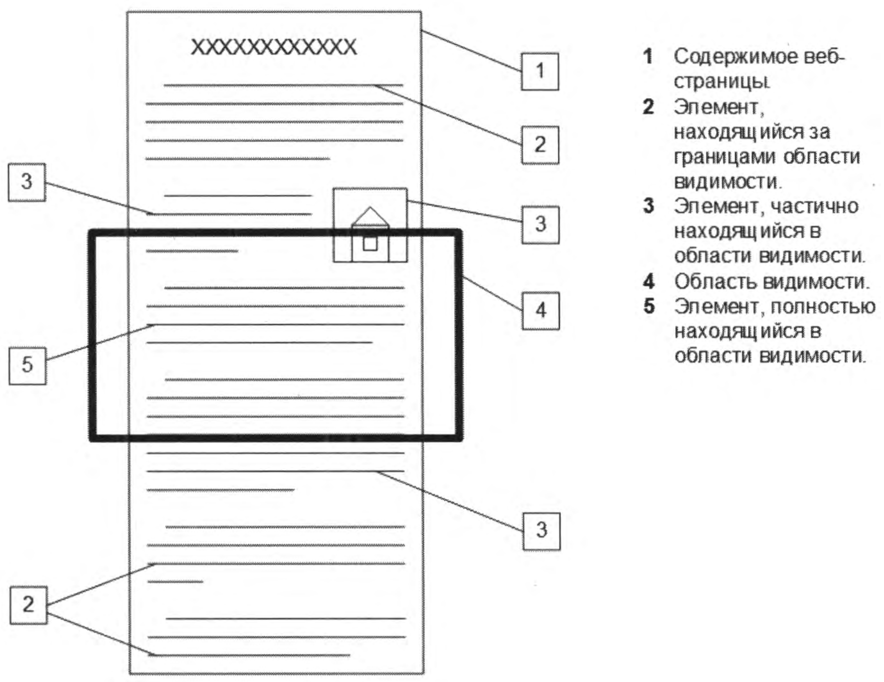


Рис. 5.1. Схематичное изображение области видимости

При прокрутке страницы (или содержимого элемента с прокруткой) область видимости перемещается по странице (содержимому элемента). При этом одни элементы постепенно входят в область видимости и становятся видимыми, а другие уходят за ее границы и, соответственно, скрываются.

Отслеживание попадания элементов в область видимости может пригодиться, например, при реализации отложенной загрузки изображений.

> *Отложенная загрузка* — загрузка только тех элементов страницы (обычно изображений), которые находятся в области видимости.

Отложенная загрузка изображений позволяет ускорить первоначальный вывод страниц, поскольку изначально загружаются и выводятся только те изображения, что непосредственно видимы посетителю. Остальные изображения загружаются позже, как только посетитель прокрутит страницу до тех ее участков, в которых находятся эти изображения.

Ранее для проверки попадания каких-либо элементов в область видимости приходилось обрабатывать событие `scroll`, возникающего при прокрутке, вычислять местоположение каждого отслеживаемого элемента и проверять, находится ли он в границах области видимости (которые также приходилось вычислять).

5.1. Использование детектора видимости

Новые версии JavaScript также принесли многочисленные полезные нововведения в интерфейсы HTML API. Одним из таких нововведений является детектор видимости.

> *HTML API* (HTML Application Programming Interface, интерфейс программирования приложений HTML) — набор программных интерфейсов, не относящихся к управлению страницей и веб-обозревателем.

> *Детектор видимости* — программный инструмент, отслеживающий появление заданного элемента страницы в области видимости и выход из нее.

5.1.1. Создание детектора видимости

Детектор видимости представляется объектом класса `IntersectionObserver`. Конструктор этого класса имеет следующий формат вызова:

```
IntersectionObserver(<функция-обработчик>[, <параметры>])
```

Функция-обработчик вызывается, когда часть какого-либо из отслеживаемых элементов, заданная указанным пределом (о нем — чуть позже), в процессе прокрутки окажется в области видимости. Мы рассмотрим эту функцию позже.

> **Внимание!**
>
> *Функция-обработчик* в процессе прокрутки вызывается дважды: когда отслеживаемый элемент входит в область видимости и когда он уходит из нее.

Необязательные *параметры* задаются в виде служебного объекта со следующими свойствами:

- `root` — может содержать:
  - `null` — детектор будет отслеживать попадание элементов в область видимости окна веб-обозревателя в результате прокрутки самой страницы;
  - ссылку на объект элемента с прокруткой — тогда будет отслеживаться попадание элементов, вложенных в указанный элемент, в область видимости этого элемента.

  Значение по умолчанию: `null`;
- `rootMargin` — отступы вокруг области видимости, заданные в таком же формате, что поддерживается атрибутами стилей CSS `margin` и `padding`. Указание отступов позволяет уменьшить размеры области видимости для достижения каких-либо специальных эффектов. Значение по умолчанию: `'0px'` (т. е. отступы отсутствуют);
- `threshold` — предел или массив пределов.

  > *Предел* — числовое значение, указывающее часть отслеживаемого элемента, которая должна присутствовать в области видимости.

  Предел указывается в виде вещественного числа от `0.0` (хотя бы один пиксел отслеживаемого элемента должен попасть в область видимости) до `1.0` (отслеживаемый элемент должен полностью попасть в область видимости). Можно указать:
  - один предел — тогда *функция-обработчик* будет вызвана один раз, как только обозначенная заданным пределом часть отслеживаемого элемента окажется в области видимости.

    Например, если указать предел `0.5`, то при прокрутке страницы, как только половина отслеживаемого элемента войдет в область видимости, будет вызвана *функция-обработчик*. При дальнейшей прокрутке, когда элемент станет уходить из области видимости и будет видим наполовину, *функция* будет вызвана во второй раз;
  - массив пределов — тогда *функция-обработчик* будет вызываться каждый раз, когда часть отслеживаемого элемента, обозначенная очередным пределом из заданного массива, окажется в области видимости.

    Например, если указать массив `[0.1, 0.9]`, то когда элемент будет входить в область видимости, *функция-обработчик* вызовется дважды: как только видимыми станут 10% и 90% элемента. А когда элемент будет уходить из области видимости, *функция* также вызовется дважды: когда видимыми останутся 90% и 10% отслеживаемого элемента.

  Значение свойства по умолчанию: `0.0`.

Примеры:

```
// Отслеживаем прокрутку самой страницы в веб-обозревателе.
// Детектор видимости просигнализирует, когда половина очередного
// отслеживаемого элемента окажется в области видимости.
const options1 = {threshold: 0.5};
const vd1 = new IntersectionObserver(vdHandler, options1);

// То же самое, только отслеживаем прокрутку содержимого элемента
// с прокруткой scroller. Отслеживаемые элементы должны быть вложены
// в этот элемент.
const elScroller = document.getElementById('scroller');
const options2 = {root: elScroller, threshold: 0.5};
const vd2 = new IntersectionObserver(vdHandler, options2);

// Детектор видимости просигнализирует, когда в области видимости
// окажется 1/4, 1/2, 3/4 отслеживаемого элемента и, наконец, элемент
// полностью.
const options3 = {threshold: [0.25, 0.5, 0.75, 1.0]};
const vd3 = new IntersectionObserver(vdHandler, options3);

// Отслеживаемый элемент должен появиться в области видимости полностью,
// причем на расстоянии в 10 пунктов от верхнего или нижнего ее края
// (в зависимости от направления прокрутки)
const options4 = {threshold: 1.0, rootMargin: '10pt 0pt'};
const vd4 = new IntersectionObserver(vdHandler, options4);
```

Класс `IntersectionObserver` поддерживает доступные только для чтения свойства `root`, `rootMargin` и `threshold`, соответствующие одноименным свойствам переданного конструктору конфигурационного объекта с *параметрами*.

5.1.2. Указание отслеживаемых элементов

Сразу после создания объект детектора видимости не отслеживает ни один элемент страницы. Отслеживаемые им элементы следует указать явно.

Для указания отслеживаемого элемента применяется метод `observe(<элемент>)` класса `IntersectionObserver`. *Элемент* должен задаваться в виде ссылки на представляющий его объект. Пример:

```
const elImg1 = document.getElementById('img1');
vd1.observe(elImg1);
```

Метод `unobserve(<элемент>)` того же класса убирает заданный *элемент* из числа отслеживаемых:

```
const elImg3 = document.getElementById('img3');
vd1.unobserve(elImg3);
```

Метод `disconnect()` убирает из числа отслеживаемых все ранее заданные элементы.

5.1.3. Отслеживание попадания элементов в область видимости

Когда часть какого-либо из отслеживаемых элементов, обозначенная одним из заданных в параметрах пределов, оказывается в области видимости (причем неважно, входит ли этот элемент в область или уходит из нее), детектор видимости вызывает заданную при его создании функцию-обработчик (подробности о создании объекта детектора — в *разд. 5.1.1*).

Эта функция должна принимать два параметра:

- массив объектов класса `IntersectionObserverEntry`, каждый из которых хранит сведения об одном из отслеживаемых элементов, находящихся в области видимости;
- ссылку на объект текущего детектора видимости.

Класс `IntersectionObserverEntry` поддерживает следующие свойства, доступные только для чтения:

- `target` — ссылка на сам отслеживаемый элемент;
- `intersectionRatio` — видимая часть элемента в виде вещественного числа от `0.0` (элемент полностью находится за границами области видимости и, таким образом, невидим) до `1.0` (элемент полностью находится в области видимости и, таким образом, видим целиком);
- `intersectionRect` — объект класса `DOMRect` с координатами и размерами видимой области текущего элемента, представленными в пикселах. Этот объект аналогичен тому, что возвращается методом `getBoundingClientRect()` элемента страницы;
- `isIntersecting` — если `true`, элемент находится в области видимости, если `false` — за ее пределами;
- `rootBounds` — объект класса `DOMRect` с координатами и размерами области видимости, представленными в пикселах;
- `boundingClientRect` — объект класса `DOMRect` с координатами и размерами текущего элемента, представленными в пикселах. Это тот же самый объект, что возвращается методом `getBoundingClientRect()` элемента страницы;
- `time` — текущее время в виде вещественного числа. Представляет собой количество миллисекунд, прошедшее с момента открытия страницы. Может быть использовано для вычисления времени, в течение которого элемент присутствует в области видимости.

Пример отслеживания попадания элементов в область видимости:

```
function vdHandler(els, detector) {
    els.forEach((data) => {
        if (data.intersectionRatio > 0) {
            // Элемент видим. Делаем с ним что-либо.
            . . .
```

```
        } else {
            // Элемент невидим. Делаем с ним что-либо.
            . . .
        }
    });
}
```

Массив объектов класса, описывающих видимость отслеживаемых элементов, также можно получить в любой момент, вызвав метод `takeRecords()` у объекта детектора видимости:

```
const els = vd1.takeRecord();
els.forEach((data) => {
    . . .
});
```

5.2. Упражнение. Делаем подсвечивающиеся изображения

Сделаем так, чтобы графические изображения, присутствующие на странице, как бы подсвечивались, попадая в область видимости.

Для этого будем менять их уровень непрозрачности от 20% (когда изображение только входит в область видимости) до полной (когда изображение станет полностью видимым). Уровень непрозрачности изображения будет меняться, как только очередная его $^1/_5$ часть окажется в области видимости. У области видимости укажем отступы сверху и снизу равные 10 пунктам, а слева и справа — равные нулю.

1. Найдем в папке 5\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы index.html (веб-страница с изображениями), styles.css (таблица стилей с оформлением страницы) и папку images (с шестью файлами, хранящими изображения). Скопируем их куда-либо на локальный диск.

   Страница index.html содержит шесть тегов `<img>`, выводящих изображения и вложенных в теги `<section>`.

2. Откроем в текстовом редакторе копию страницы index.html и поместим в находящийся в конце HTML-кода «пустой» тег `<script>` код, создающий конфигурационный объект с параметрами для детектора видимости:

   ```
   <script type="text/javascript">
       const options = {
           rootMargins: '20pt 0pt',
           threshold: [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
       };
   </script>
   ```

   Свойству `threshold` мы присвоили массив с шестью пределами, отличающимися друг от друга на 0,2. В результате очередное изображение будет менять свою непрозрачность, как только очередная его $^1/_5$ часть окажется в области видимости (как мы условились ранее).

3. Добавим объявление функции-обработчика попадания элемента в область видимости:

```
function vdHandler(els, detector) {
    els.forEach((data) => {
        data.target.style.opacity = 0.2 + data.intersectionRatio *
                                          0.8;
    });
}
```

Эта функция будет менять уровень непрозрачности элемента от 20 до 100%.

4. Добавим код, создающий объект детектора видимости:

```
const vd = new IntersectionObserver(vdHandler, options);
```

5. Добавим код, который получает доступ ко всем тегам `<img>`, вложенным в теги `<section>`, и добавляет их в число отслеживаемых:

```
const cImgs = document.querySelectorAll('section img');
cImgs.forEach((el) => {
    vd.observe(el);
});
```

Откроем страницу index.html в веб-обозревателе и прокрутим ее вниз. Чем бóльшая часть очередного изображения окажется в области видимости, тем сильнее оно будет подсвечено (рис. 5.2).

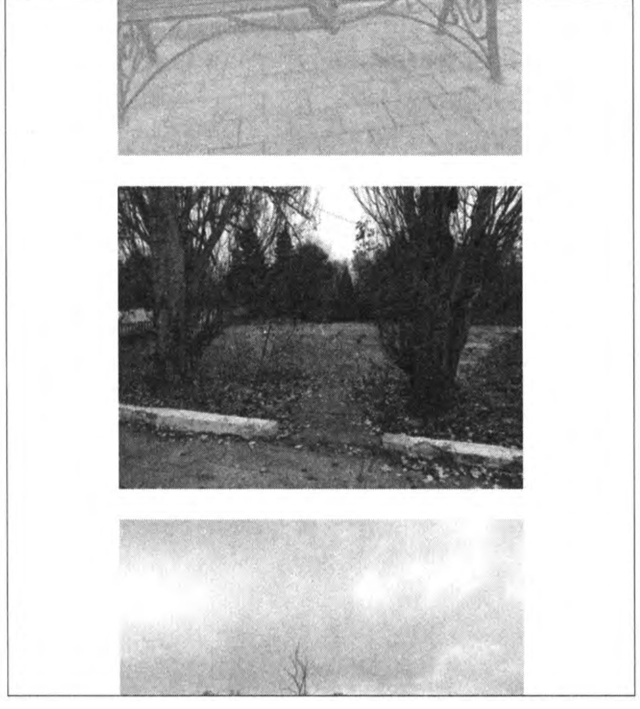


Рис. 5.2. Веб-страница с подсвечивающимися изображениями (изображение в центре подсвечено, изображения выше и ниже его — нет)

5.3. Самостоятельное упражнение

Реализуйте отложенную загрузку изображений, использовав как отправную точку файлы index.html, styles.css и папку images из папки 5\!sources. Несколько подсказок:

- укажите интернет-адреса файлов с изображениями в атрибутах с именами, например, `dsrc` тегов `<img>` (а не в атрибутах `src`, как обычно). Пример (атрибут тега `dsrc` подчеркнут):

```
<img dsrc="images/1.jpg">
```

- как только очередное изображение попадает в область видимости (это можно выяснить, проверив значение свойства `isIntersecting` объекта, хранящего сведения об отслеживаемом элементе, подробности — в *разд. 5.1.3*), занесите интернет-адрес из атрибута тега `dsrc` в атрибут тега `src`. Это вызовет загрузку и вывод изображения.

  Созданный ранее в теге `<img>` атрибут `dsrc` не поддерживается веб-обозревателями, и получить к нему доступ через одноименное свойство не удастся. Чтобы извлечь значение атрибута тега с указанным *именем*, используйте метод `getAttribute(<имя атрибута тега>)`. Пример:

```
let deferredSrc = elImg.getAttribute('dsrc');
```

- сразу после этого удалите загруженное изображение из числа отслеживаемых (поскольку отслеживать их больше не нужно);
- для наилучшего эффекта укажите в каждом теге `<img>` размеры выводящегося в нем изображения, использовав атрибуты тега `width` и `height` (ширина и высота соответственно). Пример:

```
<img dsrc="images/1.jpg" width="400" height="533">
```

  И задайте для детектора видимости предел равный 0,3.

Урок 6
Загрузчик данных AJAX

Новый загрузчик данных AJAX

Указание заголовков HTTP

Создание запросов

Работа с полученными ответами

Ранее загрузка данных посредством технологии AJAX требовала применения класса `XMLHttpRequest` и написания довольно громоздкого и запутанного кода (см. пример из *разд. 2.1.1*).

Новые версии HTML API предоставляют для этой цели весьма удобный загрузчик данных, реализованный в виде единственного метода и возвращающий в качестве результата промис. Последний может быть обработан как указанием функций then и catch, так и с помощью оператора ожидания (подробности — в *главе 2*).

6.1. Отправка запроса на загрузку данных

Для отправки запроса на загрузку данных с указанного *интернет-адреса* посредством нового загрузчика следует использовать метод `fetch()` класса `Window`:

```
fetch(<интернет-адрес>[, <параметры>])
```

Необязательные *параметры* указываются в виде служебного объекта со следующими свойствами:

- `method` — HTTP-метод, используемый для получения данных, в виде строкового обозначения (`'get'`, `'post'` и т. д.). Значение по умолчанию: `'get'`;
- `body` — данные, передаваемые серверу. Могут указываться в виде служебного объекта или объекта класса `FormData`. Принимаются во внимание, только если для получения данных используется метод, отличный от GET и HEAD. Значение по умолчанию: `null` (никакие данные серверу не передаются);
- `headers` — дополнительные заголовки, добавляемые в отправляемый HTTP-запрос. Могут указываться в виде служебного объекта или объекта класса `Headers`. Задание заголовков запроса будет рассмотрено позже. Значение по умолчанию: `null` (никакие дополнительные заголовки в запрос не добавляются);
- `cache` — обозначение режима работы кэша веб-обозревателя, заданное в виде строки. Подробно разные режимы работы кэша и их обозначения будут рассмотрены позже.

Метод `fetch()` в качестве результата возвращает промис, который:

- подтверждается — при получении *любого серверного ответа, даже содержащего сообщение об ошибке*;
- отклоняется — *только если запрос не удалось отправить* из-за проблем с сетью или некорректных параметров, заданных в вызове метода.

Примеры:

```
// Загружаем файл
const prmPictures = fetch('/data/pictures.json');

// Отправляем серверной программе сведения о пользователе для выполнения
// входа и ожидаем получение результата — признака, удалось ли выполнить
// вход на сайт
const fd = new FormData();
fd.append('name', 'pupkin_vv');
fd.append('password', 'sUpErUsEr');
const prmLogin = fetch('/programs/admin/login.php',
                       { method: 'post', body: fd });
```

6.1.1. Задание заголовков HTTP-запроса

Иногда бывает необходимо добавить в отправляемый HTTP-запрос дополнительные заголовки, описывающие либо отправляемые бэкенду данные, либо тип результата, которые фронтенд желает получить от бэкенда. Например, если бэкенд ожидает от сервера ответ в формате JSON, он должен добавить в отправляемый запрос заголовок `Accept: application/json`.

Дополнительные заголовки для запроса указываются в свойстве `headers` конфигурационного объекта с параметрами, передаваемого методу `fetch()`. Заголовки можно указать в виде:

- служебного объекта, в котором свойства задают имена заголовков, а значения свойств — значения этих заголовков:

```
const options = {
    headers: {
        Accept: 'application/json',
        // Также указываем серверу, чтобы он отправил ответ с
        // русскоязычным текстом.
        // Если имя заголовка включает символы, недопустимые в именах
        // свойств, его следует заключить в одинарные или двойные
        // кавычки.
        'Accept-Language': 'ru-RU'
    }
};
. . .
const prmData = fetch('/programs/get-data.php', options);
```

- объекта класса `Headers`. Конструктор этого класса вызывается в формате:

  `Headers([<служебный объект с заголовками>])`

  Заголовки, заданные в указанном *служебном объекте*, будут сразу же добавлены в создаваемый объект заголовков. Пример:

  ```
  const oHeaders = new Headers({
      Accept: 'application/json',
      'Accept-Language': 'ru-RU'
  });
  const prmData2 = fetch('/programs/get-data.php',
                         { headers: oHeaders });
  ```

  Набор заголовков, представленный в виде объекта класса `Headers`, обрабатывается быстрее, чем записанный в виде служебного объекта. Поэтому класс `Headers` следует использовать в случае, если один и тот же набор заголовков должен заноситься в разные HTTP-запросы, — это немного повысит производительность.

Класс `Headers` поддерживает следующие полезные методы:

- `append(<имя>, <значение>)` — добавляет в текущий объект новый заголовок с заданными *именем* и *значением*. Этот метод удобно использовать, если какой-либо заголовок следует добавить только при выполнении какого-либо условия. Пример:

  ```
  const oHeaders2 = new Headers();
  oHeaders2.append('Accept-Language', 'ru-RU');
  if (expectJSONType)
      oHeaders2.append('Accept', 'application/json');
  ```

  Если заголовок с заданным *именем* уже существует, указанное *значение* будет добавлено к уже существующему и отделено от него запятой:

  ```
  oHeaders2.append('Accept-Language', 'en-US');
  // В результате отправляемый запрос будет содержать заголовок:
  // Accept-Language: ru-RU, en-US
  ```

- `get(<имя>)` — возвращает значение заголовка с заданным *именем* или `null`, если такого заголовка нет:

  ```
  let s = oHeaders2.get('Accept-Language'); // Результат: 'ru-RU, en-US'
  s = oHeaders2.get('Other-Header');        // Результат: null
  ```

- `has(<имя>)` — возвращает `true`, если заголовок с заданным именем существует, и `false` — в противном случае:

  ```
  s = oHeaders2.has('Accept-Language');     // Результат: true
  s = oHeaders2.has('Other-Header');        // Результат: false
  ```

- `set()` — аналогичен `append()`, только, если заголовок с указанным именем уже существует, полностью заменяет это значение новым:

```
oHeaders2.set('Accept-Language', 'ru-RU');
let s = oHeaders2.get('Accept-Language'); // Результат: 'ru-RU'
```

- `delete(<имя>)` — удаляет из текущего объекта заголовок с заданным *именем*. Если такого заголовка нет, ничего не делает;
- `entries()` — возвращает итератор, позволяющий перебрать все заголовки, которые имеются в текущем объекте, в цикле перебора (подробности — в *разд. 3.1*). На каждой итерации заданной в цикле переменной присваивается массив из двух элементов: имени очередного заголовка и его значения. Пример:

```
for (let pair of oHeaders2.entries())
    console.log(pair[0] + ': ' + pair[1]);
```

- `keys()` — то же самое, что и `entries()`, только на каждой итерации заданной в цикле переменной присваивается имя очередного заголовка;
- `values()` — то же самое, что и `entries()`, только на каждой итерации заданной в цикле переменной присваивается значение очередного заголовка.

6.1.2. Управление кэшированием загруженных данных

> *Кэширование* — сохранение загруженных с сервера файлов в локальном кэше. Выполняется веб-обозревателем автоматически.
>
> *Локальный кэш* (или *клиентский кэш*) — хранилище, в котором веб-обозреватель сохраняет все загруженные с сервера файлы. В разных веб-обозревателях может быть реализовано в виде папки с файлами или одного большого файла.

Когда впоследствии веб-обозревателю понадобится открыть ранее загруженный файл, он извлечет его из кэша (если кэшированная копия файла еще актуальна), не загружая по сети повторно. Это значительно повышает производительность.

При отправке файла веб-обозревателю сервер может указать время, до которого копия этого файла, сохраненная в кэше, является актуальной. Перед извлечением файла из кэша веб-обозреватель проверяет, не истекло ли указанное время актуальности (оно также сохраняется в кэше). Если время истекло, файл загружается с сервера повторно и записывается в кэш вместо старой редакции того же файла.

В случае, если время актуальности кэшированной копии файла определить не удается (например, сервер при отправке файла не указал его), веб-обозреватель отправляет серверу проверочный запрос с целью выяснить, актуален ли этот файл. Если полученный от сервера ответ подтвердит актуальность файла, веб-обозреватель извлекает файл из кэша, в противном случае — загружает его с сервера (и, опять же, записывает в кэш, затерев старую редакцию того же файла).

Таков режим работы локального кэша по умолчанию, и, как правило, он задействуется в большинстве случаев. Однако иногда какой-либо устаревший файл надолго остается сохраненным в кэше, и веб-обозреватель выдает его раз за разом несмотря на то, что на сервере уже появилась более новая редакция этого файла.

Эта проблема носит название «*застревания в кэше*», и причиной ее является ошибка в серверной программе или некорректная настройка сервера. Решить такую проблему можно, указав при загрузке проблемного файла другой режим работы кэша.

Режим работы кэша задается в виде строкового обозначения в свойстве `cache` конфигурационного объекта с параметрами, передаваемого методу `fetch()`. Поддерживаются следующие режимы:

- `'default'` — режим работы по умолчанию (был описан ранее);
- `'no-store'` — запрашиваемый файл всегда загружается с сервера (даже если в кэше есть его актуальная копия) и *не записывается* в кэш.

  Этот режим применяется для загрузки каких-либо часто обновляемых данных, чтобы гарантированно исключить их «застревание в кэше» (при этом не захламляя кэш копиями файлов, которые никогда не будут их него извлекаться). Также его применяют для загрузки секретных данных (например, имени и пароля пользователя), которые нежелательно сохранять где-либо на локальном диске.

  Остальные режимы применяются в крайне специфических случаях:

- `'reload'` — запрашиваемый файл всегда загружается с сервера (даже если в кэше есть его актуальная копия) и *записывается* в кэш;
- `'no-cache'` — аналогичен режиму по умолчанию, только, если запрашиваемый файл есть в кэше, даже еще актуальный, веб-обозреватель всегда отправляет серверу проверочный запрос, чтобы подтвердить актуальность файла;
- `'force-cache'` — аналогичен режиму по умолчанию, только, если запрашиваемый файл есть в кэше (даже если его актуальность определить не удается), всегда используется именно он;
- `'only-if-cached'` — всегда используются только файлы, имеющиеся в кэше. Если какого-либо файла в кэше нет, возвращается сообщение об ошибке с кодом 504 (превышено допустимое время выполнения запроса).

Пример:

```
// Предписываем загрузить файл с сервера, даже если он уже кэширован,
// и не кэшировать (поскольку из кэша мы его брать все равно не будем)
const prmPictures = fetch('/data/pictures.json', { cache: 'no-store' });
```

> ***НА ЗАМЕТКУ...***
>
> В конфигурационном объекте передаваемому методу `fetch()` можно указать и другие параметры. Однако они применяются в очень специфических случаях и в книге не описываются. Полное описание метода `fetch()` со всеми поддерживаемыми им параметрами можно найти по адресу: **https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch**.

6.1.3. Использование объектов запросов

Если в разных местах кода необходимо загружать данные с одного и того же интернет-адреса с одними и теми же параметрами, удобнее заранее создать готовый объект запроса и подставлять в вызов метода `fetch()` именно его. Готовый объект

запроса обрабатывается веб-обозревателем немного быстрее, чем интернет-адрес и параметры запроса, заданные по отдельности.

Объект запроса создается на основе класса `Request`. Его конструктор вызывается в формате, схожем с форматом вызова метода `fetch()`:

```
Request(<интернет-адрес>[, <параметры>])
```

При использовании *готового объекта запроса* метод `fetch()` следует вызвать в формате:

```
fetch(<готовый объект запроса>)
```

Пример:

```
const options2 = {
    headers: {
        Accept: 'application/json',
        'Accept-Language': 'ru-RU'
    },
    cache: 'no-store'
};
const rqPictures = new Request('/data/pictures.json', options2);
const prmPictures3 = fetch(rqPictures);
```

6.2. Получение загруженных данных

Как отмечалось в *разд. 6.1*, метод `fetch()` возвращает промис. Этот промис подтверждается, если от сервера был получен хоть какой-то ответ, даже ошибочный, и отклоняется, если запрос не удастся отправить.

Нагрузкой этого промиса является объект класса `Response`, представляющий полученный ответ. Этот класс поддерживает ряд доступных только для чтения свойств, позволяющих выяснить, увенчалась ли успехом загрузка данных, и узнать некоторые сведения о самом ответе:

- `ok` — если `true`, данные были успешно получены, и `false` — если на стороне сервера произошла какая-либо ошибка.

  Данные считаются успешно полученными, если полученный от сервера ответ имеет код статуса от 200 до 299 включительно;
- `status` — целочисленный код статуса ответа;
- `statusText` — сам статус ответа в строковом виде;
- `headers` — заголовки ответа в виде объекта класса `Headers` (был описан в *разд. 6.1.1*);
- `url` — интернет-адрес, с которого пришел ответ.

Чтобы извлечь сами полученные с ответом данные, следует вызвать один из приведенных далее методов класса `Response`. Какой именно метод следует вызвать — зависит от типа полученных данных (обычный текст или HTML-код; данные в формате JSON; графическое изображение).

Каждый из этих методов возвращает промис, подтверждаемый после успешного извлечения данных в затребованном формате. Нагрузкой промиса являются сами полученные данные.

А теперь — сами методы:

- `text()` — извлекает данные в виде текста. Применяется для получения фрагментов обычного текста и HTML-кода;
- `json()` — извлекает данные, записанные в формате JSON, которые уже представлены в виде служебного объекта, готового к обработке;
- `blob()` — извлекает двоичные данные. Применяется для получения данных, представленных в двоичном виде: графических изображений, аудио, видео и др.

Примеры:

```
// Загружаем фрагмент HTML-кода
const prmHTML = fetch('/fragments/fr5.html');
(async function () {
    // Получаем из промиса, возвращенного методом fetch(), объект ответа
    const oResponse = await prmHTML;
    // Если файл был успешно загружен, получаем и выводим его содержимое
    if (oResponse.ok) {
        const sFragment = await oResponse.text();
        oOutput.innerHTML = sFragment;
    }
})();

// Загружаем данные в формате JSON
const prmJSON = fetch('/programs/main-data/get-json-data.php');
(async function () {
    const oResponse = await prmJSON;
    // Если файл был успешно загружен...
    if (oResponse.ok) {
        const oData = await oResponse.json();
        // ...пускаем полученные данные в обработку
        . . .
    }
})();

// Загружаем картинку
(async function () {
    const oResponse = await fetch('/images/img3.jpg');
    if (oResponse.ok) {
        const oBlob = await oResponse.blob();
        // Преобразуем полученные двоичные данные изображения в data URL,
        // используя статический метод createObjectURL() класса URL
        const sURL = URL.createObjectURL(oBlob);
        // Выводим изображение
        oOutputImage.src = sURL;
    }
})();
```

6.3. Упражнение. Загружаем и выводим список языков программирования

Напишем страницу, которая загрузит из JSON-файла список популярнейших языков программирования в мире и выведет его в виде таблицы. Используем для этого новый загрузчик данных AJAX.

1. Найдем в папке 6\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы index.html (сама веб-страница), styles.css (таблица стилей) и languages.json (список языков программирования). Скопируем их куда-либо на локальный диск.

   Страница index.html содержит только заголовок и абзац поясняющего текста.

2. Откроем в текстовом редакторе копию файла languages.json и посмотрим на его содержимое:

   ```
   {
       "data": [
           { "number": 1, "name": "JavaScript" },
           { "number": 2, "name": "HTML и CSS" },
           . . .
           { "number": 7, "name": "C++" }
       ]
   }
   ```

   Свойство `data` хранит сам список языков в виде массива. Каждый из элементов этого массива является объектом, представляющим один язык и содержащим свойства `number` (порядковый номер языка в рейтинге) и `name` (название языка). Запомним это.

3. Откроем в текстовом редакторе копию файла index.html и запишем в «пустой» тег `<script>`, находящийся в конце HTML-кода, объявление асинхронного замыкания и, в его теле, выражения, загружающие файл languages.json и проверяющие, успешно ли он был загружен:

   ```
   <script type="text/javascript">
       (async function () {
           const oResponse = await fetch('languages.json');
           if (oResponse.ok) {
           } else {
           }
       })();
   </script>
   ```

   Получив данные, можно начинать создание таблицы, в которой будет выводиться рейтинг языков.

4. Запишем код, извлекающий загруженные данные, создающий саму таблицу, а в ней — строку шапки:

```
(async function () {
    const oResponse = await fetch('languages.json');
    if (oResponse.ok) {
        const oData = await oResponse.json();
        const oTable = document.createElement('table');
        let oRow = document.createElement('tr');
        let oCell = document.createElement('th');
        oCell.textContent = '№№';
        oRow.appendChild(oCell);
        oCell = document.createElement('th');
        oCell.textContent = 'Название языка';
        oRow.appendChild(oCell);
        oTable.appendChild(oRow);
    } else {
    }
})();
```

5. Добавим код, перебирающий массив языков из свойства `data` полученных JSON-данных и на основе каждого элемента этого массива создающий строку таблицы со сведениями о языке:

```
(async function () {
    . . .
    if (oResponse.ok) {
        . . .
        oTable.appendChild(oRow);
        for (const {number, name} of oData.data) {
            oRow = document.createElement('tr');
            oCell = document.createElement('td');
            oCell.textContent = number;
            oRow.appendChild(oCell);
            oCell = document.createElement('td');
            oCell.textContent = name;
            oRow.appendChild(oCell);
            oTable.appendChild(oRow);
        }
    } else {
    }
})();
```

В выражении цикла перебора мы применили деструктурирование объекта, чтобы немного упростить код.

6. Допишем выражение, добавляющее созданную таблицу в секцию тела страницы (тег `<body>`):

```
(async function () {
    . . .
    if (oResponse.ok) {
        . . .
```

```
        document.body.appendChild(oTable);
    } else {
    }
})();
```

7. Добавим код, выполняющийся в случае неудачной загрузки файла и выводящий сообщение об ошибке:

```
(async function () {
    . . .
    if (oResponse.ok) {
        . . .
    } else {
        const oP = document.createElement('p');
        oP.textContent = oResponse.status + ': ' +
                         oResponse.statusText;
        document.body.appendChild(oP);
    }
})();
```

Скопируем файлы index.html, styles.css и languages.json в корневую папку веб-сервера, запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/**. Страница покажет рейтинг языков программирования в виде таблицы (рис. 6.1).

Популярнейшие языки программирования

Рейтинг ресурса «Stack Overflow».

№№	Название языка
1	JavaScript
2	HTML и CSS
3	Python
4	Java
5	C#
6	PHP
7	C++

Рис. 6.1. Рейтинг языков программирования

Дадим файлу languages.json какое-либо другое имя и снова попробуем открыть страницу. На ней должно появиться сообщение о том, что файл не был найден.

6.4. Самостоятельные упражнения

Переделайте результаты выполнения *упражнений 2.2* и *3.7* с использованием загрузчика данных AJAX.

Урок 7
Встроенная СУБД

Базы данных. СУБД

Хранилища, документы, ключи, индексы, транзакции и запросы

Создание баз данных

Добавление, выборка, правка и удаление документов

Отладочные инструменты для работы с базами данных

Ранее для сохранения каких-либо данных на стороне клиента использовались сессионное и постоянное хранилища. В них можно было хранить лишь строковые величины, а процедура записи в хранилище (например, объекта) была весьма нетривиальной.

7.1. Введение во встроенную СУБД

Современные веб-обозреватели позволяют хранить на стороне клиента значения практически любых типов (включая объекты), записывая их в базы данных. Фактически они содержат достаточно развитую встроенную СУБД.

> *База данных* — организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации, преимущественно больших объемов.
>
> *Система управления базами данных* (СУБД) — комплекс программ, предназначенных для создания и ведения баз данных. Может быть реализована в виде независимого программного пакета или подсистемы, встроенной в другой пакет (например, веб-обозреватель).

Количество баз данных, создаваемых и обрабатываемых встроенной в веб-обозреватель СУБД, не ограничено. Каждая база данных может содержать сколько угодно хранилищ, в каждое хранилище можно записать сколько угодно документов.

> *Хранилище* — раздел базы данных, предназначенный для хранения информации, которая относится к определенной теме.
>
> *Документ* — отдельное значение, записанное в хранилище. Может принадлежать к любому типу (строковому, числовому и др.), однако чаще всего является объектом (что позволяет сохранить в одном документе произвольный набор значений, записав их в отдельные свойства). Каждый документ-объект может иметь произвольный набор свойств.

Каждому документу при сохранении присваивается ключ, по которому этот документ можно будет впоследствии найти и извлечь для обработки.

> *Ключ* — уникальное значение, однозначно идентифицирующее документ.

Ключ может:

- задаваться явно, при добавлении документа в хранилище, — и может принадлежать к любому типу.

  Как вариант, в качестве ключа может использоваться значение какого-либо свойства добавляемого документа;

- генерироваться самой СУБД — тогда его не нужно задавать при добавлении документа. Генерируемые автоматически ключи представляют собой последовательно увеличивающиеся целые числа.

Также в хранилище можно создать произвольное количество индексов по каким-либо свойствам документов.

> *Индекс* — компактный массив, содержащий упорядоченные значения свойства, на основе которого был создан этот индекс (*индексированного свойства*), и ключи документов, хранящих эти значения. Служит для поиска и фильтрации документов по значениям индексированного свойства.
> *Поиск* — выборка единственного документа, имеющего указанный ключ или хранящий заданное значение в индексированном свойстве (в последнем случае используется индекс, созданный на основе этого свойства).
>
> *Фильтрация* — выборка произвольного количества документов, хранящих заданное значение в индексированном свойстве. Для фильтрации всегда используется индекс.

Все операции с базами данных выполняются в транзакциях — это сделано, чтобы предотвратить повреждение баз при программных сбоях и внезапных отключениях электропитания.

> *Транзакция* — группа элементарных операций с базой данных (поиск, фильтрация, добавление, правка, удаление документов, создание, удаление хранилищ и индексов), выполняемых как единое целое. Если все операции, заключенные в транзакцию, выполняются успешно, транзакция *подтверждается*. Если какая-либо операция выполнилась с ошибкой, транзакция *откатывается* (или *отклоняется*), — и база данных возвращается в изначальное состояние, в котором она находилась перед началом выполнения этой транзакции.

Рассмотрим случай, когда требуется одновременно исправить какой-либо документ, находящийся в одном хранилище, и добавить другой документ в другое хранилище. Если добавить документ по какой-либо причине не удалось, транзакция откатывается, в результате чего исправления, внесенные в первый документ, аннулируются, и база данных окажется в изначальном состоянии.

Большинство операций с базами данных асинхронны. Это значит, что вызов метода, выполняющего нужную операцию, лишь запускает ее выполнение, сразу же возвращает в качестве результата запрос, после чего начинает исполняться следующий за вызовом метода код.

> *Запрос* — объект особого класса, представляющий асинхронную операцию с базой данных.

Как только операция будет выполнена, в объекте возвращенного ранее запроса возникает особое событие, в обработчике которого можно каким-либо образом отреагировать на завершение этой операции (например, получить выбираемые документы или сигнализировать посетителю об успешном добавлении нового документа).

> **ВНИМАНИЕ!**
>
> Большинство веб-обозревателей позволяют работать со встроенной СУБД только страницам, загруженным с веб-сервера. Исключением является лишь Google Chrome, предоставляющий доступ к СУБД также страницам, которые были загружены с локального диска.

7.2. Создание баз данных

Класс `Window` получил поддержку свойства `indexedDB`, хранящего объект класса `IDBFactory`, который представляет подсистему управления базами данных.

7.2.1. Создание и удаление самих баз данных

Класс `IDBFactory` предоставляет метод `open()`, который применяется и для создания новых баз данных, и для открытия уже существующих:

```
open(<имя базы данных>[, <версия базы данных>])
```

Имя базы данных задается в виде строки.

> **ВНИМАНИЕ!**
>
> Имена баз данных должны быть уникальны.

Необязательная для указания *версия базы данных* должна представлять собой целое число. Версию можно использовать, скажем, чтобы узнать, какой версией веб-приложения была создана база данных, и, если версии не совпадают, как-то отреагировать на это (добавить в старую базу данных новые хранилища или что-либо еще).

Метод возвращает запрос, представленный объектом класса `IDBOpenDBRequest`.

При вызове метода `open()` могут возникнуть следующие ситуации:

- база данных с заданным *именем* *не* существует — в этом случае она будет создана. Если *версия* не была указана, создаваемой базе дается версия 1.

  После создания «пустой» базы данных в объекте возвращенного запроса возникнет событие `upgradeneeded`. В его обработчике следует создать все хранилища, которые должны присутствовать в базе.

Класс `IDBOpenDBRequest` поддерживает свойство `result`, хранящее результат выполнения запроса, в нашем случае — только что созданную базу данных в виде объекта класса `IDBDatabase`. Этот класс предоставляет методы, в частности, и для создания хранилищ.

Получить сам объект запроса в обработчике события `upgradeneeded` можно, как обычно, из свойства `target` объекта события.

После успешного выполнения обработчика события `upgradeneeded` в объекте запроса возникнет событие `success`, сообщающее о том, что база данных успешно создана и открыта;

- база данных с заданным *именем* уже существует:
  - *версия базы данных* не указана — база данных с указанным *именем* будет открыта.

    После ее открытия в объекте запроса возникнет событие `success`. В обработчике этого события можно, например, прочитать хранящиеся в базе данные и вывести их на страницу.

    Получить саму базу данных можно из свойства `result` объекта запроса, а объект запроса — из свойства `target` объекта события;
  - *версия базы данных* указана, и она *совпадает* с записанной в базе данных — база открывается аналогичным образом;
  - *версия базы данных* указана, и она *больше*, чем записанная в базе — база данных с указанным *именем* будет открыта, и в нее будет записана указанная *версия*.

    После этого в объекте запроса возникнет событие `upgradeneeded`. В его обработчике можно выполнить обновление структуры базы до новой версии (добавить новые хранилища, удалить ненужные и др.).

    Обработчику события `upgradeneeded` передается в качестве параметра объект класса `IDBVersionChangeEvent`, производного от класса `Event`. Этот класс поддерживает два полезных свойства, доступные только для чтения:
    - `oldVersion` — номер предыдущей версии базы данных в виде целого числа. Если база данных ранее не существовала, хранит 0.

      Проверив значение этого свойства, можно узнать, обновляется база данных с предыдущей версии или создается «с нуля»;
    - `newVersion` — номер новой версии базы данных в виде целого числа.

    Как только обработчик события `upgradeneeded` выполнится без ошибок, в объекте запроса возникнет событие `success`;
  - *версия базы данных* указана, и она *меньше*, чем записанная в базе — база данных с указанным *именем* вообще не будет открыта.

Если при создании или открытии базы данных произошла ошибка, в объекте запроса возникнет событие `error`. В его обработчике можно каким-либо образом отреа-

гировать на ошибку. Объект исключения, описывающий ошибку, хранится в свойстве `error` класса `IDBOpenDBRequest`.

Пример кода, формирующего базу данных `addressbook` версии 2 (также обрабатывается ее открытие и обновление с предыдущей версии):

```
const oReq = indexedDB.open('addressbook', 2);
oReq.addEventListener('upgradeneeded', (evt) => {
    // Получаем базу данных
    const oDB = evt.target.result;
    if (evt.oldVersion == 0) {
        // База данных ранее не существовала.
        // Создаем необходимые хранилища.
    } else {
        // База данных существовала ранее и была обновлена до версии 2.
        // Создаем хранилища, которые должны присутствовать в новой
        // версии, и удаляем ненужные.
    }
});
oReq.addEventListener('success', (evt) => {
    // Получаем базу данных
    const oDB = evt.target.result;
    // Выполняем какие-либо действия с ней
});
oReq.addEventListener('error', (evt) => {
    // Получаем объект исключения, описывающий ошибку
    const oExc = evt.target.error;
    // Выполняем какие-либо действия
});
```

Извлечь список существующих баз данных позволяет метод `databases()` класса `IDBFactory`. Он возвращает промис, который после подтверждения получает в качестве нагрузки массив служебных объектов, каждый из которых представляет одну базу данных и имеет свойства `name` (имя базы данных) и `version` (номер ее версии в виде целого числа).

> ***Внимание!***
> Метод `databases()` в настоящее время не поддерживается Mozilla Firefox.

Пример:

```
(async function () {
    // Предварительно проверяем, поддерживается ли метод databases()
    if (indexedDB.databases)
        for (const oDB of await indexedDB.databases())
            console.log(oDB.name + ' v.' + oDB.version);
})();
```

Удаление ненужной базы данных с заданным *именем* выполняет метод `deleteDatabase(<имя базы данных>)`. В качестве результата он возвращает объект

запроса, аналогичный выдаваемому методом `open()` (см. ранее). Отследить успешное или неуспешное удаление базы данных можно, обрабатывая событие, соответственно, `success` или `error` этого запроса. Пример:

```
const oReq2 = indexedDB.deleteDatabase('addressbook');
oReq2.addEventListener('success', (evt) => {
    // База данных успешно удалена
});
```

Если удаляемая база данных не существует, ничего не произойдет.

Класс `IDBDatabase`, представляющий базу данных, поддерживает доступные только для чтения свойства `name` и `version`, хранящие, соответственно, имя и версию текущей базы данных. Пример:

```
oReq.addEventListener('upgradeneeded', (evt) => {
    const oDB = evt.target.result;
    const dbName = oDB.name;
    const dbVersion = oDB.version;
});
```

7.2.2. Создание и удаление хранилищ

Для создания нового хранилища в базе данных применяется метод `createObjectStore()` класса `IDBDatabase`:

```
createObjectStore(<имя хранилища>[, <параметры>])
```

Этот метод следует вызвать у той базы данных, в которой необходимо создать новое хранилище.

Имя хранилища задается в виде строки.

> **Внимание!**
>
> Имена хранилищ должны быть уникальны на уровне базы данных, в которой они находятся.

Если хранилище с заданным *именем* уже существует в базе данных, будет возбуждено исключение `ConstraintError`.

Необязательные *параметры* задаются в виде служебного объекта со следующими свойствами:

- `keyPath` — имя свойства документов, значение которого будет использоваться в качестве ключа очередного сохраняемого документа, в виде строки (в этом случае ключи будут храниться в составе документов). Если равно `null` или `undefined`, ключи будут храниться отдельно от документов. Значение по умолчанию — `undefined`;
- `autoIncrement` — если `true`, ключ очередного сохраняемого документа будет формироваться самой СУБД в виде уникального целого числа. Если `false`, ключ следует задавать явно при сохранении документа. Значение по умолчанию — `false`.

Если свойству `autoIncrement` дано значение `true` и одновременно указано свойство `keyPath`, автоматически сгенерированные ключи будут сохраняться в указанном свойстве документов.

Метод сразу же возвращает объект класса `IDBObjectStore`, представляющий созданное хранилище.

Метод `createObjectStore()` может быть вызван только в обработчике события `upgradeended` запроса. Если вызвать его в любом другом месте кода, будет возбуждено исключение `InvalidStateError`.

Пример создания в базе данных `addressbook` хранилища `phones` с автоматически генерируемыми ключами, хранящимися отдельно от документов:

```
const oReq3 = indexedDB.open('addressbook');
oReq3.addEventListener('upgradeneeded', (evt) => {
    const oDB = evt.target.result;
    const oPhones = oDB.createObjectStore('phones',
                                          { autoIncrement: true });
});
```

Пример создания аналогичного хранилища, в котором в качестве ключа будет использоваться значение свойства `name` сохраняемого документа:

```
const oPhones2 = oDB.createObjectStore('phones', { keyPath: 'name' });
```

Пример создания хранилища, в котором автоматически сгенерированные ключи будут сохраняться в свойстве `__key` документа:

```
const oPhones3 = oDB.createObjectStore('phones',
                            { keyPath: '__key', autoIncrement: true });
```

Получить массив имен хранилищ, имеющихся в базе данных, можно из доступного только для чтения свойства `objectStoreNames` класса `IDBDatabase`. Пример:

```
if (oDB.objectStoreNames.includes('addresses')) {
    // База данных содержит хранилище addresses
}
```

Удалить хранилище с заданным *именем* можно вызовом метода `deleteObjectStore(<имя хранилища>)` класса `IDBDatabase`. Метод удаляет хранилище немедленно и не возвращает результата. Пример:

```
oReq3.addEventListener('upgradeneeded', (evt) => {
    const oDB = evt.target.result;
    if (evt.oldVersion >= 0)
        oDB.deleteObjectStore('addresses');
});
```

Метод `deleteObjectStore()` может быть вызван только в обработчике события `upgradeended` запроса. Если вызвать его в любом другом месте кода, будет возбуждено исключение `InvalidStateError`. Если выполняется попытка удаления несуществующего хранилища, возбуждается исключение `NotFoundError`.

Класс `IDBObjectStore` поддерживает доступные только для чтения свойства `name`, `keyPath` и `autoIncrement`, хранящие, соответственно, имя текущего хранилища, значения свойств `keyPath` и `autoIncrement` конфигурационного объекта с параметрами, заданного при создании этого хранилища.

7.2.3. Создание индексов

Для создания нового индекса в хранилище служит метод `createIndex()` класса `IDBObjectStore`:

createIndex(<*имя индекса*>, <*имя индексированного свойства*>, <*параметры*>)

Этот метод следует вызвать у того хранилища, в котором необходимо создать новый индекс.

Имена индекса и *индексированного свойства*, на основе которого создается индекс, задаются в виде строк.

> **Внимание!**
> Имена индексов должны быть уникальны на уровне хранилища, в котором они созданы.

Если индекс с заданным *именем* уже существует, будет возбуждено исключение `ConstraintError`.

Параметры задаются в виде служебного объекта со свойствами:

- `unique` — если `true`, будет создан уникальный индекс, если `false` — обычный.

  > *Уникальный индекс* — позволяет хранить только уникальные значения индексированного свойства. При попытке добавить документ, в котором индексированное свойство хранит значение, содержащееся в том же свойстве какого-либо уже имеющегося в хранилище документа, возникнет ошибка.

- `multiEntry` — принимается во внимание, если индексированное свойство хранит массив.

  Если `true`, в индексе будут созданы записи, хранящие отдельные элементы из массива, содержащегося в индексированном свойстве. Если `false`, в индексе будет создана лишь одна запись, хранящая сам массив.

Метод сразу же возвращает объект класса `IDBIndex`, представляющий созданный индекс.

Метод `createIndex()` может быть вызван только в обработчике события `upgradeended` запроса. Если вызвать его в любом другом месте кода, будет возбуждено исключение `InvalidStateError`.

Пример создания хранилища `phones` с обычным индексом `name` и уникальным — `phone` на основе одноименных свойств документов:

```
const oPhones = oDB.createObjectStore('phones');
const oIndexName = oPhones.createIndex('name', 'name', { unique: false });
const oIndexPhone = oPhones.createIndex('phone', 'phone', { unique: true});
```

Получить массив имен индексов, имеющихся в хранилище, можно из доступного только для чтения свойства indexNames класса IDBObjectStore. Пример:

```
if (oPhones.indexNames.includes('phone')) {
    // Хранилище содержит индекс phone
}
```

Класс IDBIndex поддерживает доступные только для чтения свойства name, keyPath, unique и multiEntry, хранящие, соответственно, имя текущего индекса, имя индексированного свойства, значения свойств unique и multiEntry конфигурационного объекта с параметрами, заданного при создании этого индекса.

7.2.4. Создание и удаление индексов в существующих хранилищах

Создать индекс можно и в уже существующем хранилище. Далее показан пример создания в хранилище phones нового индекса type по одноименному свойству:

```
oReq3.addEventListener('upgradeneeded', (evt) => {
    const oDB = evt.target.result;
    // Получаем объект транзакции, в которой создается структура
    // базы данных, обратившись к свойству transaction запроса
    const oTr = evt.target.transaction;
    // Получаем хранилище, в котором нужно создать новый индекс, вызовом
    // метода objectStore(<имя хранилища>) транзакции
    const oPhones = oTr.objectStore('phones');
    // Создаем новый индекс
    oPhones.createIndex('type', 'type');
});
```

Также можно удалить уже существующий индекс с заданным *именем*, вызвав метод deleteIndex(<*имя индекса*>) класса IDBObjectStore. Пример:

```
const oPhones = oTr.objectStore('phones');
oPhones.deleteIndex('gender');
```

7.3. Запись и выборка данных

Все операции записи и выборки данных выполняются внутри транзакций.

7.3.1. Открытие базы данных

Открытие базы данных выполняется с помощью метода open() класса IDBFactory, подробно описанного в *разд. 7.2.1*. В этом случае следует указать имя гарантированно существующей базы данных — в противном случае будет создана новая база.

7.3.2. Добавление документов. Работа с транзакциями

Процесс добавления документов состоит из трех этапов: запуска новой транзакции, получения хранилища, в которое следует добавить документ, и собственно добавления документа.

7.3.2.1. Этап 1: запуск транзакции

Запуск новой транзакции выполняется вызовом метода `transaction()` класса `IDBDatabase`:

```
transaction(<хранилища>[, <режим транзакции>[, <параметры>]])
```

Этот метод следует вызвать у той базы данных, в которую будут записываться и (или) из которой будут выбираться данные.

В первом параметре метода задаются *хранилища*, с которыми будет выполняться работа. Здесь можно указать:

- имя одного хранилища в виде строки — если работа будет выполняться с одним хранилищем;
- массив с именами хранилищ, заданными в виде строк, — если планируется работа с несколькими хранилищами.

Следует указывать лишь те хранилища, с которыми реально будет выполняться работа. Задание слишком большого количества хранилищ снижает производительность.

В качестве *режима транзакции* следует указать одну из двух строковых величин:

- `'readonly'` — если планируется лишь извлекать данные (запись данных в этом случае невозможна);
- `'readwrite'` — если планируется и извлекать, и записывать данные.

  При работе в этом режиме снижается производительность, поэтому использовать его рекомендуется лишь при необходимости записать данные в базу.

Если режим транзакции не указан, используется `readonly`.

Необязательные *параметры* задаются в виде служебного объекта, содержащего одноименные параметрам свойства. В настоящее время поддерживается лишь параметр `durability`, который указывает политику записи данных на диск. Можно задать одно из трех возможных строковых значений:

- `'relaxed'` — веб-обозреватель отдает операционной системе команду записать добавленную в базу информацию на диск, *но не проверяет*, действительно ли она была записана. Обеспечивает более высокую производительность, но может привести к потере данных при внезапном закрытии веб-обозревателя. Рекомендуется к применению, если требуется часто записывать быстро изменяющиеся, но малоценные данные (например, кэшировать информацию, полученную с сервера);
- `'strict'` — веб-обозреватель отдает операционной системе команду записать добавленную в базу информацию на диск *и проверяет*, была ли она записана.

Обеспечивает большую надежность хранения данных за счет снижения производительности. Рекомендуется, если нужно хранить важные данные;

- `'default'` — веб-обозреватель сам выбирает политику записи. Рекомендуется в большинстве случаев.

По умолчанию используется `'default'`.

> ***Внимание!***
>
> Параметр `durability` в настоящее время не поддерживается Mozilla Firefox. Параметр можно указать, но Firefox его проигнорирует.

Метод `transaction()` возвращает объект класса `IDBTransaction`, представляющий только что созданную и запущенную транзакцию.

Класс `IDBTransaction` поддерживает доступные только для чтения свойства `db`, `objectStoreNames` и `mode`, хранящие, соответственно, объект текущей базы данных, массив с именами хранилищ, с которыми будет выполняться работа, и обозначение режима транзакции.

7.3.2.2. Этап 2: получение хранилища

Запустив транзакцию, следует получить из нее нужное хранилище. Для этого служит метод `objectStore()` класса `IDBTransaction`:

```
objectStore(<имя хранилища>)
```

Имя хранилища указывается в виде строки.

> ***Внимание!***
>
> Можно получить лишь одно из тех хранилищ, что были указаны при запуске транзакции, в вызове метода `transaction()` (*см. разд. 7.3.2.1*).

При попытке получить хранилище, не указанное при запуске транзакции, возникнет исключение `NotFoundError`.

Метод возвращает объект класса `IDBObjectStore`, представляющий полученное хранилище.

7.3.2.3. Этап 3: собственно добавление документов

Добавить новый документ в полученное ранее хранилище можно вызовом метода `add()` класса `IDBObjectStore`:

```
add(<добавляемый документ>[, <ключ>])
```

Ключ указывается только в том случае, если он должен задаваться явно (т. е. если он не генерируется автоматически и не берется из какого-либо свойства *добавляемого документа*).

Метод возвращает запрос, представляющий выполняемую операцию, в виде объекта класса `IDBRequest`. Обрабатывая событие `success` этого запроса, можно отследить момент успешного добавления документа, а обрабатывая событие `error` — возникшую при добавлении нештатную ситуацию.

Свойство `result` возвращенного объекта запроса хранит ключ вновь добавленного документа.

> **Внимание!**
>
> Встроенная в веб-обозреватель СУБД не требует, чтобы все заносимые в хранилище документы имели одинаковый набор свойств и содержали все индексированные свойства, на основе которых были созданы индексы.
>
> Однако, если добавляемый документ не содержит какое-либо индексированное свойство, он не попадет в соответствующий индекс (хотя будет успешно добавлен в хранилище и может быть позже извлечен из него по ключу или значению другого индексированного свойства, присутствующего в документе).

Метод `add()` в случае нештатной ситуации может возбудить одно из следующих исключений:

- `ConstraintError` — если индексированное свойство, по которому был создан уникальный индекс, в добавляемом документе хранит уже присутствующее в индексе значение;
- `DataError` — в случае если:
  - используются автоматически генерируемые ключи или ключи, хранящиеся в одном из свойств документов, — и *ключ* был явно указан в вызове метода `add()`;
  - используются ключи, хранящиеся в одном из свойств документов, — и это свойство не содержит никакого значения;
  - ключи должны задаваться явно — и *ключ* не был указан;
- `ReadOnlyError` — если запущенная транзакция работает в режиме `readonly`.

7.3.2.4. Подтверждение и откат транзакций

Транзакция:

- подтверждается (тогда все изменения, сделанные ей в базе данных, сохраняются):
  - автоматически — как только будет выполнен весь текущий код, и веб-обозреватель начнет простаивать, ожидая действий пользователя;
  - принудительно — при вызове метода `commit()`, поддерживаемого классом `IDBTransaction`, у объекта транзакции;
- откатывается (тогда все изменения, сделанные ей в базе данных, отменяются):
  - автоматически — при возникновении любого исключения во время добавления (выборки, правки или удаления) документа;
  - принудительно — при вызове метода `abort()`, поддерживаемого классом `IDBTransaction`, у объекта транзакции.

Класс `IDBTransaction` поддерживает следующие события:

- `complete` — возникает после успешного подтверждения транзакции, неважно, автоматического или принудительного;

- error — возникает после автоматического отката транзакции.

  Объект исключения, описывающий возникшую нештатную ситуацию, можно получить из свойства error класса IDBTransaction;
- abort — возникает после принудительного отката транзакции, а также в случае ошибки дискового ввода/вывода.

Пример добавления нового документа:

```
const oReq4 = indexedDB.open('addressbook');
oReq4.addEventListener('success', (evt) => {
    const oDB = evt.target.result;
    // Запускаем транзакцию
    const oTransaction = oDB.transaction('phones', 'readwrite')
    // Привязываем обработчики к событиям транзакции
    oTransaction.addEventListener('complete', (evt) => {
        console.log('Документ добавлен.');
    });
    oTransaction.addEventListener('error', (evt) => {
        console.log('Произошла ошибка: ' + evt.target.error);
    });
    // Получаем хранилище из запущенной транзакции
    const oPhones = oTransaction.objectStore('phones');
    // Добавляем документ
    const oNewDoc = { name: 'Вася Пупкин', phone: '322223322223' };
    oPhones.add(oNewDoc);
});
```

Также можно обрабатывать не события complete и error транзакции, а аналогичные события success и error запроса на добавление документа. В наиболее простых случаях так и поступают. Пример:

```
const oReq = oPhones.add(oNewDoc1);
oReq.addEventListener('success', (evt) => {
    console.log('Документ добавлен.');
});
oReq.addEventListener('error', (evt) => {
    console.log('При добавлении документа возникла ошибка: ' +
                evt.target.error);
});
```

В сложных случаях, если в транзакции выполняется несколько действий с базой данных (например, добавляются несколько документов), для отслеживания успешного выполнения каждого из действий можно обрабатывать события success и error и самой транзакции, и отдельных запросов. Пример:

```
const oTransaction = oDB.transaction('phones', 'readwrite')
oTransaction.addEventListener('complete', (evt) => {
    console.log('Оба документа добавлены.');
});
```

```
oTransaction.addEventListener('error', (evt) => {
    console.log('При добавлении документов возникла ошибка: ' +
                evt.target.error);
});
const oPhones = oTransaction.objectStore('phones');
. . .
const oReq1 = oPhones.add(oNewDoc1);
oReq1.addEventListener('success', (evt) => {
    console.log('Документ 1 добавлен.');
});
oReq1.addEventListener('error', (evt) => {
    console.log('При добавлении документа 1 возникла ошибка: ' +
                evt.target.error);
});
const oReq2 = oPhones.add(oNewDoc2);
oReq2.addEventListener('success', (evt) => {
    console.log('Документ 2 добавлен.');
});
oReq2.addEventListener('error', (evt) => {
    console.log('При добавлении документа 2 возникла ошибка: ' +
                evt.target.error);
});
```

7.2.3.5. Добавление начальных документов

Если в хранилище изначально должны присутствовать какие-либо документы, их следует добавить сразу после создания хранилища.

Операция создания хранилища выполняется в транзакции, автоматически запускаемой самим веб-обозревателем. Объект этой транзакции хранится в доступном только для чтения свойстве `transaction` класса `IDBObjectStore`, т. е. хранилища.

Перед тем как добавлять во вновь созданное хранилище начальные документы, следует дождаться подтверждения этой транзакции. Отследить момент подтверждения транзакции можно, обрабатывая ее событие `complete` (см. *разд. 7.3.2.4*).

Вот пример добавления одного начального документа сразу после создания хранилища `phones`:

```
const oPhones = oDB.createObjectStore('phones');
oPhones.transaction.addEventListener('complete', (evt) => {
    // Запускаем новую транзакцию и получаем из нее хранилище
    const oTransaction = oDB.transaction('phones', 'readwrite')
    const oPhones = oTransaction.objectStore('phones');
    // Добавляем начальный документ
    const oInitialDoc = { . . . };
    oPhones.add(oInitialDoc);
});
```

7.3.3. Поиск документов

При поиске из хранилища извлекается всего один документ, удовлетворяющий заданным условиям (например, записанный под указанным ключом).

> **Внимание!**
>
> При выполнении поиска документа, а также всех остальных описываемых далее операций предварительно следует запустить транзакцию и получить из нее хранилище, с которым будет производиться работа.

7.3.3.1. Поиск документа по ключу

Поиск документа по заданному *ключу* (поиск в хранилище) выполняет метод `get(<ключ>)` класса `IDBObjectStore`, который следует вызвать у того хранилища, в котором хранится искомый документ.

Метод возвращает объект класса `IDBRequest`, представляющий запрос на выполнение поиска. По завершении поиска свойство `result` объекта запроса будет хранить:

- если документ будет найден — сам найденный документ;
- если документ не будет найден — значение `undefined`.

Пример:

```
const oReq5 = indexedDB.open('addressbook');
oReq5.addEventListener('success', (evt) => {
    const oDB = evt.target.result;
    const oTransaction = oDB.transaction('phones')
    const oPhones = oTransaction.objectStore('phones');
    // Ищем документ с ключом 2
    const oReq = oPhones.get(2);
    oReq.addEventListener('success', (evt) => {
        const oDoc = evt.target.result;
        if (oDoc) {
            // Документ найден
        } else {
            // Документ не найден
        }
    });
});
```

7.3.3.2. Поиск документа по значению индексированного свойства

Если же требуется найти документ по значению индексированного свойства (выполнить поиск в индексе), следует выполнить следующие действия:

1. Получить индекс, созданный на основе нужного свойства, — вызовом метода `index(<имя индекса>)` класса `IDBObjectStore` у того хранилища, в котором хранится искомый документ. *Имя индекса* задается в виде строки.

   Метод вернет объект класса `IDBIndex`, представляющий затребованный индекс.

2. Найти документ — вызовом описанного в *разд. 7.3.3.1* метода `get(<искомое значение>)` у полученного ранее индекса.

Пример:

```
const oReq6 = indexedDB.open('addressbook');
oReq6.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones')
    const oPhones = oTransaction.objectStore('phones');
    // Ищем Васю Пупкина
    const oReq = oPhones.index('name').get('Вася Пупкин');
    oReq.addEventListener('success', (evt) => {
        const oDoc = evt.target.result;
        . . .
    });
});
```

> **Внимание!**
>
> Если существует несколько документов, хранящих в индексированном свойстве заданное значение, метод `get()` вернет самый первый из них.
>
> Не забываем, что документ, не содержащий индексированного свойства, не попадет в соответствующий индекс и, таким образом, не может быть найден в этом индексе. Тем не менее его все же можно найти по ключу или в другом индексе (если документ содержит индексированное свойство, по которому был создан этот индекс).

Класс `IDBIndex` также поддерживает метод `getKey(<искомое значение>)`. В отличие от метода `get()`, он выдает не сам найденный документ, а его ключ. Пример:

```
const oReq = oPhones.index('name').getKey('Вася Пупкин');
oReq.addEventListener('success', (evt) => {
    // Получаем ключ найденного документа
    const oDocKey = evt.target.result;
    // Используем его в работе
});
```

7.3.3.3. Поиск документа в диапазоне значений

Часто бывает необходимо найти документ по ключу или значению индексированного свойства, укладывающемуся в заданный диапазон.

Для указания таких критериев применяется класс `IDBKeyRange` и его статические методы, приведенные далее:

- `bound()` — задает поиск в диапазоне значений от *нижнего* до *верхнего* предела:

  ```
  bound(<нижний предел>, <верхний предел>[,
        <исключить нижний предел?>=false[,
        <исключить верхний предел?>=false]])
  ```

  По умолчанию *нижний* и *верхний пределы* входят в задаваемый диапазон. Если требуется исключить из диапазона *нижний предел*, следует дать параметру *исклю-*

чить нижний предел значение `true`, а чтобы исключить *верхний предел* — дать значение `true` параметру *исключить верхний предел*;

- `lowerBound(<нижний предел>[, <исключить?>=false])` — задает поиск в диапазоне, содержащем только *нижний предел*.

  По умолчанию *нижний предел* входит в создаваемый диапазон. Чтобы исключить его из диапазона, нужно дать параметру *исключить* значение `true`;

- `upperBound(<верхний предел>[, <исключить?>=false])` — задает поиск в диапазоне, содержащем только *верхний предел*.

  По умолчанию *верхний предел* входит в создаваемый диапазон. Чтобы исключить его из диапазона, нужно дать параметру *исключить* значение `true`;

- `only(<значение>)` — задает поиск по конкретному *значению*. Фактически бесполезен — искать документы по конкретному значению удобнее способами, описанными в *разд. 7.3.3.1* и *7.3.3.2*.

Все четыре метода возвращают объект класса `IDBKeyRange`, представляющий созданный диапазон значений.

Для выполнения поиска в заданном *диапазоне значений* применяется другой формат вызова метода `get()` — `get(<диапазон значений>)`.

Примеры:

```
// Ищем первый документ с ключом от 10 и выше исключительно
const oRange = IDBKeyRange.lowerBound(10, true);
const oReq = oPhones.get(oRange);

// Ищем первого из имеющихся в телефонной книге Кириллов.
// Чтобы указать верхнюю границу для строкового диапазона, мы добавили
// к имени «Кирилл» последнюю букву кириллицы — «я».
const oRange = IDBKeyRange.bound('Кирилл', 'Кирилля');
const oReq = oPhones.index('name').get(oRange);
```

Метод `getKey(<диапазон значений>)`, поддерживаемый классами `IDBObjectStore` и `IDBIndex`, в отличие от `get()`, выдает не сам найденный документ, а его ключ. Пример:

```
const oRange = IDBKeyRange.bound('Кирилл', 'Кирилля');
const oReq = oPhones.index('name').getKey(oRange);
oReq.addEventListener('success', (evt) => {
    // Получаем ключ первого найденного Кирилла
    const oDocKey = evt.target.result;
    // Используем его в работе
});
```

Класс `IDBKeyRange` поддерживает следующие свойства, доступные только для чтения:

- `lower` — нижняя граница текущего диапазона;
- `upper` — верхняя граница текущего диапазона.

Если диапазон был создан вызовом метода `only()`, оба этих свойства хранят одно и то же значение;

- `lowerOpen` — `false`, если нижняя граница входит в текущий диапазон, и `true` — в противном случае;
- `upperOpen` — `false`, если верхняя граница входит в текущий диапазон, и `true` — в противном случае.

Метод `includes(<значение>)` класса `IDBKeyRange` возвращает `true`, если заданное *значение* укладывается в текущий диапазон, и `false` — в противном случае.

7.3.4. Выборка и фильтрация документов

Можно выбрать все документы, записанные в каком-либо хранилище. Также можно выбрать лишь документы, отфильтрованные по заданным критериям.

7.3.4.1. Последовательная выборка документов. Курсоры

Как правило, документы выбираются последовательно, один за другим. Поскольку в оперативной памяти хранится лишь один документ — извлекаемый в текущий момент, это экономит системные ресурсы.

Последовательная выборка производится посредством курсора.

> *Курсор* — программный инструмент, позволяющий выбирать документы последовательно, один за другим, перемещаясь по хранилищу в заданном направлении (обычно от начала к концу).

Получить курсор можно вызовом метода `openCursor()`, который поддерживается классами `IDBObjectStore` и `IDBIndex`:

```
openCursor([<диапазон значений>=undefined[, <направление>='next']])
```

Будучи вызванным у хранилища, этот метод выдаст курсор, позволяющий выбирать документы в том порядке, в котором они записаны в хранилище, — т. е. упорядоченными по ключу (выборка документов непосредственно из хранилища). Если вызвать этот метод у индекса, документы будут выбираться упорядоченными по значению индексированного свойства, на основе которого был создан индекс (выборка документов из индекса).

Диапазон значений задается в виде объекта класса `IDBKeyRange` (см. *разд. 7.3.3.3*), при этом будут выбираться документы, чьи ключи (или значение индексированного свойства) укладываются в этот диапазон (т. е. будет выполняться фильтрация документов). Если *диапазон* не указывать, будут выбраны все документы.

В качестве *направления* перемещения по хранилищу (индексу) можно указать одно из следующих строковых значений:

- `'next'` — от начала к концу;
- `'nextunique'` — от начала к концу с выборкой документов с уникальными значениями индексированного свойства (в случае выборки документов из храни-

лица или уникального индекса указывать это направление не имеет смысла, поскольку ключи или значения индексированного свойства в этом случае, так или иначе, всегда уникальны);

- `'prev'` — от конца к началу;
- `'prevunique'` — от конца к началу с выборкой документов с уникальными значениями индексированного свойства (указывается только при выборке документов из неуникального индекса).

Метод `openCursor()` возвращает объект класса `IDBRequest`, представляющий запрос на получение курсора. После удачного выполнения этого запроса в его свойстве `result` будет храниться объект класса `IDBCursorWithValue`, представляющий созданный курсор. Если хранилище пусто, это свойство будет хранить значение `undefined`.

Класс `IDBCursorWithValue` поддерживает ряд доступных только для чтения свойств, хранящих очередной выбранный документ, его ключ и значение индексированного свойства:

- `value` — очередной выбранный документ;
- `key` — в зависимости от того, откуда выбираются документы:
  - непосредственно из хранилища — ключ документа;
  - из индекса — значение индексированного свойства документа;
- `primaryKey` — всегда — ключ документа;
- `direction` — направление выборки документов, заданное при создании курсора.

Для перемещения по хранилищу или индексу служат следующие методы класса `IDBCursorWithValue`:

- `continue()` — выполняет перемещение на следующий документ;
- `continue([<значение>])` — выполняет перемещение на документ, ключ которого совпадает с заданным *значением* или индексированное свойство которого хранит заданное *значение*;
- `continuePrimaryKey(<значение>, <ключ>)` — используется при выборке документов из индекса в том случае, если необходимо возобновить выборку после прерывания (если выполняется выборка из хранилища, проще воспользоваться методом `continue()`, вызванным с параметром).

  Метод выполняет перемещение на документ со значением индексированного свойства, совпадающим с заданным *значением*, и с указанным *ключом*;
- `advance(<количество>)` — выполняет перемещение на заданное *количество* документов.

Все эти методы дают текущему запросу указание переместиться на заданный документ. По завершении перемещения в текущем запросе снова возникнет событие `success`, в обработчике которого можно извлечь следующий документ.

Пример извлечения документов из хранилища:

```
const oReq7 = indexedDB.open('addressbook');
oReq7.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones');
    const oPhones = oTransaction.objectStore('phones');
    const oReq = oPhones.openCursor();
    oReq.addEventListener('success', (evt) => {
        const oC = evt.target.result;
        if (oC) {
            // Очередной документ получен
            // Извлекаем его ключ...
            const key = oC.key;
            // ...и содержимое
            const name = oC.value.name;
            const phone = oC.value.phone;
            // После чего перемещаемся на следующий документ
            oC.continue();
        } else {
            // Документов более нет
        }
    });
});
```

Пример извлечения документов из индекса:

```
const oReq = oPhones.index('name').openCursor();
oReq.addEventListener('success', (evt) => {
    const oC = evt.target.result;
    if (oC) {
        // Извлекаем ключ очередного документа...
        const key = oC.primaryKey;
        // Значение индексированного свойства можно извлечь
        // из свойства key запроса
        const name = oC.key;
        const phone = oC.value.phone;
        oC.continue();
    } else {
        ...
    }
});
```

Классы `IDBObjectStore` и `IDBIndex` также поддерживают метод `openKeyCursor()`, полностью аналогичный методу `openCursor()`. Различие лишь в том, что метод `openKeyCursor()` выдает объект класса `IDBCursor`, поддерживающий все свойства класса `IDBCursorWithValue`, за исключением свойства `value`, и, таким образом, не способный выдать сам документ (собственно, класс `IDBCursorWithValue` является производным от класса `IDBCursor` и добавляет поддержку упомянутого ранее свой-

ства). Этот метод может пригодиться, если нужно извлекать лишь ключи документов или значения индексированных свойств. Пример:

```
const oReq = oPhones.index('name').openKeyCursor();
oReq.addEventListener('success', (evt) => {
    const oC = evt.target.result;
    if (oC) {
        // Извлекаем имя абонента из очередного документа
        const name = oC.key;
        oC.continue();
    } else {
        ...
    }
});
```

7.3.4.2. Одновременная выборка всех документов

Также есть возможность одновременно выбрать все документы из хранилища или индекса в виде обычного массива.

Для одновременной выборки документов предназначен метод `getAll()`, поддерживаемый классами `IDBObjectStore` и `IDBIndex`:

```
getAll([<диапазон значений>=undefined[, <количество документов>=2^32-1]])
```

Диапазон значений задается в виде объекта класса `IDBKeyRange` (см. *разд. 7.3.3.3*). Если его не указывать, будут выбраны все документы.

Можно также указать *количество* выбираемых документов. Если оно не задано, будут извлечены все документы.

Метод возвращает объект класса `IDBRequest`, представляющий запрос на получение документов. После удачного выполнения этого запроса в его свойстве `result` будет храниться массив со всеми выбранными документами.

Пример выборки всех Кириллов:

```
const oRange = IDBKeyRange.bound('Кирилл', 'Кирилля');
const oReq = oPhones.index('names').getAll(oRange);
oReq.addEventListener('success', (evt) => {
    // Получаем массив с Кириллами
    const oDocs = evt.target.result;
    // Используем его в работе
});
```

Этот метод рекомендуется применять только для выборки небольшого количества документов малого объема. В противном случае выбранные документы займут слишком много оперативной памяти.

Аналогичный метод `getAllKeys()` выдает массив ключей выбранных документов.

7.3.5. Получение количества документов

Получить количество документов, содержащихся в хранилище, можно вызовом метода `count()` класса `IDBObjectStore`:

```
count([<диапазон значений>=undefined])
```

Если *диапазон значений* не указан, будет выдано общее количество документов.

Метод возвращает объект класса `IDBRequest`, представляющий запрос на получение количества документов. После удачного выполнения этого запроса в его свойстве `result` будет храниться количество документов.

Пример получения общего количества документов:

```
const oReq8 = indexedDB.open('addressbook');
oReq8.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones');
    const oPhones = oTransaction.objectStore('phones');
    const oReq = oPhones.count();
    oReq.addEventListener('success', (evt) => {
        // Извлекаем количество документов
        const count = evt.target.result;
        // Используем его в работе
    });
});
```

Пример получения количества всех Кириллов:

```
const oRange = IDBKeyRange.bound('Кирилл', 'Кирилля');
const oReq = oPhones.count(oRange);
oReq.addEventListener('success', (evt) => {
    const count = evt.target.result;
    . . .
});
```

7.3.6. Правка документов

7.3.6.1. Правка произвольного документа

Правка уже имеющегося в хранилище документа выполняется путем его замены новым документом, содержащим нужные исправления. Для этого применяется метод `put()` класса `IDBObjectStore`:

```
put(<исправленный документ>[, <ключ>])
```

Ключ указывается лишь в том случае, если он не хранится в одном из свойств самого *исправляемого документа*.

> **ВНИМАНИЕ!**
> Если указать несуществующий ключ, будет создан новый документ.

Метод возвращает объект класса `IDBRequest`, представляющий запрос на правку документа. После удачного выполнения этого запроса в его свойстве `result` будет храниться ключ исправленного документа.

Метод `put()` возбуждает те же исключения, что и метод `add()` (см. *разд. 7.3.2.4*), и в тех же случаях.

Пример замены телефона в документе с ключом 2:

```
const oReq9 = indexedDB.open('addressbook');
oReq9.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones',
                                                       'readwrite');
    const oPhones = oTransaction.objectStore('phones');
    const oReq = oPhones.get(2);
    oReq.addEventListener('success', (evt) => {
        const oDoc = evt.target.result;
        oDoc.phone = '322223223322';
        const oReq = oPhones.put(oDoc, 2);
        oReq.addEventListener('success', (evt) => {
            // Документ исправлен
        };
    });
});
```

7.3.6.2. Правка документов в процессе их выборки

Еще можно править документы в процессе их последовательной выборки с помощью курсора (см. *разд. 7.3.4.1*).

Правка документа, выбираемого в настоящий момент, выполняется вызовом метода `update(<исправленный документ>)` класса `IDBCursor`. Этот метод возвращает объект класса `IDBRequest`, представляющий запрос на правку документа. После удачного выполнения этого запроса в его свойстве `result` будет храниться ключ исправленного документа.

Пример:

```
const oReq10 = indexedDB.open('addressbook');
oReq10.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones',
                                                       'readwrite');
    const oPhones = oTransaction.objectStore('phones');
    // Перебираем телефонные номера и добавляем в каждый пометку
    // «домашний»
    const oPCursor = oPhones.openCursor();
    oPCursor.addEventListener('success', (evt) => {
        const oC = evt.target.result;
        if (oC) {
            const oDoc = oC.value;
            oDoc.type = 'Домашний';
```

```
            oC.update(oDoc);
            oC.continue();
        }
    });
});
```

7.3.7. Удаление документов

7.3.7.1. Удаление произвольных документов

Для удаления документов из хранилища предназначен метод `delete()` класса `IDBObjectStore`:

`delete(<ключ>|<диапазон ключей>)`

Чтобы удалить один документ, следует указать его *ключ*. Также можно удалить сразу несколько документов, задав *диапазон ключей*.

Метод возвращает объект класса `IDBRequest`, представляющий запрос на удаление документа.

Пример удаления документа с ключом 9:

```
const oReq11 = indexedDB.open('addressbook');
oReq11.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones', 'readwrite');
    const oPhones = oTransaction.objectStore('phones');
    const oReq = oPhones.delete(9);
    oReq.addEventListener('success', (evt) => {
        // Документ удален
    });
});
```

7.3.7.2. Удаление документов в процессе их выборки

Также можно удалять документы в процессе их последовательной выборки с помощью курсора (см. *разд. 7.3.4.1*).

Удаление документа, выбираемого в настоящий момент, выполняется вызовом метода `delete()` класса `IDBCursor`. Этот метод возвращает объект класса `IDBRequest`, представляющий запрос на удаление документа.

Пример:

```
const oReq12 = indexedDB.open('addressbook');
oReq12.addEventListener('success', (evt) => {
    const oTransaction = evt.target.result.transaction('phones',
                                                      'readwrite');
    const oPhones = oTransaction.objectStore('phones');
    // Перебираем телефонные номера и удаляем «запасные»
    const oPCursor = oPhones.openCursor();
    oPCursor.addEventListener('success', (evt) => {
        const oC = evt.target.result;
```

```
        if (oC) {
            if (oC.value.type == 'Запасной')
                oC.delete();
            oC.continue();
        }
    });
});
```

7.3.8. Закрытие базы данных

Чтобы сэкономить системные ресурсы, рекомендуется закрывать базу данных, когда она не используется. Закрытие базы данных выполняется вызовом метода `close()` класса `IDBDatabase`.

Перед закрытием базы данных принудительно выполняются все еще не выполненные транзакции. После закрытия ни считать данные из базы, ни записать их в нее уже не получится — придется открывать ее заново.

7.4. Упражнение. Пишем веб-приложение — телефонную книгу

Напишем веб-приложение телефонной книги, которая будет хранить имена, фамилии и телефоны.

Создадим для этого базу данных с единственным хранилищем, в котором и будет храниться телефонная книга. Каждый документ будет хранить сведения об отдельном абоненте и содержать фамилию, имя и телефон. Приложение позволит добавлять новых абонентов, править и удалять уже существующих (позже, выполняя самостоятельное упражнение, мы добавим в нее возможности фильтрации абонентов по начальным символам их фамилий).

Детали реализации обсудим в процессе работы.

1. Найдем в папке 7\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы index.html (веб-страница, которая станет интерфейсом веб-приложения) и styles.css (таблица стилей). Скопируем их куда-либо на локальный диск.

   Страница index.html содержит:

   - таблицу, секция основного содержания (тег `<tbody>`) которой имеет якорь `table_body`, — в этой секции будет выводиться телефонная книга;
   - веб-форму с якорем `enter` — для занесения новых телефонных номеров и правки уже существующих.

     В этой веб-форме находятся поля ввода `txtName1`, `txtName2` и `txtPhone` для занесения, соответственно, фамилии, имени и телефона абонента, а также кнопка отправки данных **Сохранить**.

Также на этой странице находится веб-форма search, но она нам понадобится позже.

Программный код веб-приложения мы поместим в два файла:

- base.js — код модели, которую мы реализуем в виде класса DB;
- ui.js — код контроллера.

> *Модель* — программный код, непосредственно работающий с каким-либо хранилищем данных (например, встроенной в веб-обозреватель СУБД). Обычно реализуется в виде класса.
>
> *Контроллер* — программный код, реализующий взаимодействие с пользователем. Контроллер обращается к хранилищу данных посредством модели.

Разделение кода на модель и контроллер позволит упростить программирование и сделать код универсальным (например, можно написать универсальную модель и использовать ее с разными контроллерами).

Тег <script>, привязывающий к странице файл base.js с кодом модели, поместим в самом начале страницы, в ее секции заголовка (в теге <head>). Так модель будет загружена и обработана в самую первую очередь.

2. Откроем в текстовом редакторе копию файла index.html и вставим в секцию заголовка страницы тег <script>, привязывающий файл base.js:

```
<html>
    <head>
        . . .
        <link href="styles.css" rel="stylesheet" type="text/css">
        <script src="base.js" type="text/javascript"></script>
    </head>
    <body>
        . . .
    </body>
</html>
```

А привязку файла ui.js поместим в самый конец HTML-кода страницы. Поскольку контроллер напрямую взаимодействует с элементами страницы, его следует загружать и обрабатывать только тогда, когда вся страница уже загружена, и все ее элементы сформированы.

3. Вставим в самом конце кода страницы тег <script>, привязывающий файл ui.js:

```
<html>
    . . .
</html>
<script src="ui.js" type="text/javascript"></script>
```

4. Создадим файл base.js в той же папке, в которой сохранен файл index.html, и откроем его в текстовом редакторе.

Класс DB, реализующий нашу модель, будет иметь только статические методы. Благодаря этому нам не придется создавать объект этого класса и расходовать под его хранение оперативную память.

Объявляя класс модели DB, сразу же добавим в него служебный метод processRequest(<*запрос*>). Этот метод будет возвращать промис, который подтвердится в случае выполнения указанного *запроса* на выполнение операции с базой данных и получит в качестве нагрузки результат его выполнения (которым может быть открытая база данных, список выбранных из хранилища документов, отдельный выбранный документ и т. п.).

Обрабатывать запросы подобным образом придется при выполнении практически каждого действия с базой данных. Поэтому имеет смысл вынести код, производящий такую обработку, в отдельный метод, что заметно сократит код модели.

5. Запишем в файл base.js объявление класса DB со статическим методом processRequest():

```
class DB {
    static processRequest(req) {
        return new Promise((resolve, reject) => {
            req.addEventListener('success', (evt) => {
                resolve(evt.target.result);
            });
            req.addEventListener('error', (evt) => {
                reject(evt.target.error);
            });
        });
    }
}
```

Далее объявим метод open(), который создаст базу данных, если она еще не существует, и откроет ее. Поскольку создание и открытие базы данных — операции асинхронные, результат метод вернет в виде промиса, получающего в качестве нагрузки открытую базу данных.

Условимся назвать нашу базу данных addressbook и дать ей номер версии 1. В ней создадим хранилище phones, в котором и будет содержаться телефонная книга. Каждый документ будет иметь свойства name1 (фамилия), name2 (имя) и phone (номер телефона). Ключи документов будем генерировать автоматически и хранить отдельно от самих документов. Также создадим на будущее обычный индекс name по свойству name1 (в котором будет храниться фамилия абонента).

6. Добавим в класс DB статический метод open():

```
class DB {
    . . .
    static open() {
        const oReq = indexedDB.open('addressbook', 1);
```

```
        oReq.addEventListener('upgradeneeded', (evt) => {
            const oDB = evt.target.result;
            const oPhones = oDB.createObjectStore('phones',
                                    { autoIncrement: true });
            oPhones.createIndex('name', 'name1', { unique: false });
        });
        return this.processRequest(oReq);
    }
}
```

Для обработки запроса на открытие базы данных мы применили объявленный ранее служебный метод `processRequest()`.

Одна из целей объявления модели — максимально упростить работу с хранилищем данных. В идеале, модель должна выдавать данные, уже преобразованные в удобный для обработки вид, и не требовать взаимодействия непосредственно с хранилищем данных.

Стремясь достичь заявленных ранее целей, добавим в класс `DB` два статических метода:

- `getKeys()` — возвращает массив с ключами всех имеющихся в хранилище документов;
- `get(`*<ключ>*`)` — возвращает документ с заданным *ключом*.

7. Добавим в класс `DB` статические методы `getKeys()` и `get()`:

```
class DB {
    . . .
    static async getKeys() {
        const oDB = await this.open();
        const oReq = oDB.transaction('phones')
                        .objectStore('phones')
                        .getAllKeys();
        oDB.close();
        return this.processRequest(oReq);
    }

    static async get(key) {
        const oDB = await this.open();
        const oReq = oDB.transaction('phones')
                        .objectStore('phones')
                        .get(key);
        oDB.close();
        return this.processRequest(oReq);
    }
}
```

В теле каждого из этих методов получаем базу данных (вызовом объявленного ранее метода `open()`), запускаем транзакцию в режиме `readonly`, получаем хранилище `phones` и, собственно, начинаем выполнять нужную операцию, вызвав

соответствующий метод хранилища — `getAllKeys()` или `get()`. Далее, получив запрос на выполнение операции, сразу же указываем закрыть базу данных, как только запущенная операция будет выполнена (так мы сэкономим системные ресурсы). И наконец, обрабатываем запрос, использовав метод `processRequest()`, объявленный ранее.

Осталось добавить в класс еще три статических метода:

- `add(`*<документ>*`)` — добавляет в хранилище заданный *документ* и выдает его ключ;
- `update(`*<исправленный документ>*`, `*<ключ>*`)` — перезаписывает в хранилище *исправленный документ* под заданным *ключом*;
- `delete(`*<ключ>*`)` — удаляет из хранилища документ с заданным *ключом*.

8. Добавим в класс `DB` статические методы `add()`, `update()` и `delete()`:

```
class DB {
    . . .
    static async add(doc) {
        const oDB = await this.open();
        const oReq = oDB.transaction('phones', 'readwrite')
                        .objectStore('phones')
                        .add(doc);
        oDB.close();
        return this.processRequest(oReq);
    }

    static async update(doc, key) {
        const oDB = await this.open();
        const oReq = oDB.transaction('phones', 'readwrite')
                        .objectStore('phones')
                        .put(doc, key);
        oDB.close();
        return this.processRequest(oReq);
    }

    static async delete(key) {
        const oDB = await this.open();
        const oReq = oDB.transaction('phones', 'readwrite')
                        .objectStore('phones')
                        .delete(key);
        oDB.close();
        return this.processRequest(oReq);
    }
}
```

Не забываем запускать транзакцию в режиме `readwrite`, иначе никаких изменений в базу данных внести не сможем.

С классом модели закончили. Начнем писать код контроллера.

9. Создадим файл ui.js в той же папке, в которой сохранен файл index.html, и откроем его в текстовом редакторе.

   Сначала необходимо объявить переменные и сохранить в них ссылки на объекты, представляющие секцию основного содержания таблицы, в которой будет выводиться адресная книга, поля ввода для занесения фамилии, имени и телефона абонента, а также на саму веб-форму, применяемую для работы с абонентами.

   Также следует создать переменную `docKey` для хранения ключа документа. Если в этой переменной хранится ключ, значит, в веб-форме выполняется правка уже имеющегося в базе данных документа, и при нажатии кнопки отправки этот документ следует перезаписать. Если же в переменной хранится `undefined`, то в веб-форму заносится новый абонент, и его следует добавить в базу.

10. Запишем в файл ui.js код, объявляющий необходимые переменные:

```
const oTableBody = document.getElementById('table_body');
const txtName1 = document.getElementById('txtName1');
const txtName2 = document.getElementById('txtName2');
const txtPhone = document.getElementById('txtPhone');
const frmEnter = document.getElementById('enter');

let docKey = undefined;
```

   Объявим функцию `showPhones()`, выводящую содержание телефонной книги в секцию основного содержания таблицы `table_body`. Поскольку эта функция будет выполнять асинхронные операции по взаимодействию с базой данных (посредством ранее написанной модели `DB`), сделаем эту функцию асинхронной.

   Содержание телефонной книги будет выводиться в пять столбцов: фамилия, абонента, его имя, телефон, гиперссылки **Исправить** и **Удалить**.

11. Добавим объявление функции `showPhones()`:

```
async function showPhones() {
    oTableBody.innerHTML = '';
    let oTR, oTD, oA;
    for (const key of await DB.getKeys()) {
        const doc = await DB.get(key);
        oTR = document.createElement('tr');
        oTD = document.createElement('td');
        oTD.textContent = doc.name1;
        oTR.appendChild(oTD);
        oTD = document.createElement('td');
        oTD.textContent = doc.name2;
        oTR.appendChild(oTD);
        oTD = document.createElement('td');
        oTD.textContent = doc.phone;
        oTR.appendChild(oTD);
```

```
        oTD = document.createElement('td');
        oA = document.createElement('a');
        oA.textContent = 'Исправить';
        oA.href = '#';
        oA.docKey = key;
        oA.addEventListener('click', editPhone);
        oTD.appendChild(oA);
        oTR.appendChild(oTD);
        oTD = document.createElement('td');
        oA = document.createElement('a');
        oA.textContent = 'Удалить';
        oA.href = '#';
        oA.docKey = key;
        oA.addEventListener('click', deletePhone);
        oTD.appendChild(oA);
        oTR.appendChild(oTD);
        oTableBody.appendChild(oTR);
    }
}
```

Здесь программно формируются, одна за другой, строки таблицы, в них добавляются ячейки, в которые помещаются сведения об очередном абоненте из телефонной книги. Единственный нюанс: в объектах, представляющих гиперссылки **Исправить** и **Удалить**, создается свойство `docKey`, хранящее ключ текущего документа, — он понадобится в дальнейшем.

К каждой гиперссылке **Исправить** в качестве обработчика события `click` привязывается функция `editPhone()`, а к каждой гиперссылке **Удалить** — функция `deletePhone()`. Эти функции мы скоро объявим.

После сохранения добавленного или исправленного абонента следует очистить веб-форму и занести в переменную `docKey` значение `undefined`, тем самым подготовив приложение для добавления нового абонента. Этим будет заниматься функция `clearForm()`.

12. Добавим объявление функции `clearForm()`:

```
function clearForm() {
    docKey = undefined;
    txtName1.value = '';
    txtName2.value = '';
    txtPhone.value = '';
}
```

Настала пора заняться функцией `editPhone()` — обработчиком событий `click` гиперссылок **Исправить**. Она извлечет документ с ключом, записанным в свойстве `docKey` гиперссылки, присвоит ключ переменной `docKey` и занесет содержимое документа в поля ввода веб-формы, подготовив сведения об абоненте к правке. Поскольку операция получения документа по ключу является асинхронной, саму функцию тоже следует сделать асинхронной.

13. Добавим объявление функции `editPhone()`:

```
async function editPhone(evt) {
    evt.preventDefault();
    docKey = evt.target.docKey;
    const doc = await DB.get(docKey);
    txtName1.value = doc.name1;
    txtName2.value = doc.name2;
    txtPhone.value = doc.phone;
}
```

Функция `deletePhone()` — обработчик событий `click` гиперссылок **Удалить** удалит документ с ключом из свойства `docKey` гиперссылки, на всякий случай очистит веб-форму и повторно выведет телефонную книгу на экран.

14. Добавим объявление функции `deletePhone()`:

```
async function deletePhone(evt) {
    evt.preventDefault();
    await DB.delete(evt.target.docKey);
    clearForm();
    showPhones();
}
```

Теперь нужно реализовать сохранение нового или исправленного абонента. Для этого привяжем обработчик к событию `submit` веб-формы. В обработчике сформируем новый документ на основе данных, занесенных в поля ввода, и проверим значение переменной `docKey`. Если переменная хранит ключ документа (т. е. выполняется правка существующего абонента), перезапишем старый документ новым, в противном случае (если вводится новый абонент) — добавим новый документ. После этого очистим веб-форму, подготовив ее к вводу нового абонента, и повторно выведем телефонную книгу.

15. Добавим обработчик события `submit` веб-формы:

```
frmEnter.addEventListener('submit', (evt) => {
    evt.preventDefault();
    const doc = {
        name1: txtName1.value,
        name2: txtName2.value,
        phone: txtPhone.value
    };
    if (docKey)
        DB.update(doc, docKey);
    else
        DB.add(doc);
    clearForm();
    showPhones();
});
```

Осталось сделать так, чтобы сразу после открытия страницы index.html на ней выводилась телефонная книга.

16. Добавим вызов функции `showPhones()`:

    ```
    showPhones();
    ```

    Скопируем файлы index.html, styles.css, base.js и ui.js в корневую папку веб-сервера, запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/**.

    Введем несколько абонентов, последовательно занося сведения о них в расположенную внизу страницы веб-форму и нажимая кнопку **Сохранить**. Попробуем исправить какого-либо абонента, щелкнув на расположенной в соответствующей строке таблицы гиперссылке **Исправить**, исправив сведения об этом абоненте в веб-форме и нажав кнопку **Сохранить**. Наконец, удалим какого-либо абонента щелчком на гиперссылке **Удалить**. У нас получится то, что показано на рис. 7.1.

Найти [ ] Вперед

Фамилия	Имя	Телефон		
Иванов	Иван	9705687546	Исправить	Удалить
Петров	Петр	456839545	Исправить	Удалить
Григорьев	Григорий	76564976	Исправить	Удалить
Борисов	Борис	5479046587	Исправить	Удалить
Александров	Александр	6554796435	Исправить	Удалить
Сидоров	Сидор	37506784667	Исправить	Удалить

Фамилия [ ]
Имя [ ]
Телефон [ ]
Сохранить

Рис. 7.1. Веб-приложение телефонной книги

Веб-форма поиска абонента по его фамилии, расположенная вверху страницы, пока не работает. Мы заставим ее работать позже.

7.5. Отладочные инструменты для работы с базами данных

В составе отладочных инструментов, встроенных в веб-обозреватель, присутствуют средства, позволяющие просматривать и удалять созданные к настоящему времени базы данных, хранилища, индексы и документы в них.

Чтобы добраться до этих средств, следует вывести панель с отладочными инструментами, нажав клавишу <F12>, переключиться на вкладку **Application** этой панели и найти в области **Storage** иерархического списка, расположенного в левой части панели, ветвь **IndexedDB** — там будут выводиться все существующие базы данных.

- Непосредственно в ветвь **IndexedDB** вложены пункты, представляющие базы данных.

Если выбрать базу данных, правее иерархического списка будут выведены сведения о ней (рис. 7.2). Они включают в себя:

- имя базы данных (выводится вверху увеличенным шрифтом);
- **Security origin** — интернет-адрес хоста, с которого была загружена страница. Если указано **file://** — страница была загружена с локального диска (как на рис. 7.2);
- **Version** — версия базы данных;
- **Object stores** — количество хранилищ в базе данных.

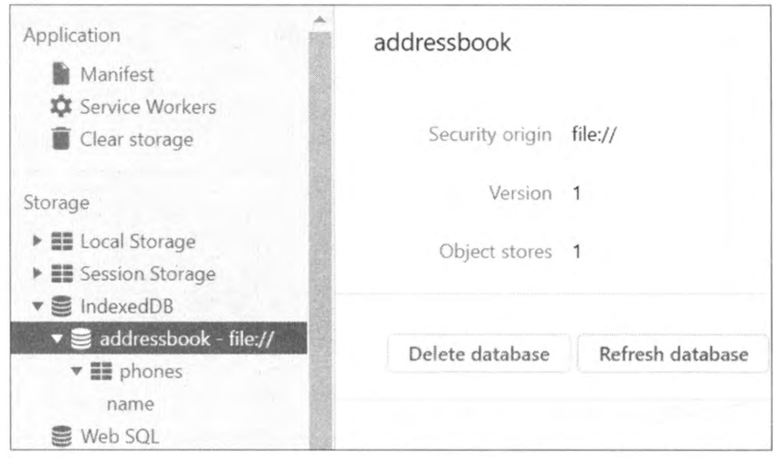


Рис. 7.2. Сведения о выбранной базе данных

♦ В пункт с именем базы данных (в нашем случае: **addressbook – file://**) вложены пункты, представляющие хранилища.

Если выбрать хранилище (в нашем случае: **phones**), правее появится список документов, записанных в этом хранилище и отсортированных по ключам (рис. 7.3). Он организован в виде таблицы со следующими столбцами:

- **#** — порядковый номер документа;
- **Key** — ключ документа;
- **Value** — содержание документа.

В левом верхнем углу ячейки с содержимым документа присутствует небольшая черная стрелка, направленная вправо. Щелкнув на ней, можно развернуть содержимое документа, приведя его к более удобному для восприятия виду (на рис. 7.3 таким образом развернут документ № 3 с ключом 11).

Для свертывания документа достаточно щелкнуть на той же стрелке (которая после развертывания станет указывать вниз).

Для развертывания документа также можно щелкнуть на нем правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Expand Recursively**. Выбрав пункт **Collapse** в том же меню, можно свернуть развернутый документ.

Рис. 7.3. Сведения о выбранном хранилище

Над списком документов находится поле ввода. Если ввести в него какой-либо ключ, в списке будут показаны только документы с ключами, начинающимися с введенного. Так, если ввести ключ 10, будут выведены документы с ключами 10, 11, 12 и 13.

♦ В пункт с именем хранилища вложены пункты, представляющие отдельные индексы.

  Если выбрать индекс (в нашем случае: **name**), правее появится список документов, содержащихся в этом индексе и отсортированных по значениям индексированного свойства (рис. 7.4). Он организован в виде таблицы со следующими столбцами:

  - **#** — порядковый номер документа;
  - **Key** — значение индексированного свойства;
  - **Primary Key** — ключ документа;
  - **Value** — содержание документа.

  В остальном этот список аналогичен описанному ранее.

С базами данных и их содержимым можно производить следующие операции:

♦ обновление:

  - конкретного хранилища — выбрав в левом иерархическом списке хранилище и щелкнув на расположенной выше списка документов кнопке **Refresh** (⟳).

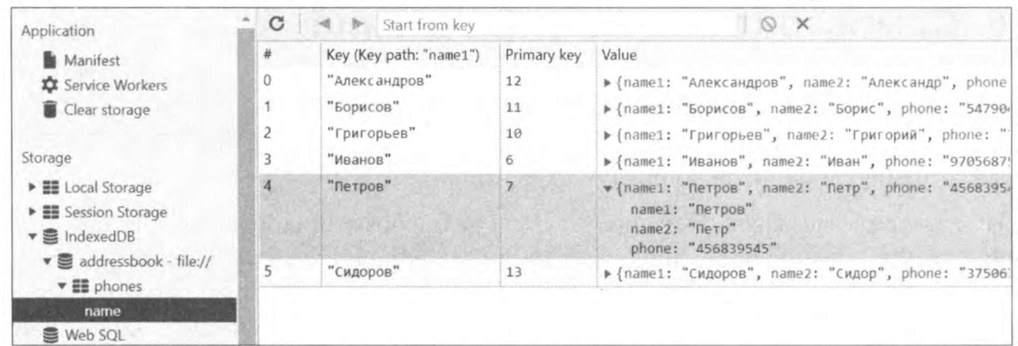


Рис. 7.4. Сведения о выбранном индексе

Также можно щелкнуть на списке документов правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Refresh**;

- конкретной базы данных — выбрав в левом иерархическом списке базу данных и щелкнув на расположенной справа кнопке **Refresh database** (см. рис. 7.2);
- всех баз данных — щелкнув правой кнопкой мыши на базе данных и выбрав в контекстном меню пункт **Refresh Indexed DB**;

Внимание!

После изменения структуры базы данных, добавления, правки и удаления документов ее содержимое, показанное в панели отладочных инструментов, не обновляется автоматически. Его следует обновить вручную.

- удаление:
  - отдельного документа — переключившись на нужное хранилище или индекс, выбрав удаляемый документ и щелкнув на расположенной выше списка документов кнопке **Delete selected** (✕).

    Также можно щелкнуть на удаляемом документе правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Delete**;
  - всех документов из хранилища — переключившись на нужное хранилище или индекс и щелкнув на расположенной выше списка документов кнопке **Clear object store** (⃠).

    Также можно щелкнуть на пункте левого списка, представляющем нужное хранилище, правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Clear**;
  - базы данных целиком — переключившись на нужную базу данных, щелкнув на расположенной справа кнопке **Delete database** и нажав на кнопку **OK** появившегося на экране окна-предупреждения.

Добавлять и править документы с помощью отладочных инструментов, к сожалению, нельзя.

7.6. Самостоятельное упражнение

Реализуйте в телефонной книге:

- сортировку абонентов по фамилиям;
- фильтрацию абонентов по началу их фамилий.

  На странице index.html уже присутствует веб-форма с якорем `search`, в которой находится поле ввода ключевого слова для фильтрации с якорем `txtSearch` и кнопка отправки данных **Вперед**. Пример кода, фильтрующего документы по началу значения какого-либо индексированного свойства, можно найти в *разд. 7.3.3.3*.

Урок 8
Фоновые задачи

Фоновые задачи

Синхронный вывод на экран

Все веб-сценарии, встроенные в страницу, выполняются в основном потоке.

> *Поток* — единица исполнения программного кода, своего рода виртуальный процессор, создаваемый операционной системой для выполнения программы. Пока поток занят выполнением одной задачи, другую задачу он исполнить не может.
>
> *Основной поток* — поток, в котором выполняется код обычных веб-сценариев, а также код самого веб-обозревателя.

Если какой-либо сценарий выполняется в течение продолжительного времени, веб-обозреватель «повиснет». Мы не сможем ни прокрутить страницу, ни перейти на другую щелчком на гиперссылке. Более того, у нас даже не получится свернуть или закрыть окно веб-обозревателя, поскольку код самого веб-обозревателя также выполняется в основном потоке, который сейчас занят другой задачей.

Для повышения отзывчивости страниц можно выполнять длительные задачи по частям, по сигналам периодического таймера, либо в фоновых потоках.

> *Фоновый поток* — поток, работающий параллельно с основным. Часто применяется для выполнения длительных вычислений, что позволяет повысить отзывчивость страниц и веб-приложений.

Однако очередной сигнал таймера может прийти в момент, когда основной поток занят выполнением другой задачи, и веб-обозреватель все равно «тормознет». А организация фоновых потоков и управление ими — процедура весьма трудоемкая.

8.1. Фоновые задачи

В качестве выхода из указанного затруднения можно организовать фоновую задачу, которая будет выполняться, когда веб-обозреватель находится в состоянии простоя.

> *Состояние простоя* — состояние веб-обозревателя, когда он ожидает действий пользователя.

> *Фоновая задача* — задача, выполняемая веб-обозревателем в состоянии простоя.

Фоновая задача реализуется в виде обычной функции и регистрируется в качестве, собственно, фоновой задачи. Будучи зарегистрированной, функция будет вызвана, как только веб-обозреватель войдет в состояние простоя.

ВНИМАНИЕ!

Функция, реализующая фоновую задачу, вызывается однократно. Если необходимо выполнить фоновую задачу повторно, ее следует зарегистрировать снова.

8.1.1. Регистрация и выполнение фоновых задач

Для регистрации фоновой задачи применяется метод `requestIdleCallback()` класса `Window`:

```
requestIdleCallback(<функция, реализующая фоновую задачу>[,
                    <параметры>={}])
```

Функция, реализующая фоновую задачу, должна принимать один параметр, о котором мы поговорим позже, и не должна возвращать результат.

Необязательные *параметры* задаются в виде служебного объекта со свойствами, хранящими значения одноименных им параметров. В настоящее время поддерживается только параметр `timeout`, задающий максимальное время, через которое должна быть запущена заданная *функция*, в виде целого числа в миллисекундах.

Метод возвращает целочисленный идентификатор зарегистрированной фоновой задачи. Его можно сохранить в какой-либо переменной и использовать позже, если потребуется отменить эту фоновую задачу до ее выполнения.

Пример регистрации фоновой задачи, реализованной в функции `taskFunc()` и выполняемой не позже, чем через 1 с:

```
let hTask = requestIdleCallback(taskFunc, { timeout: 1000 });
```

Можно зарегистрировать произвольное количество фоновых задач, в этом случае они будут выполняться в том порядке, в каком были зарегистрированы. Однако если при регистрации какой-либо задачи было указано максимальное время, через которое задача должна быть выполнена (параметр `timeout`), и это время истекает, возможно, задача будет выполнена вне порядка регистрации (и, возможно, даже когда веб-обозреватель занят выполнением другой задачи, необязательно фоновой).

ВНИМАНИЕ!

Фоновые задачи не должны производить какие-либо долго выполняющиеся операции, в противном случае веб-обозреватель не сможет оперативно реагировать на действия пользователя. Длительные операции лучше разбивать на отдельные, быстро выполняющиеся части и выполнять их в отдельных фоновых задачах.

Также фоновые задачи, по возможности, не должны ничего выводить на страницу (для вывода данных они должны применять синхронный вывод, описываемый далее).

8.1.2. Получение сведений о времени, отводимом на выполнение фоновой задачи

Функция, реализующая фоновую задачу, должна принимать один параметр. Этим параметром передается объект класса `IdleDeadline`, хранящий сведения о времени, которое веб-обозреватель может отвести для выполнения фоновой задачи.

Класс `IdleDeadline` поддерживает одно доступное только для чтения свойство и один метод:

- `didTimeout` — свойство, хранит значение:
  - `false` — если фоновая задача была запущена вследствие того, что веб-обозреватель вошел в состояние простоя;
  - `true` — если фоновая задача была запущена вследствие того, что истекло указанное при ее регистрации время;
- `timeRemaining()` — метод, возвращает приблизительное время, оставшееся до выхода веб-обозревателя из состояния простоя, в виде вещественного числа в миллисекундах. Если метод вернет 0, фоновая задача должна прервать выполнение.

> ***Внимание!***
>
> Нужно иметь в виду: если фоновая задача была запущена вследствие того, что истекло указанное при ее регистрации время, метод `timeRemaining()` вернет результат, близкий к 0.

Пример:

```
function taskFunc(oDL) {
    // Если веб-обозреватель может отвести для выполнения этой фоновой
    // задачи не менее 10 мс, или если задача запустилась на выполнение
    // вследствие истечения указанного максимального времени...
    if (oDL.timeRemaining() >= 10 || oDL.didTimeout) {
        // ...выполняем необходимые действия
    } else {
        // В противном случае - не выполняем и регистрируем задачу
        // повторно, чтобы она выполнилась позже, когда у
        // веб-обозревателя будет больше свободного времени
        hTask = requestIdleCallback(taskFunc, { timeout: 1000 });
    }
}
```

8.1.3. Отмена фоновых задач

Если требуется отменить зарегистрированную фоновую задачу до ее выполнения, следует воспользоваться методом `cancelIdleCallback()` класса `Window`:

`cancelIdleCallback(`*<идентификатор фоновой задачи>*`)`

Идентификатор фоновой задачи — тот самый, который возвращается методом `requestIdleCallback()` (см. *разд. 8.1.1*).

Пример:

```
cancelIdleCallback(hTask);
```

8.2. Синхронный вывод на экран

Любой вывод информации на экран — отображение программно созданных элементов страницы, подсветка гиперссылки при наведении курсора мыши, прокрутка содержимого страницы и т. п. — веб-обозреватель выполняет в строго определенные моменты времени, когда производится обновление изображения на мониторе.

Промежуток времени между обновлениями изображения на мониторе задается его частотой синхронизации. Так, если частота синхронизации монитора составляет 60 Гц, изображение на нем обновляется каждую $^1/_{60}$ секунды.

Крайне желательно, чтобы фоновая задача реализовывала синхронный вывод — это повысит производительность.

> *Синхронный вывод* — вывод информации на страницу строго во время обновления изображения на мониторе.
>
> *Задача синхронного вывода* — задача, реализующая синхронный вывод на экран. Выполняется в моменты обновления изображения на мониторе.

Задача синхронного вывода подобна фоновой задаче. Она также реализуется в виде функции и также регистрируется, для чего применяется метод `requestAnimationFrame()` класса `Window`:

```
requestAnimationFrame(<функция, реализующая задачу синхронного вывода>)
```

Указываемая *функция* должна принимать единственный параметр — текущее время в виде вещественного числа в миллисекундах.

Метод возвращает целочисленный идентификатор зарегистрированной задачи синхронного вывода. Его можно сохранить в какой-либо переменной и использовать позже, если потребуется отменить эту задачу до ее выполнения.

Пример:

```
let hSO;
function taskFunc(oDL) {
    . . .
    // Вычисляем какие-либо данные.
    // Для их вывода регистрируем задачу синхронного вывода,
    // реализованную функцией soFunc().
    hSO = requestAnimationFrame(soFunc);
}
```

Для отмены зарегистрированной задачи синхронного вывода до ее выполнения служит метод `cancelAnimationFrame()` класса `Window`:

```
cancelAnimationFrame(<идентификатор задачи синхронного вывода>)
```

Идентификатор задачи синхронного вывода — тот самый, который возвращается методом `requestAnimationFrame()`.

8.3. Упражнение. Вычисление чисел Фибоначчи, вариант 2

Напишем еще одно веб-приложение, вычисляющее заданное количество чисел Фибоначчи. Числа будут вычисляться посредством фоновой задачи, выполняющейся каждые 0,1 с, а выводиться на экран — в задаче синхронного вывода.

1. Найдем в папке 8\!sources сопровождающего книгу электронного архива (см. *приложение*) файл 8.3.html (страница, на основе которой будет создано веб-приложение) и скопируем его куда-либо на локальный диск.

   Страница включает абзац с якорем `output`, в котором будет выведен результат. В конце ее HTML-кода находится пустой тег `<script>`, в котором мы запишем программный код.

   Каждое вычисленное число Фибоначчи будем заключать в тег `<span>` и помещать его в абзац `output`. Страница включает внутреннюю таблицу стилей, которая задает оформление для этих тегов.

   Сначала объявим константу `fibCount`, в которой будет храниться количество вычисляемых чисел Фибоначчи.

2. Откроем в текстовом редакторе копию файла 8.3.html и запишем в «пустой» тег `<script>` код, объявляющий константу `fibCount` со значением 20:

```
<script type="text/javascript">
    const fibCount = 20;
</script>
```

   Двадцати чисел хватит для того, чтобы проверить приложение в работе.

   Объявим константу для хранения конфигурационного объекта, используемого при регистрации фоновой задачи. И заодно получим доступ к абзацу `output`.

3. Добавим код, объявляющий необходимые константы:

```
const fibCount = 20;
const oOutput = document.getElementById('output');
const oParams = { timeout: 100 };
```

   Для вычисления чисел нам понадобятся следующие переменные:

   - `i` — счетчик уже вычисленных чисел (изначально — 0);
   - `pprev` — предпредыдущее число;
   - `prev` — предыдущее число;
   - `aFibs` — массив для хранения готовых тегов `<span>`, содержащих уже вычисленные числа (изначально «пустой»).

     «Складывая» готовые теги `<span>` в массив, мы обезопасим себя на тот случай, если фоновая задача, вычисляющая числа, будет выполняться чаще, чем задача синхронного вывода, выводящая эти теги на экран;

- `bIsRegged` — признак, была ли уже зарегистрирована задача синхронного вывода, выводящая на экран теги <span> с вычисленными числами, или еще нет. Будет указываться в виде логической величины, соответственно `true` или `false`. Изначально — `false`.

  Если фоновая задача будет выполняться чаще задачи синхронного вывода, может случиться так, что последняя окажется зарегистрирована несколько раз подряд. Это совершенно ни к чему и, вдобавок, может снизить производительность. Введенный нами признак позволит исключить многократную регистрацию задачи синхронного вывода.

4. Добавим код, объявляющий описанные ранее переменные:

```
let i = 0, pprev, prev, aFibs = [], bIsRegged = false;
```

5. Добавим объявление функции `taskSOShowFibs()`, реализующей задачу синхронного вывода, которая будет выводить вычисленные числа на экран:

```
function taskSOShowFibs(time) {
    if (aFibs.length > 0) {
        const oSpan = aFibs.shift();
        oOutput.appendChild(oSpan);
        requestAnimationFrame(taskSOShowFibs);
    } else
        bIsRegged = false;
}
```

Если в массиве `aFibs` есть готовые теги <span> с вычисленными числами, функция вынимает из массива первый тег, добавляет его в абзац `output` и сразу же заново регистрирует себя в качестве задачи синхронного вывода (чтобы позже извлечь и вывести остальные теги). Если же тегов в массиве более нет, она заносит в переменную `bIsRegged` значение `false`, сообщая тем самым, что задача синхронного вывода не зарегистрирована (тогда для вывода очередного числа фоновой задаче придется заново зарегистрировать ее).

6. Добавим объявление функции `taskCalculate()`, реализующей фоновую задачу, которая вычисляет числа:

```
function taskCalculate(oDL) {
    if (i < fibCount) {
        if (oDL.timeRemaining() > 10 || oDL.didTimeout) {
            let current;
            if (i <= 1)
                current = i;
            else
                current = pprev + prev;
            [pprev, prev] = [prev, current];
            const oSpan = document.createElement('span');
            oSpan.textContent = current;
            aFibs.push(oSpan);
            i++;
```

```
            if (!bIsRegged) {
                requestAnimationFrame(taskSOShowFibs);
                bIsRegged = true;
            }
        }
        requestIdleCallback(taskCalculate, oParams);
    }
}
```

Если еще не все требуемые числа вычислены, проверяем, есть ли у веб-обозревателя свободное время, или задача запустилась вследствие истечения указанных при ее регистрации 0,1 с. Если это так, вычисляем очередное число, как делали это при выполнении *упражнения 3.3*. Далее создаем новый тег `<span>`, помещаем в него вычисленное число, добавляем тег в массив `aFibs` (откуда их впоследствии извлечет задача синхронного вывода) и инкрементируем счетчик вычисленных чисел `i`. Если задача синхронного вывода еще не зарегистрирована (что может случиться в начале выполнения приложения или если все числа уже выведены на экран), регистрируем ее, не забыв занести в переменную `bIsRegged` значение `true` (чтобы случайно не зарегистрировать эту же задачу еще раз, что, как мы уже знаем, нежелательно).

Неважно, есть у веб-обозревателя свободное время или нет, регистрируем ту же фоновую задачу еще раз, чтобы вычислить следующее число.

Если же все числа уже вычислены, ничего не делаем, и задача завершает работу.

7. Добавим код, который регистрирует фоновую задачу, реализованную функцией `taskCalculate()`, и тем самым запустит вычисления:

   ```
   requestIdleCallback(taskCalculate, oParams);
   ```

   Откроем страницу 8.3.html в веб-обозревателе и подождем, пока не будут выведены все 20 чисел Фибоначчи (рис. 8.1).

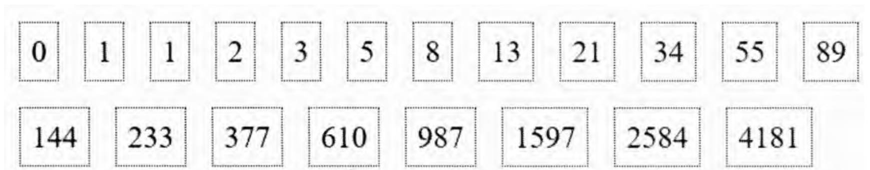


Рис. 8.1. Веб-приложение, вычисляющее числа Фибоначчи

Для эксперимента вы можете указать другое количество вычисляемых чисел в константе `fibCount` и обновить открытую в веб-обозревателе страницу, чтобы увидеть результат.

8.4. Самостоятельное упражнение

Напишите аналогичное приложение, вычисляющее квадратные корни от последовательности целых чисел из указанного диапазона (аналог написанного при выполнении *упражнения 3.5*).

ЧАСТЬ III
HTML-компоненты и шаблоны

- HTML-компоненты
- Автономные HTML-компоненты и расширенные теги
- Скрытая DOM
- Шаблоны
- Слоты

Урок 9
HTML-компоненты

HTML-компоненты

Автономные HTML-компоненты и расширенные теги

Теги компонентов

Скрытая DOM

Результатом выполнения *упражнения 1.2* стал веб-компонент — сложный элемент веб-страницы, реализованный в виде класса и пригодный для повторного применения. Веб-компоненты удобны в использовании и позволяют значительно сократить трудоемкость разработки.

Однако у веб-компонентов имеется существенный недостаток — для их размещения на странице придется прибегать к программированию. Для размещения одного компонента потребуется написать, по крайней мере, одно выражение JavaScript — создающее объект компонента на основе заданного базового элемента страницы.

Этого недостатка лишены HTML-компоненты.

> *HTML-компонент* — веб-компонент, размещаемый на странице средствами HTML, без применения JavaScript.

Для размещения HTML-компонента на странице достаточно вставить в нужное место HTML-кода тег компонента. Никакого программирования для этого не требуется.

> *Тег компонента* — HTML-тег, сопоставленный с HTML-компонентом. Применяется для размещения компонента на странице.

9.1. Создание HTML-компонентов

Можно создавать две разновидности HTML-компонентов:

- автономные компоненты.

  > *Автономный HTML-компонент* — не наследуется от какого-либо стандартного HTML-тега.

  Позволяют реализовать в компоненте произвольные внутреннюю структуру и функциональность, вследствие чего применяются в большинстве случаев;

- расширенные теги.

> *Расширенный тег* — HTML-компонент, расширяющий функциональность какого-либо стандартного HTML-тега.

  Применяются, когда нужно нарастить функциональность какого-либо стандартного тега.

Для создания нового компонента необходимо:

- объявить класс, реализующий функциональность компонента;
- зарегистрировать компонент в подсистеме HTML-компонентов веб-обозревателя.

9.1.1. Объявление класса HTML-компонента

Классы компонентов удобнее объявлять, применяя новый синтаксис (см. *разд. 1.1*).

В зависимости от того, к какой разновидности относится создаваемый компонент, его класс следует делать производным:

- автономный компонент — от класса `HTMLElement`.

  Класс `HTMLElement` является базовым для всех классов, реализующих элементы страниц разных типов: класса абзаца `HTMLParagraphElement`, класса блока `HTMLDivElement`, класса изображения `HTMLImageElement` и др.

  Пример:

  ```
  class BigText extends HTMLElement { . . . }
  ```

- расширяемый тег — от класса, представляющего тег, функциональность которого расширяет создаваемый компонент (например, если планируется расширить тег абзаца <p>, следует объявить подкласс класса `HTMLParagraphElement`). Пример:

  ```
  class GreenText extends HTMLParagraphElement { . . . }
  ```

Вся внутренняя структура компонента — входящие в его состав элементы страницы (абзацы, блоки, заголовки, списки, веб-формы и пр.) — создается программно в конструкторе класса, после вызова унаследованного конструктора.

9.1.1.1. Создание скрытой DOM

Однако перед конструированием внутренней структуры компонента обязательно следует создать в нем скрытую DOM.

> *DOM* (Document Object Model, объектная модель документа) — структура взаимосвязанных объектов, представляющих различные элементы страницы.
>
> *Скрытая DOM* — DOM, представляющая внутреннюю структуру HTML-компонента и изолированная от DOM остальной веб-страницы.

Заключение всех элементов внутренней структуры компонента в скрытую DOM позволяет:

♦ защитить «внутренности» компонента от несанкционированного доступа из сторонних веб-сценариев (что может нарушить работу компонента);

♦ привязать к компоненту отдельную таблицу стилей, действующую только «внутри» компонента.

Для создания скрытой DOM применяется метод `attachShadow(<параметры>)`, который следует вызывать у самого компонента.

Параметры указываются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Доступны для указания два параметра:

♦ `mode` — обязательный, указывает, будет ли создаваемая скрытая DOM доступна для внешних веб-сценариев. Значение должно представлять собой одну из следующих строк:

  • `'closed'` — скрытая DOM не будет доступна «извне»;

  • `'open'` — скрытая DOM будет доступна «извне».

    В этом случае ссылку на объект, представляющий скрытую DOM, можно получить, обратившись к свойству `shadowRoot` самого компонента.

    У обычных тегов и компонентов с недоступной «извне» скрытой DOM (параметру `mode` было дано значение `'closed'`) это свойство будет хранить `null`;

♦ `delegatesFocus` — необязательный, указывает поведение компонента, когда на каком-либо элементе его внутренней структуры, не способном принимать фокус ввода, щелкают мышью, в виде логической величины:

  • `false` — ничего не происходит;

  • `true` — фокус ввода получит ближайший элемент внутренней структуры, способный принять фокус ввода.

  Значение по умолчанию — `false`.

> ***Внимание!***
> В настоящее время параметр `delegatesFocus` игнорируется Mozilla Firefox.

Метод `attachShadow()` возвращает объект класса `ShadowRoot`, представляющий созданную скрытую DOM.

9.1.1.2. Создание внутренней структуры HTML-компонента

Создать элементы внутренней структуры компонента можно двумя способами:

♦ сконструировав элементы вызовами метода `createElement()` класса `HTMLDocument` и добавив их в скрытую DOM с помощью метода `appendChild()` или аналогичных ему (поддержку этих методов класс `ShadowRoot` получает от суперкласса `HTMLElement`);

♦ сформировав строку с HTML-кодом, создающим элементы внутренней структуры, и присвоив ее свойству `innerHTML` класса `ShadowRoot`.

Листинг 9.1 содержит полный код автономного HTML-компонента `BigText`, выводящего абзац текста увеличенным кеглем (соответствующее оформление задается во внешней таблице стилей big-text.css, привязываемой к компоненту).

Листинг 9.1. Код компонента BigText

```
class BigText extends HTMLElement {
    constructor() {
        // Не забываем вызвать унаследованный конструктор
        super();
        // Создаем скрытую DOM
        const oShadow = this.attachShadow({ mode: 'closed' });
        // Создаем внутреннюю структуру компонента.
        // Сначала - тег <link>, привязывающий внешнюю таблицу стилей
        const oLink = document.createElement('link');
        oLink.type = 'text/css';
        oLink.rel = 'stylesheet';
        oLink.href = 'big-text.css';
        // Добавляем вновь созданный тег в ранее созданную скрытую DOM
        oShadow.appendChild(oLink);
        // Потом создаем абзац с выводимым текстом
        const oP = document.createElement('p');
        oP.textContent = 'Большой текст';
        oShadow.appendChild(oP);
    }
}
```

Не забываем, что стили, записанные в привязанной к компоненту таблице стилей, действуют только «внутри» компонента, не затрагивая остальную страницу.

9.1.1.3. Передача параметров HTML-компонентам

От компонента `BigText` будет больше пользы, если выводимую им строку можно будет произвольно задавать, указав ее в качестве параметра. Проще всего передавать параметры компонентам посредством атрибутов, записываемых в теге компонента.

Для получения значений этих атрибутов можно использовать метод `getAttribute()`, вызываемый непосредственно у компонента:

```
getAttribute(<имя атрибута тега>)
```

Он возвращает:

- значение атрибута тега с заданным *именем* в виде строки;
- «пустую» строку — если заданный атрибут тега не имеет значения (это справедливо в случае атрибутов тега без значений — например, `disabled`);
- `null` — если заданный атрибут вообще не указан в теге.

Пример получения в компоненте `BigText2` строки, заданной в атрибуте тега `text`, и вывода ее на экран:

```
class BigText2 extends HTMLElement {
    constructor() {
        super();
        . . .
        const oP = document.createElement('p');
        oP.textContent = this.getAttribute('text');
        oShadow.appendChild(oP);
    }
}
```

Нужно помнить, что метод `getAttribute()` сможет извлечь значения атрибутов тега только после того, как в памяти, в процессе формирования DOM страницы, будет создан объект, представляющий этот тег. Поэтому перед регистрацией компонента (о ней будет рассказано позже) следует дождаться, пока вся DOM страницы не будет сформирована. Добиться этого можно, записав код, выполняющий регистрацию компонента, после HTML-кода самой страницы. Пример:

```
<html>
    . . .
</html>
<script src="big-text.js" type="text/javascript"></script>
```

> ***Внимание!***
>
> Еще раз: чтобы компонент смог получить доступ к атрибутам или содержимому своего тега, код его регистрации следует поместить после HTML-кода страницы.

9.1.1.4. Получение содержимого тега HTML-компонента

Если компоненту требуется передать какое-либо большое строковое значение, удобнее указать его в качестве содержимого тега компонента. Получить это содержимое со всеми присутствующими в нем HTML-тегами можно, обратившись к свойству `innerHTML` компонента, а извлечь содержимое без тегов — обратившись к свойству `textContent`.

После извлечения содержимого тега компонента рекомендуется удалить его, поскольку оно более не нужно, и представляющие его объекты отныне лишь будут занимать оперативную память. Сделать это можно присваиванием свойству `innerHTML` или `textContent` компонента «пустой» строки.

Пример получения в компоненте `BigText3` содержимого тега этого компонента:

```
class BigText3 extends HTMLElement {
    constructor() {
        super();
        . . .
        const oP = document.createElement('p');
        oP.innerHTML = this.innerHTML;
```

```
        this.innerHTML = '';
        oShadow.appendChild(oP);
    }
}
```

9.1.2. Регистрация HTML-компонентов

После объявления класса HTML-компонента его следует зарегистрировать, чтобы веб-обозреватель «узнал» о его существовании.

Для управления HTML-компонентами, в том числе и для их регистрации, применяется объект класса `CustomElementRegistry`, представляющий подсистему HTML-компонентов веб-обозревателя. Этот объект хранится в свойстве `customElements` класса `Window`.

Для регистрации компонента применяется метод `define()` класса `CustomElementRegistry`:

```
define(<регистрационное имя компонента>, <класс компонента>[,
                                          <параметры>=undefined])
```

Регистрационное имя компонента задается в виде строки и обязательно должно содержать хотя бы один дефис.

> **Внимание!**
>
> Регистрационные имена компонентов должны быть уникальными в пределах страницы, на которой будут располагаться эти компоненты.

Если ранее уже был зарегистрирован компонент с таким же *регистрационным именем* и (или) *классом*, будет возбуждено исключение `NotSupportedError`.

Необязательные *параметры* задаются в виде служебного объекта. При регистрации:

- автономного компонента — *параметры* ***не указываются*** (также можно указать в качестве *параметров* «пустой» служебный объект).

  Пример регистрации автономного компонента `BigText` под именем `big-text`:

  ```
  customElements.define('big-text', BigText);
  ```

- расширенного тега — в качестве *параметров* указывается служебный объект со свойством `extends`. Значением этого свойства должна быть строка с именем расширяемого HTML-тега, записанным в нижнем регистре *без угловых скобок*.

  Пример регистрации расширенного тега `GreenText`, расширяющего тег абзаца <p>, под именем `green-text`:

  ```
  customElements.define('green-text', GreenText, { extends: 'p' });
  ```

Как только компонент будет зарегистрирован, веб-обозреватель начинает просматривать DOM страницы в поисках тегов этого компонента. Найдя очередной тег, он создает экземпляр компонента и выводит его на страницу.

9.2. Размещение HTML-компонентов на веб-странице

Компонент размещается на странице путем вставки тега этого компонента в нужное место HTML-кода. Этот тег у разных разновидностей компонентов формируется по-разному:

- у автономного компонента — имеет имя, совпадающее с регистрационным именем компонента, и всегда является парным. Примеры:

```
// Размещение компонента BigText
<big-text></big-text>
// То же самое, с указанием выводимой в компоненте строки
// в атрибуте тега text
<big-text2 text="Огромный текст"></big-text2>
// То же самое, с указанием выводимой строки непосредственно
// в теге компонента
<big-text3>Огромный текст</big-text3>
```

- у расширенного тега — представляет собой расширяемый тег с записанным в нем атрибутом `is`, в качестве значения которого указывается регистрационное имя компонента. Примеры:

```
// Размещение компонента GreenText, расширяющего тег <p>
<p is="green-text"></p>
// То же самое, с указанием выводимой строки разными способами
<p is="green-text2" text="Зеленый — значит, безопасно"></p>
<p is="green-text3">Зеленый — значит, безопасно</p>
```

HTML-компоненты можно создавать и программно:

- автономные компоненты — вызовом метода `createElement()` класса `HTMLDocument` с передачей этому методу регистрационного имени компонента в качестве параметра. Пример:

```
const oBT = document.createElement('big-text');
document.body.appendChild(oBT);
```

- расширенные теги — с применением расширенного синтаксиса метода `createElement()`:

```
createElement(<имя расширяемого тега>, <параметры>)
```

Параметры указываются в виде служебного объекта со свойством `is`, содержащего строку с регистрационным именем компонента. Пример:

```
const oGT = document.createElement('p', { is: 'green-text' });
document.body.appendChild(oGT);
```

Для программного указания значений атрибутов у тега компонента применяется метод `setAttribute()`, вызываемый у компонента:

```
setAttribute(<имя атрибута тега>, <значение атрибута тега>)
```

`Имя атрибута тега` следует указывать в виде строки, `значение` может быть любого типа. Пример:

```
const oBT2 = document.createElement('big-text2');
oBT2.settAttribute('text', 'Огромный текст');
document.body.appendChild(oBT2);
```

9.3. Упражнение. Пишем HTML-компонент — спойлер

Напишем HTML-компонент, реализующий спойлер. Заголовок спойлера будем передавать через атрибут `header` тега компонента, а его содержимое — непосредственно в теге компонента. В остальном он будет аналогичен веб-компоненту, написанному при выполнении *упражнения 1.2*.

Сделаем компонент автономным, дадим его классу имя `HTMLSpoiler` и зарегистрируем под именем `html-spoiler`. Для оформления используем готовую внешнюю таблицу стилей htmlspoiler.css. Детали реализации обговорим в процессе программирования.

1. Найдем в папке 9\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы 9.3.html (веб-страница с двумя заготовками для спойлеров), 9.3.css (таблица стилей с оформлением страницы) и htmlspoiler.css (таблица стилей с оформлением спойлера). Скопируем их куда-либо на локальный диск.
2. Создадим в той же папке, где хранятся копии указанных файлов, файл htmlspoiler.js, в котором будет записан код компонента, и откроем его в текстовом редакторе.
3. Добавим в файл htmlspoiler.js объявление функции `HTMLSpoiler_headerClick()` — обработчика события `click` гиперссылки, входящей в состав спойлера. Эта функция будет полностью аналогична функции `Spoiler_headerClick()` из кода ранее написанного спойлера (см. *упражнение 1.2*).

   По аналогии с ранее написанным спойлером новый компонент будет содержать:

   - блок-«обертку», заключающий всю разметку спойлера, которая состоит из:
     - гиперссылки-заголовка, по которой посетитель будет щелкать мышью, чтобы развернуть или свернуть спойлер;
     - блока с содержимым, показывающимся или скрывающимся при развертывании или свертывании спойлера.

   Поскольку мы пишем автономный компонент, его класс `HTMLSpoiler` сделаем производным от класса `HTMLElement`. И сразу же объявим в нем свойство `oDivWrapper` — для хранения блока-«обертки», к которому придется обращаться в методах объявляемого класса.

4. Добавим объявление класса компонента `HTMLSpoiler` со свойством `oDivWrapper`:

   ```
   class HTMLSpoiler extends HTMLElement {
       oDivWrapper
   }
   ```

В конструкторе класса создадим скрытую DOM и всю внутреннюю структуру компонента.

5. Добавим в класс `HTMLSpoiler` объявление конструктора:

```
class HTMLSpoiler extends HTMLElement {
    . . .
    constructor() {
        super();
        const oShadow = this.attachShadow({ mode: 'closed' });
        const oLink = document.createElement('link');
        oLink.type = 'text/css';
        oLink.rel = 'stylesheet';
        oLink.href = 'htmlspoiler.css';
        oShadow.appendChild(oLink);
        this.oDivWrapper = document.createElement('div');
        this.oDivWrapper.classList.add('spoiler');
        const oA = document.createElement('a');
        oA.classList.add('header');
        oA.href = '#';
        oA.compSpoiler = this;
        oA.textContent = this.getAttribute('header');
        oA.addEventListener('click', HTMLSpoiler__headerClick);
        this.oDivWrapper.appendChild(oA);
        const oDivContent = document.createElement('div');
        oDivContent.classList.add('content');
        oDivContent.innerHTML = this.innerHTML;
        this.innerHTML = '';
        this.oDivWrapper.appendChild(oDivContent);
        oShadow.appendChild(this.oDivWrapper);
    }
}
```

Здесь создаем скрытую DOM — и в ней;

- тег `<link>`, привязывающий внешнюю таблицу стилей htmlspoiler.css;
- блок-«обертку» со стилевым классом `spoiler` — и в нем:
  - гиперссылку-заголовок со стилевым классом `header` и содержимым, взятым из атрибута `text` тега компонента. Не забываем создать в гиперссылке свойство `compSpoiler`, хранящее ссылку на текущий объект компонента, и привязать к ее событию `click` обработчик `HTMLSpoiler_headerClick()`;
  - блок для содержимого спойлера со стилевым классом `content` и содержимым, взятым из тега компонента.

6. Добавим в класс компонента объявления методов `toggle()` и геттера динамического свойства `shown` (их код будет слегка отличаться от написанного при выполнении *упражнения 1.2*):

```
class HTMLSpoiler extends HTMLElement {
    . . .
    toggle() {
        this.oDivWrapper.classList.toggle('shown');
    }

    get shown() {
        return this.oDivWrapper.classList.contains('shown');
    }
}
```

7. Добавим в класс компонента объявление сеттера динамического свойства shown, который можно без изменений взять из класса Spoiler (см. *упражнение 1.2*).
8. Добавим после объявления класса компонента выражение, регистрирующее компонент:

```
class HTMLSpoiler extends HTMLElement {
    . . .
}

customElements.define('html-spoiler', HTMLSpoiler);
```

9. Откроем в текстовом редакторе копию страницы 9.3.html и добавим в конце ее кода привязку файла сценария htmlspoiler.js:

```
<html>
    . . .
</html>
<script src="htmlspoiler.js" type="text/javascript"></script>
```

10. Запишем в тег <article>, присутствующий в коде этой страницы, теги двух компонентов HTMLSpoiler:

```
<article class="cont">
    <html-spoiler header="JavaScript">
        <!-- Содержимое первого спойлера -->
    </html-spoiler>
    <html-spoiler header="TypeScript">
        <!-- Содержимое второго спойлера -->
    </html-spoiler>
</article>
```

Каждый созданный здесь спойлер будет содержать два абзаца с кратким описанием соответствующего языка программирования. Эти абзацы можно взять из кода страницы 1\!sources\index.html.

Откроем страницу 9.3.html в веб-обозревателе. Попробуем развернуть и свернуть оба спойлера.

9.4. Дополнительные инструменты для работы с HTML-компонентами

9.4.1. Волшебные методы HTML-компонентов

> *Волшебный метод* — метод класса, вызываемый самим веб-обозревателем при возникновении определенных условий (например, когда значение атрибута тега компонента изменяется программно).

Волшебные методы позволяют добавить в класс своего рода «волшебство».

Вот волшебные методы, доступные для объявления в классе компонента:

- `attributeChangedCallback()` — вызывается сразу после того, как значение какого-либо атрибута тега компонента будет изменено программно. Формат объявления:

  ```
  attributeChangedCallback(<имя атрибута тега>, <старое значение>,
                                                <новое значение>)
  ```

  Объявив в классе этот метод, следует указать веб-обозревателю, при изменении значений каких атрибутов тега компонента следует его вызывать. Это производится объявлением статического, доступного только для чтения (содержащего лишь геттер) свойства `observedAttibutes`. Это свойство должно выдавать массив имен нужных атрибутов тегов, заданных в виде строк.

  Пример реагирования на программное изменение атрибута `text` тега компонента `BigText2`:

  ```
  class BigText2 extends HTMLElement {
      oP
      constructor() {
          super();
          . . .
          this.oP = document.createElement('p');
          this.oP.textContent = this.getAttribute('text');
          oShadow.appendChild(this.oP);
      }

      attributeChangedCallback(name, oldValue, newValue) {
          if (name == 'text')
              this.oP.textContent = newValue;
      }

      static get observedAttributes() {
          return ['text'];
      }
  }
  ```

- `connectedCallback()` — вызывается при помещении компонента на страницу.

  В документации по HTML-компонентам сказано, что этот метод также может быть вызван после удаления компонента со страницы (например, в результате удаления его тега). Проверить, помещен ли компонент на страницу, можно обращением к доступному только для чтения свойству `isConnected` компонента. Если компонент помещен на страницу, свойство будет хранить `true`, в противном случае — `false`;

- `disconnectedCallback()` — вызывается при удалении компонента со страницы (например, вследствие программного удаления его тега).

  Пример:

```
class Bigtext extends HTMLElement {
    // Объявляем статическое свойство, хранящее количество созданных
    // компонентов
    static count = 0;
    . . .
    connectedCallback() {
        // При создании нового компонента увеличиваем это количество
        if (this.isConnected)
            this.count++;
    }
    disconnectedCallback() {
        // А при удалении компонента — уменьшаем
        this.count--;
    }
}
```

9.4.2. Средства CSS для работы с HTML-компонентами

В языке CSS для работы с HTML-компонентами появилась поддержка нескольких новых селекторов: четырех псевдоклассов и одного псевдоэлемента.

Новые псевдоклассы:

- `:defined` — указывает на любой тег, поддерживаемый веб-обозревателем, — стандартный HTML-тег или тег *уже зарегистрированного* компонента. На теги еще не зарегистрированных компонентов не указывает. Используется во «внешних» по отношению к компоненту таблицах стилей. Пример:

```
/* Компонент BigText3 зарегистрирован, и его содержимое будет выделено
   полужирным шрифтом */
big-text3:defined        {font-weight: bold;}
/* Компонент SmallText не зарегистрирован, и его содержимое будет
   зачеркнуто */
small-text:not(:defined) {text-decoration: line-through;}
. . .
<big-text3>Огромный текст</big-text3>
<small-text>Маленький текст</small-text>
```

- `:host` — указывает на тег компонента и используется во «внутренних» таблицах стилей, привязанных к компонентам:

```
/* Таблица стилей big-text3.css, «внутренняя» по отношению к
   компоненту BigText3 */
/* Выводим все текстовое содержимое компонента курсивом */
:host {font-style: italic;}
. . .
<big-text3>Это <strong>большой</scrong> текст</big-text3>
```

- `:host(<селектор>)` — аналогичен `:host`, но указывает на тег компонента, который соответствует заданному *селектору*:

```
/* Таблица стилей big-text3.css, «внутренняя» по отношению
   к компоненту BigText3 */
:host(.upper) {text-transform: uppercase;}
. . .
<big-text3>Это большой текст</big-text3>
<big-text3 class="upper">ЭТО БОЛЬШОЙ ТЕКСТ</big-text3>
```

- `:host-context(<селектор>)` — аналогичен `:host`, но указывает на тег компонента, родитель которого соответствует заданному селектору.

  Внимание!

  Псевдокласс `:host-content()` в настоящее время не поддерживается Mozilla Firefox.

  Пример:

```
/* Таблица стилей big-text3.css, «внутренняя» по отношению к
   компоненту BigText3 */
:host-content(section) {text-decoration: underline;}
. . .
<big-text3>Это большой текст</big-text3>
<div><big-text3>Это большой текст</big-text3></div>
<section><big-text3>Это большой текст</big-text3></section>
```

Новый псевдоэлемент `::part(<часть>)` указывает на тег, составляющий внутреннюю структуру компонента, у которого в атрибуте `part` записана заданная *часть*. Применяется в таблицах стилей, «внешних» по отношению к компоненту. Пример:

```
// Код компонента BigText3
const oP = document.createElement('p');
oP.innerHTML = '<span part="main">&laquo;' + this.innerHTML +
               '&raquo;</span> крупным шрифтом';
oShadow.appendChild(oP);
. . .
big-text3::part(main) {font-weight: bold;}
. . .
<big-text3>Внимание!</big-text3>
<!-- Результат: «Внимание!» крупным шрифтом -->
```

9.4.3. Проверка регистрации HTML-компонентов

Класс `CustomElementRegistry`, представляющий подсистему HTML-компонентов, поддерживает два метода, позволяющих выяснить, зарегистрирован ли заданный компонент или нет:

- `get(<`*`регистрационное имя`*`>)` — возвращает ссылку на функцию-конструктор класса компонента с заданным *регистрационным именем*. Если компонент с заданным *именем* еще не зарегистрирован, возвращается `undefined`. Пример:

```
if (customElements.get('green-text')) {
    // Компонент с именем green-text зарегистрирован
}
```

- `whenDefined(<`*`регистрационное имя`*`>)` — возвращает промис, подтверждаемый как только компонент с заданным *регистрационным именем* будет зарегистрирован. В качестве нагрузки возвращенный промис получит ссылку на функцию-конструктор класса компонента.

  Пример удаления абзаца `placeholder` с текстом-подстановкой после загрузки компонента `BigText3`:

```
<p id="placeholder">Здесь скоро будет выведен компонент BigText3...</p>
<big-text3>Огромный текст</big-text3>
. . .
(async function () {
    const oPP = document.getElementById('placeholder');
    await customElements.whenDefined('big-text3');
    oPP.parentElement.removeChild(oPP);
})();
```

9.5. Самостоятельные упражнения

- Добавьте в компонент `HTMLSpoiler` (результат выполнения *упражнения 9.3*) возможность отслеживать программные изменения значения атрибута `header` тега компонента.
- Создайте расширенный тег `Time`, расширяющий HTML-тег `<div>`, который будет выводить текущее время. Зарегистрируйте его под именем `html-time`. Сделайте так, чтобы он выводил время:
  - по умолчанию — без секунд;
  - при включении в тег компонента атрибута без значения `seconds` или обычного атрибута тега `seconds` со значением `yes` — с секундами.

  Выведите время моноширинным шрифтом с кеглем 18 пунктов. Соответствующий стиль запишите в таблицу стилей time.css, которую привяжите к компоненту.

  Сохраните код компонента в файле time.js.

Урок 10
Шаблоны и слоты

Шаблоны

Слоты

Именованные и неименованные слоты

HTML-компоненты (см. *урок 9*) — очень удобный инструмент программирования. Однако у них есть недостаток: элементы, составляющие внутреннюю структуру компонентов, создаются программно. Отсюда вытекают две проблемы:

- взглянув на программный код компонента, сразу трудно понять, что же он выводит на экран;
- при необходимости изменить структуру компонента придется переписывать его программный код, для чего потребуются навыки веб-программирования.

10.1. Шаблоны

Описанные ранее проблемы можно решить, описав внутреннюю структуру HTML-компонента в шаблоне.

> *Шаблон* — образец, на основе которого создается внутренняя структура компонента. Пишется на языке HTML.

Шаблон может быть написан разработчиком, не знакомым с JavaScript, например веб-верстальщиком. К тому же HTML-код, создающий какие-либо элементы страницы, намного нагляднее выполняющего ту же задачу JavaScript-кода.

10.1.1. Написание шаблонов

HTML-код шаблона записывается в парном теге `<template>`. У тега `<template>` рекомендуется указать какой-либо якорь (в атрибуте `id`), чтобы впоследствии сослаться на шаблон в программном коде.

У элементов, входящих в состав содержимого шаблона, можно указывать якоря и стилевые классы, чтобы впоследствии обратиться к ним в программном коде компонента, задать их параметры и занести в них содержимое.

Листинг 10.1 показывает HTML-код шаблона компонента `BigText4`, выводящего заданный текст увеличенным шрифтом.

Листинг 10.1. Код шаблона компонента `BigText4`

```
<template id="BigText4Template">
    <link href="big-text.css" type="text/css" rel="stylesheet">
    <p id="output"></p>
</template>
```

У самого тега `<template>`, создающего шаблон, указан якорь `BigText4Template` — по нему в программном коде компонента можно будет получить доступ к шаблону. У абзаца, в котором будет выводиться заданный текст, указан якорь `output` — для этой же цели. Кроме того, к шаблону привязана внешняя таблица стилей big-text.css, задающая оформление компонента.

Код шаблона можно поместить в любое место HTML-кода страницы. Автор книги предпочитает вставлять шаблоны в самое начало секции тела страницы (сразу после открывающего тега `<body>`).

10.1.2. Использование шаблонов

Чтобы вывести на экран компонент с применением шаблона, необходимо:

1. Получить доступ к тегу `<template>`, создающему шаблон (что можно сделать, например, по заданному у этого тега якорю).

   Тег `<template>` в DOM представляется объектом класса `HTMLTemplateElement`.

2. Извлечь содержимое полученного шаблона, обратившись к свойству `content` класса `HTMLTemplateElement`.

   Содержимое шаблона представляется объектом класса `DocumentFragment`.

3. Создать копию содержимого шаблона, полученного на *шаге 2*, вызвав у него метод `cloneNode()`:

   ```
   cloneNode([<создавать полную копию?>=false])
   ```

   Если вызвать этот метод без параметров, он вернет «пустое» содержимое шаблона, совершенно бесполезное. Чтобы получить содержимое со всеми созданными в шаблоне тегами, следует дать параметру *`создавать полную копию`* значение `true`.

4. При необходимости — получить доступ к отдельным элементам содержимого шаблона, задать их параметры и занести в них содержимое, переданное компоненту (через атрибуты его тега или непосредственно в его теге, подробности — в *разд. 9.1.1.3* и *9.1.1.4*).

   Получить доступ к отдельным элементам шаблона можно с помощью методов `getElementById()`, `querySelector()` и `querySelectorAll()` — класс `DocumentFragment` их поддерживает.

   Также класс `DocumentFragment` поддерживает свойства `children`, `childElementCount`, `firstElementChild` и `lastElementChild`.

5. Добавить готовую копию содержимого шаблона в созданную ранее скрытую DOM.

Листинг 10.2 показывает код класса компонента `BigText4`, использующий для вывода текста, заданного в атрибуте тега `text`, шаблон `BigText4Template` (см. листинг 10.1).

Листинг 10.2. Код класса компонента `BigText4`

```
class BigText4 extends HTMLElement {
    constructor() {
        super();
        // Получаем шаблон BigText4Template
        const oTemplate = document.getElementById('BigText4Template');
        // Создаем полную копию содержимого шаблона
        const oTC = oTemplate.content.cloneNode(true);
        // Получаем доступ к абзацу output, входящему в состав
        // содержимого шаблона
        const oOutput = oTC.getElementById('output');
        // Заносим в этот абзац текст, заданный в атрибуте text тега
        // компонента
        oOutput.textContent = this.getAttribute('text');
        // Создаем скрытую DOM
        const oShadow = this.attachShadow({ mode: 'closed' });
        // Добавляем в скрытую DOM готовую копию содержимого шаблона
        oShadow.appendChild(oTC);
    }
}

customElements.define('big-text4', BigText4);
```

Компонент, использующий шаблон, помещается на страницу так же, как и полностью формирующийся программно, — вставкой в нужное место HTML-кода страницы тега компонента:

```
<big-text4 text="Большой текст"></big-text4>
```

Описанный здесь подход имеет недостаток: для занесения содержимого в элементы внутренней структуры компонента также требуется программирование. Следовательно, если структура компонента изменится, скорее всего, понадобится переписывать класс компонента. Веб-верстальщик сделать это не сможет — придется привлекать веб-программиста.

10.2. Слоты

Решить указанную проблему помогут слоты.

> *Слот* — место в шаблоне компонента, в котором будет выводиться то или иное значение, заданное при размещении компонента на экране.

10.2.1. Создание слотов

Слот создается парным тегом `<slot>` в шаблоне компонента. Можно создать слот, относящийся к одной из двух разновидностей:

♦ *именованный* — имя такого слота указывается в атрибуте `name` тега `<slot>`.

> **ВНИМАНИЕ!**
>
> Имена слотов должны быть уникальны в пределах компонента, в котором они созданы.

  В шаблоне можно создать произвольное количество именованных слотов;

♦ *неименованный* — атрибут `name` в теге `<slot>` такого слота не указывается. В шаблоне может быть лишь один неименованный слот.

В тег слота помещается произвольное содержимое, которое будет выводиться на экран по умолчанию.

Листинг 10.3 показывает код шаблона компонента `BigText5`, в котором для указания места вывода заданной строки используется слот `output`.

Листинг 10.3. Код шаблона компонента `BigText5`

```
<template id="BigText5Template">
    <link href="big-text.css" type="text/css" rel="stylesheet">
    <p><slot name="output">Большой текст</slot></p>
</template>
```

Значения, заданные в теге компонента, заносятся в соответствующие слоты шаблона автоматически. Следовательно, в классе компонента нет нужды писать программный код, выполняющий эту задачу, что заметно упростит программирование.

Листинг 10.4 показывает код компонента `BigText5`, использующий для вывода строки шаблон `BigText5Template` со слотом `output`.

Листинг 10.4. Код класса компонента `BigText5`

```
class BigText5 extends HTMLElement {
    constructor() {
        super();
        const oTemplate = document.getElementById('BigText5Template');
        const oTC = oTemplate.content.cloneNode(true);
        const oShadow = this.attachShadow({ mode: 'closed' });
        oShadow.appendChild(oTC);
    }
}

customElements.define('big-text5', BigText5);
```

С помощью слотов можно указать лишь содержимое элементов, составляющих внутреннюю структуру компонента. Задать параметры этих элементов (например, значения атрибутов создающих их тегов) таким способом невозможно.

10.2.2. Использование слотов

Значения, выводимые в слотах, записываются непосредственно в теге компонента. Форма записи каждого значения зависит от того, в каком слоте следует его вывести:

- в именованном — значение помещается в произвольный тег, содержащий атрибут тега `slot`, в котором записывается имя нужного слота.

  Пример размещения на странице компонента `BigText5`:

  ```
  <big-text5><span slot="output">Большой текст</span></big-text5>
  ```

  Какой именно тег использовать для вывода значения, зависит от конкретного компонента. Например, если нужно просто вывести строку без какого-либо дополнительного оформления, рекомендуется применить тег `<span>` (как показано в приведенном примере), а если требуется вывести несколько абзацев — тег `<div>`. А применив, скажем, тег `<strong>`, можно вывести строку полужирным шрифтом:

  ```
  <big-text5><strong slot="output">Большой полужирный
                                         текст</strong></big-text5>
  ```

  Внимание!

  Тег, содержащий атрибут `slot`, будет вставлен в соответствующий слот целиком.

  Например, после создания экземпляра компонента `BigText5` с применением показанного ранее кода его внутренняя структура будет такова (вложенность элементов друг в друга показана отступами):

  ```
  <big-text5>
      <скрытая DOM>
          <link>
          <p>
              <slot id="output">
                  <span>
                      Большой текст
  ```

  В ряде случаев это может привести к проблемам (наподобие той, что будет описана далее, в *упражнении 10.3*);

- в неименованном — значение помещается непосредственно в тег компонента.

  Пример размещения на странице компонента `GreenText`, в котором для вывода строки применяется неименованный слот:

  ```
  <green-text>Зеленый текст</green-text>
  ```

При программном изменении какого-либо значения, выводимого в слоте, новое значение тотчас будет выведено на экран:

```
// Получаем доступ к тегу <span>, посредством которого задается
// выводящийся в компоненте BigText5 текст
const oBT5Output = document.querySelector('big-text5 span[slot=output]');
// Заносим в него другой текст — он сразу же будет выведен на экран
oBT5Output.innerHTML = 'Обновленный текст';
```

10.3. Упражнение. Пишем HTML-компонент — спойлер, вариант 2

Переделаем спойлер, написанный при выполнении *упражнения 9.3*, с использованием шаблона.

У шаблона компонента `HTMLSpoiler` укажем якорь `HTMLSpoilerTemplate`, чтобы в дальнейшем получить к нему доступ. Для вставки в спойлер заголовка используем именованный слот `header`, а для вставки содержимого — неименованный слот

1. Найдем в папке 9\ex9.3 сопровождающего книгу электронного архива (см. *приложение*) файлы 9.3.html (веб-страница с двумя заготовками для спойлеров), 9.3.css (таблица стилей с оформлением страницы), htmlspoiler.css (таблица стилей с оформлением спойлера) и htmlspoiler.js (код класса спойлера). Скопируем их куда-либо на локальный диск.
2. Откроем в текстовом редакторе копию страницы 9.3.html и вставим в начало секции тела страницы (тег `<body>`) код шаблона `HTMLSpoilerTemplate`:

```
<body>
    <template id="HTMLSpoilerTemplate">
        <link href="htmlspoiler.css" type="text/css" rel="stylesheet">
        <div class="spoiler">
            <a href="#" class="header">
                <slot name="header">Спойлер</slot>
            </a>
            <div class="content">
                <slot></slot>
            </div>
        </div>
    </template>
    . . .
</body>
```

3. Исправим HTML-код, создающий оба спойлера, чтобы он выглядел так:

```
<html-spoiler>
    <span slot="header">JavaScript</span>
    <!-- Содержимое первого спойлера -->
</html-spoiler>
<html-spoiler>
    <span slot="header">TypeScript</span>
    <!-- Содержимое второго спойлера -->
</html-spoiler>
```

Содержимым каждого спойлера являются два абзаца с кратким описанием соответствующего языка программирования.

4. Откроем в текстовом редакторе копию файла htmlspoiler.js и исправим конструктор класса HTMLSpoiler:

```
class HTMLSpoiler extends HTMLElement {
    . . .
    constructor() {
        super();
        const oTemplate =
              document.getElementById('HTMLSpoilerTemplate');
        const oTC = oTemplate.content.cloneNode(true);
        this.oDivWrapper = oTC.querySelector('div.spoiler');
        const oA = oTC.querySelector('a.header');
        oA.compSpoiler = this;
        oA.addEventListener('click', HTMLSpoiler__headerClick);
        const oShadow = this.attachShadow({ mode: 'closed' });
        oShadow.appendChild(oTC);
    }
    . . .
}
```

Не забываем занести в свойство oDivWrapper класса компонента ссылку на объект блока со стилевым классом spoiler — он понадобится для успешной работы компонента. Также не забываем создать в объекте гиперссылки со стилевым классом header (заголовке спойлера) свойство compSpoiler, хранящее ссылку на сам компонент, и привязать к этой гиперссылке обработчик события click.

В предыдущем варианте компонента текст заголовка помещался непосредственно в гиперссылке со стилевым классом header, и событие click при щелчке мышью возникало именно в ней. И в функции HTMLSpoiler__headerClick(), обработчике события click, для получения объекта гиперссылки-заголовка мы обращались к свойству target объекта события.

Теперь же текст заголовка, помещаемый в эту гиперссылку, будет заключен в тег <span> — именно его мы использовали в коде компонента для указания текста заголовка (см. код, написанный на *шаге 3*). В этом случае свойство target объекта события будет хранить ссылку на объект тега <span>, и ранее написанный код работать не станет. Чтобы в функции HTMLSpoiler__headerClick() получить объект гиперссылки, придется обращаться к переменной this.

5. Исправим код функции HTMLSpoiler__headerClick() — обработчика события click:

```
function HTMLSpoiler__headerClick(evt) {
    evt.preventDefault();
    this.compSpoiler.toggle();
}
```

Откроем страницу 9.3.html в веб-обозревателе и проверим, как работают оба спойлера.

10.4. Дополнительные инструменты для работы с шаблонами и слотами

Слот (тег `<slot>`) представляется объектом класса `HTMLSlotElement`. Этот класс поддерживает событие `slotchange`, возникающее при указании другого содержимого у текущего слота. Это событие обрабатывается в коде компонента. Пример:

```
function BigText5_outputChange() {
    // Каким-либо образом реагируем на изменение содержимого слота
}
class BigText5 extends HTMLElement {
    constructor() {
        . . .
        const oTC = oTemplate.content.cloneNode(true);
        const oOutput = oTC.querySelector('slot[name=output]');
        oOutput.addEventListener('slotchange', BigText5_outputChange);
        . . .
    }
}
```

Класс `HTMLElement`, содержащий основную функциональность элемента страницы, получил поддержку доступного только для чтения свойства `assignedSlot`. Оно хранит:

- ссылку на объект класса `HTMLSlotElement`, представляющий связанный с элементом слот (имя которого задано в атрибуте тега `slot`);
- `null` — если элемент не связан ни с одним слотом или если скрытая DOM компонента недоступна извне (при его создании параметру `mode` было дано значение `'closed'`, подробности — в *разд. 9.1.1.1*).

В языке CSS появилась поддержка псевдоэлемента `::slotted(<селектор>)`, который указывает на любой элемент, помещенный в слот и соответствующий заданному *селектору*. Этот псевдоэлемент используется «внутри» компонентов. Пример:

```
/* Таблица стилей big-text.css, «внутренняя» по отношению к компоненту
   BigText5 */
::slotted(em) {font-weight: bold;}
. . .
<big-text5><span slot="output"><em>Большой</em> текст</span></big-text5>
```

10.5. Самостоятельное упражнение

- Реализуйте у компонента `HTMLSpoiler` указание якоря используемого шаблона посредством атрибута `templateAnchor` тега компонента. Сделайте так, чтобы,

если этот атрибут не указан в теге, использовался шаблон с якорем `HTMLSpoilerTemplate`.

- Переделайте второй спойлер на странице 9.3.html таким образом, чтобы его заголовок находился под содержимым (рис. 10.1). (Подсказка: создайте еще один шаблон и укажите его якорь в атрибуте `templateAnchor` тега второго спойлера.)

[-] JavaScript

Мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262).

Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

Язык программирования, представленный Microsoft в 2012 году и позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript.

Разработчиком языка TypeScript является Андерс Хейлсберг (англ. Anders Hejlsberg), создавший ранее Turbo Pascal, Delphi и C#.

[-] TypeScript

Рис. 10.1. Результат выполнения самостоятельного упражнения: два разных спойлера

ЧАСТЬ IV
Мультимедиа

- Захват видео со встроенной камеры
- Запись видео и звука
- Получение параметров встроенной камеры
- Задание параметров захватываемых видео и звука
- Обработка звука
- Звуковые эффекты
- Визуализация звука

Урок 11
Работа со встроенной камерой, часть 1

Захват видео и звука
Запись видео и звука
Захват видео с экрана
Захват фото

Современные компьютеры часто имеют веб-камеру и микрофон для записи видео и видеотелефонии (иногда микрофон встроен в саму веб-камеру). Неудивительно, что современные веб-обозреватели обзавелись программными инструментами для захвата видео со встроенной камеры.

11.1. Захват видео с камеры

Мультимедийными инструментами веб-обозревателя «заведует» объект класса `MediaDevices`, создаваемый автоматически и хранящийся в свойстве `mediaDevices` класса `Navigator`. Объект класса `Navigator`, также создаваемый автоматически, хранится в свойстве `navigator` класса `Window`.

> ***Внимание!***
>
> Мультимедийные инструменты доступны только для страниц, загруженных либо с локального диска, либо с локального хоста, либо со сторонних веб-серверов *по протоколу HTTPS* (не HTTP!). Это сделано ради безопасности (в частности, для предотвращения слежки за пользователем через веб-камеру).

Проще всего захватить видео со встроенной камеры и вывести его в видеопроигрывателе HTML, создаваемом тегом `<video>`. Для этого необходимо выполнить действия, перечисленные далее.

1. Запросить у пользователя доступ к встроенной камере — вызвав метод `getUserMedia(<параметры>)` класса `MediaDevices`.

   `Параметры` указываются в виде служебного объекта со свойствами, одноименными с поддерживаемыми параметрами. Так, если указать в качестве *`параметров`* объект со свойствами `audio` и `video`, хранящими значения `true`, будет выполняться захват видео со звуком. Более подробно об указании параметров захватываемого видео будет рассказано на *уроке 12*.

После вызова этого метода веб-обозреватель выведет на экран окно с запросом на доступ к камере и микрофону (рис. 11.1). Сразу же после этого он возвращает промис, который:

- подтверждается — если пользователь ответит на запрос положительно, нажав кнопку **Разрешить**. В качестве нагрузки промис получит объект медиапотока, о котором разговор пойдет позже;
- отклоняется — если пользователь ответит на запрос отрицательно, нажав кнопку **Блокировать**, или при возникновении каких-либо проблем с камерой (например, встроенная камера отсутствует, или ее использование запрещено в настройках веб-обозревателя).

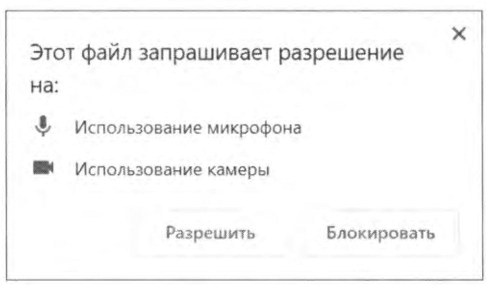


Рис. 11.1. Окно с запросом на доступ к камере и микрофону

2. При подтверждении полученного на *шаге 1* промиса — извлечь их него медиапоток, выдаваемый камерой и микрофоном.

   > *Медиапоток* — фиксируемые камерой и микрофоном видео и звук. Также, в зависимости от параметров, заданных в вызове метода `getUserMedia()`, может содержать только видео без звука или только звук.

   Медиапоток представляется объектом класса `MediaStream`.

3. Перенаправить полученный медиапоток видеопроигрывателю — присвоив его свойству `srcObject`, поддерживаемому классом `HTMLVideoElement`, который представляет тег `<video>`.

4. Запустить воспроизведение медиапотока — вызвав метод `play()` у объекта видеопроигрывателя. Этот метод следует вызывать в обработчике события `loadedmetadata` видеопроигрывателя, возникающего как только будет загружена начальная часть медиапотока, содержащая сведения о видео.

   Вместо этого можно вставить в тег `<video>`, создающий видеопроигрыватель, атрибут без значения `autoplay` — тогда проигрыватель сам запустит воспроизведение видео.

Пример захвата видео со звуком:

```
<video id="video" autoplay></video>
<input type="button" id="start" value="Старт!">
<input type="button" id="stop" value="Стоп!">
. . .
```

```
const oVideo = document.getElementById('video');
const btnStart = document.getElementById('start');
const btnStop = document.getElementById('stop');

// При нажатии кнопки Старт! начинаем захват видео
btnStart.addEventListener('click', () => {
    (async function () {
        const oStream = await navigator
                            .mediaDevices
                            .getUserMedia({ audio: true, video: true });
        oVideo.srcObject = oStream;
    })();
});

// При нажатии кнопки Стоп! останавливаем захват
btnStop.addEventListener('click', () => {
    // Для этого достаточно присвоить свойству srcObject
    // видеопроигрывателя значение undefined
    oVideo.srcObject = undefined;
});
```

Если доступ к камере не удалось получить, промис, возвращенный методом `getUserMedia()`, будет отклонен. При этом будет возбуждено одно из следующих отклоняющих исключений:

- `OverconstrainedError` — на компьютере отсутствует камера, соответствующая заданным в методе `getUserMedia()` параметрам;
- `TypeError` — в вызове метода `getUserMedia()` указаны некорректные параметры (например, свойствам `audio` и `video` объекта с параметрами были даны значения `false`);
- `NotAllowedError` — пользователь заблокировал доступ к камере либо в окне запроса (см. рис. 11.1), либо в настройках веб-обозревателя;
- `SecurityError` — была выполнена попытка получить доступ к камере на странице, загруженной со стороннего веб-сервера по протоколу HTTP;
- `NotFoundError` — камера кодирует видео и (или) звук алгоритмом, не поддерживаемым веб-обозревателем;
- `NotReadableError` — возникли проблемы с самой камерой или ее драйвером;
- `AbortError` — доступ к камере не может быть получен по неизвестной причине (возможно, камеру уже использует какая-то другая программа).

Пример обработки исключений:

```
try {
    const oStream = await navigator
                        .mediaDevices
                        .getUserMedia({ audio: true, video: true });
    . . .
}
```

```
catch(exception) {
    if (exception.name == 'OverconstrainedError')
        console.log('Нет подходящей камеры');
    else if (exception.name == 'NotAllowedError')
        console.log('Доступ к камере блокирован');
    else
        console.log('Ошибка: ' + exception.message;);
}
```

11.2. Запись видео

Записать видео, захватываемое с камеры, тоже несложно — для этого нужно выполнить действия, перечисленные далее.

1. Создать кодировщик — объект класса `MediaRecorder`, который и будет кодировать и записывать видео. Конструктор этого класса вызывается в следующем формате:

   `MediaRecorder(<медиапоток>, <параметры кодирования>=undefined)`

   Медиапоток задается в виде объекта класса `MediaStream`, полученного от метода `getUserMedia()` класса `MediaDevices` (см. *разд. 11.1*).

   Параметры кодирования задаются в виде служебного объекта со свойствами, одноименными с поддерживаемыми параметрами. Поддерживаются следующие параметры:

   - `mimeType` — формат файла с записываемым видео, обозначения алгоритмов кодирования для видео и звука. Путем экспериментов автору удалось установить значение этого параметра, дающее предсказуемый результат в разных веб-обозревателях:

     - Google Chrome — `'video/webm;codecs=vp9,opus'`;
     - Mozilla Firefox — `'video/webm;codecs:vp9,opus'`.

     Это значение указывает формат файлов WebM (MIME-тип — video/webm), алгоритм кодирования видео VP9 и алгоритм кодирования звука Opus.

     На самом деле веб-обозреватели поддерживают гораздо больше алгоритмов кодирования видео и звука, а некоторые (например, Chrome) — еще и дополнительные форматы файлов. Но если указать формат файлов и (или) алгоритмы кодирования, отличные от упомянутых ранее, Chrome по непонятной причине выберет для сохранения закодированного видео формат файла Matroska и алгоритм кодирования видео MPEG 4 AVC. Что может повлечь проблемы при дальнейшей обработке получившегося видеофайла на локальном компьютере или сервере.

     `mimeType` — единственный параметр, который настоятельно рекомендуется указывать (если этого не сделать, веб-обозреватель выберет параметры записываемого видео произвольно). Остальные параметры, приведенные далее, не обязательны к указанию;

- `videoBitsPerSecond` — битрейт видео в бит/с (по умолчанию: 2500000, т. е. 2,5 Мбит/с).

> *Битрейт* (или ширина потока данных) — объем информации в битах, отводимый под кодирование одной секунды видео или звука. Чем выше битрейт, тем выше качество видео (звука), но тем больше занимаемый им объем.

- `audioBitsPerSecond` — битрейт звука в бит/с (если не указан, подбирается самим веб-обозревателем);
- `bitsPerSecond` — совокупный битрейт видео и звука в бит/с. Если был указан только один из описанных ранее параметров — `videoBitsPerSecond` или `audioBitsPerSecond`, — используется для вычисления значения другого параметра. Если были указаны оба описанных ранее параметра, игнорируется.

2. Запустить запись видео — вызвав метод `start()` класса `MediaRecorder`:

   ```
   start([<продолжительность записи в один блок данных>=undefined])
   ```

   Закодированное видео помещается в особый блок данных, создаваемый в оперативной памяти. При запуске записи можно выбрать один из двух вариантов:

   - помещать записываемое видео в один блок — *не* указывая параметр `продолжительность записи в один блок данных` в вызове метода `start()`;
   - разбивать записываемое видео на отдельные отрезки одинаковой длины и помещать их в отдельные блоки — указав параметр `продолжительность записи в один блок данных` в вызове метода `start()`. Продолжительность указывается в миллисекундах.

   Использование первого варианта несколько упрощает программирование, использование второго — уменьшает потребление системных ресурсов, особенно при записи продолжительных видео.

3. Если видео записывается в отдельные блоки данных — получить очередной блок в обработчике события `dataavailable` класса `MediaRecorder`. Блок хранится в свойстве `data` объекта события.

   Для объединения отдельных блоков видео воедино можно объявить «пустой» массив и добавлять в него блоки вызовом метода `push()` у этого массива.

   Если же видео записывается в один блок, событие `dataavailable` в процессе записи не возникает.

4. Когда все видео записано — остановить запись, вызвав метод `stop()` класса `MediaRecorder`.

5. Получить блок данных с записанным видео — в обработчике события `dataavailable` класса `MediaRecorder`:

   - если видео записывается в один блок — это событие возникнет *единственный раз после остановки записи*. Полученный блок будет содержать все записанное видео;

- если видео записывается в отдельные блоки — событие `dataavailable` возникнет в последний раз после остановки записи. Полученный блок будет содержать остаток видео.

6. Преобразовать полученные видеоданные в файл, хранящийся в памяти, — представляемый объектом класса `Blob`. Формат конструктора этого класса:

   ```
   Blob(<данные, заносимые в файл>)
   ```

   Данные, заносимые в создаваемый файл, задаются в виде массива (например, содержащего отдельные блоки записанного видео) или строки.

7. Преобразовать полученный файл в data URL — вызовом статического метода `createObjectURL(<преобразуемый файл>)` класса `URL`. Метод возвращает data URL в виде строки.

   Этот data URL можно использовать для воспроизведения записанного видео в видеопроигрывателе (теге `<video>`) или для сохранения на локальном диске.

8. После окончания работы с полученным ранее data URL — удалить его, вызвав статический метод `revokeObjectURL(<удаляемый data URL>)` класса `URL`.

Пример записи видео со звуковым сопровождением:

```
const oStream = await navigator
                     .mediaDevices
                     .getUserMedia({ audio: true, video: true });
// Предполагается, что запись видео ведется в Google Chrome.
// Универсальный код, подходящий и для Mozilla Firefox, мы напишем потом.
// Создаем объект, который будет записывать видео.
oRec = new MediaRecorder(oStream,
                        { mimeType: 'video/webm;codecs=vp9,opus' });
// Привязываем к нему обработчик события dataavailable, в котором будем
// получать записанное видео. Предполагается, что запись будет вестись
// в один блок — тогда мы сможем получить в этом обработчике все видео.
oRec.addEventListener('dataavailable', (evt) => {
    // Преобразуем блок видеоданных в файл, хранящийся в памяти.
    // Поскольку конструктор класса Blob принимает только массивы и
    // строки, преобразуем блок данных с видео в массив из одного
    // элемента.
    const oBlob = new Blob([evt.data]);
    // Преобразуем файл в data URL
    const durl = URL.createObjectURL(oBlob);
    // Выводим записанное видео в видеопроигрывателе
    oVideo.src = durl;
});
// Все готово? Тогда начинаем запись.
oRec.start();
. . .
// В нужный момент останавливаем запись
oRec.stop();
```

Класс `MediaRecorder` поддерживает ряд полезных методов, событий и свойств. Полезные методы:

- `pause()` — приостанавливает запись видео;
- `resume()` — возобновляет запись видео после приостановки;
- `isTypeSupported(<формат видео>)` — статический, возвращает `true`, если заданный в виде строки формат записи видео поддерживается веб-обозревателем, и `false` — в противном случае:

```
MediaRecorder.isTypeSupported('video/mp4;codecs=vp9,opus');
                                        // Результат (Chrome):  true
                                        // Результат (Firefox): false
MediaRecorder.isTypeSupported('video/mp4;codecs:vp9,opus');
                                        // Результат (Chrome):  false
                                        // Результат (Firefox): true
```

События:

- `start` — возникает после начала записи видео;
- `pause` — возникает после приостановки записи видео;
- `resume` — возникает после возобновления записи видео после приостановки;
- `stop` — возникает после остановки записи видео;
- `error` — возникает при появлении проблемы в процессе записи видео. Свойство `error` объекта события хранит объект возникшего исключения.

Свойства — все доступны только для чтения:

- `state` — состояние текущего объекта класса `MediaRecorder` в виде одной из строк:
  - `'inactive'` — запись либо еще не началась, либо уже завершилась;
  - `'recording'` — запись выполняется;
  - `'paused'` — запись приостановлена;
- `mimeType` — обозначение формата записываемого видео, заданное при вызове конструктора класса `MediaRecorder`;
- `stream` — медиапоток, заданный в вызове конструктора;
- `videoBitsPerSecond` — текущий битрейт видео в бит/с (может отличаться от заданного в вызове конструктора);
- `audioBitsPerSecond` — текущий битрейт звука в бит/с (может отличаться от заданного в вызове конструктора).

11.3. Упражнение. Пишем веб-приложение для записи видео

Напишем веб-приложение для записи видео со звуком со встроенной камеры. Реализуем в нем показ изображения, захватываемого камерой, воспроизведение записанного видео, приостановку и возобновление записи, а также сохранение записанного видео на локальном диске в файле под автоматически генерируемым именем.

Приложение будет вести запись видео отдельными блоками продолжительностью в 1 с (1000 мс). Оно будет работать и в Chrome, и в Firefox.

1. Найдем в папке 11\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы 11.3.html (веб-страница, на основе которой будет писаться приложение) и 11.3.css (таблица стилей с оформлением страницы). Скопируем их куда-либо на локальный диск.

   Страница 11.3.html содержит:

   - видеопроигрыватель контрольный (показывает то, что в текущий момент снимает камера) — слева — с якорем `control`. В теге видеопроигрывателя указаны атрибуты тегов `autoplay` (для автоматического запуска воспроизведения) и `muted` (чтобы приглушить звук);
   - видеопроигрыватель, воспроизводящий записанное видео, — справа — с якорем `recorded`;
   - кнопку **Старт**, запускающую запись видео, — `start`;
   - кнопку **Пауза** — `pause`;
   - кнопку **Возобновление** — `resume`;
   - кнопку **Стоп**, останавливающую запись видео, — `stop`;
   - кнопку **Сохранить**, сохраняющую записанное видео на локальный диск, — `save`;
   - кнопку **Сброс**, удаляющую записанное ранее видео и подготавливающую приложение к записи нового, — `reset`.

     Все эти кнопки изначально недоступны для пользователя;
   - гиперссылку, применяемую для сохранения видео на локальном диске и не отображаемую на экране, — `download`.

   Еще на странице присутствует изначально недоступная кнопка **Выгрузить** с якорем `upload`. Она понадобится нам при выполнении следующего упражнения.

2. Откроем в текстовом редакторе копию страницы 11.3.html и вставим в самый низ ее кода привязку файла веб-сценария 11.3.js, который скоро создадим:

```
<html>
    . . .
</html>
<script src="11.3.js" type="text/javascript"></script>
```

3. Создадим в той же папке, где хранятся копии файлов 11.3.html и 11.3.css, файл 11.3.js, откроем его в текстовом редакторе и запишем выражения, получающие доступ к ранее упомянутым элементам страницы 11.3.html:

   ```
   const oControl = document.getElementById('control');
   const oRecorded = document.getElementById('recorded');
   const btnStart = document.getElementById('start');
   const btnPause = document.getElementById('pause');
   const btnResume = document.getElementById('resume');
   const btnStop = document.getElementById('stop');
   const btnSave = document.getElementById('save');
   const btnReset = document.getElementById('reset');
   const aDownload = document.getElementById('download');
   ```

   Объявим переменные `videoParams` (для хранения объекта с параметрами записываемого видео), `oRec` (объект-кодировщик), `chunks` (массив для хранения отдельных блоков с записанным видео, изначально «пустой»), `oBlob` (файл, хранящий записанное видео) и `durl` (data URL этого файла).

4. Добавим выражение, объявляющее указанные переменные:

   ```
   let videoParams, oRec, chunks = [], oBlob, durl;
   ```

   Чтобы сформировать служебный объект с параметрами записываемого видео, который передается конструктору класса `MediaRecorder`, нужно знать, в каком веб-обозревателе работает приложение: Chrome или Firefox (о задании параметров видео рассказывалось в *разд. 11.2*). Выяснить это можно, получив строку с описанием веб-обозревателя из свойства `userAgent` класса `Navigator` и проверив, присутствует ли в ней подстрока `'Firefox'`. Если такая подстрока присутствует, значит, приложение работает в Firefox, в противном случае — в Chrome.

5. Добавим код, создающий служебный объект с параметрами записываемого видео:

   ```
   if (navigator.userAgent.includes('Firefox'))
       videoParams = { mimeType: 'video/webm;codecs:vp9,opus' };
   else
       videoParams = { mimeType: 'video/webm;codecs=vp9,opus' };
   ```

   Далее попытаемся получить доступ к встроенной камере. Поскольку это асинхронная операция, заключим выполняющий ее код в асинхронное замыкание — для упрощения программирования.

   После получения доступа к камере подключим ее к левому, контрольному, видеопроигрывателю.

6. Добавим асинхронное замыкание с кодом, получающим доступ к камере и подключающим ее к контрольному видеопроигрывателю:

   ```
   (async function () {
       const oStream = await navigator
                             .mediaDevices
                             .getUserMedia({ audio: true, video: true });
   ```

```
    oControl.srcObject = oStream;
})();
```

7. Добавим в асинхронное замыкание выражение, создающее объект-кодировщик и присваивающий его переменной oRec:

```
(async function () {
    . . .
    oRec = new MediaRecorder(oStream, videoParams);
})();
```

8. Добавим в асинхронное замыкание обработчик события dataavailable объекта oRec, помещающий блоки с записанным видео в массив chunks:

```
(async function () {
    . . .
    oRec.addEventListener('dataavailable', (evt) => {
        chunks.push(evt.data);
    });
})();
```

Теперь нужно привязать к кодировщику обработчики следующих событий (подробности о них — в *разд. 11.2*):

- start — сделает кнопку **Старт** недоступной, а кнопки **Пауза** и **Стоп** доступными;
- pause — сделает кнопку **Пауза** недоступной, а кнопку **Возобновление** — доступной;
- resume — сделает кнопку **Возобновление** недоступной, а кнопку **Пауза** — доступной.

9. Добавим в асинхронное замыкание код, привязывающий к объекту oRec обработчики всех этих событий:

```
(async function () {
    . . .
    oRec.addEventListener('start', () => {
        btnStart.disabled = true;
        btnPause.disabled = false;
        btnStop.disabled = false;
    });
    oRec.addEventListener('pause', () => {
        btnPause.disabled = true;
        btnResume.disabled = false;
    });
    oRec.addEventListener('resume', () => {
        btnPause.disabled = false;
        btnResume.disabled = true;
    });
})();
```

Обработчик события stop кодировщика будет самым сложным. Он сформирует на основе массива chunks файл с записанным видео, создаст на его основе data URL, занесет последний в правый видеопроигрыватель (чтобы пользователь смог просмотреть записанное видео), сделает кнопки **Пауза**, **Возобновление** и **Стоп** недоступными, а кнопки **Сохранить** и **Сброс** — доступными.

А еще он сделает так, чтобы пользователь впоследствии смог сохранить записанное видео на локальном диске. Для этого он занесет data URL видеофайла в атрибут href тега скрытой гиперссылки download, после чего присвоит автоматически сгенерированное имя, под которым будет сохранен этот файл, атрибуту download тега той же гиперссылки (если не задать этот атрибут тега, веб-обозреватель не сохранит целевой файл на диске, а откроет в своем окне). Имя файла формируется в виде псевдослучайного целого числа с добавленным расширением webm.

10. Добавим в асинхронное замыкание код, привязывающий к объекту oRec обработчик события stop:

```
(async function () {
    . . .
    oRec.addEventListener('stop', () => {
        btnPause.disabled = true;
        btnResume.disabled = true;
        btnStop.disabled = true;
        oBlob = new Blob(chunks);
        durl = URL.createObjectURL(oBlob);
        oRecorded.src = durl;
        aDownload.download = Math.ceil(Math.random() *
                             10000000000000000) + '.webm';
        aDownload.href = durl;
        btnSave.disabled = false;
        btnReset.disabled = false;
    });
})();
```

Закончив подготовительные действия, можем разрешить пользователю записывать видео.

11. Добавим в асинхронное замыкание выражение, делающее доступной кнопку **Старт**:

```
(async function () {
    . . .
    btnStart.disabled = false;
})();
```

Остальной код будет находиться вне асинхронного замыкания.

12. Добавим код, привязывающий к кнопкам **Старт**, **Пауза**, **Возобновление** и **Стоп** обработчики событий click, которые будут выполнять соответствующие кнопкам действия:

```
(async function () {
    . . .
})();
btnStart.addEventListener('click', () => {
    oRec.start(1000);
});
btnPause.addEventListener('click', () => {
    oRec.pause();
});
btnResume.addEventListener('click', () => {
    oRec.resume();
});
btnStop.addEventListener('click', () => {
    oRec.stop();
});
```

Чтобы реализовать сохранение записанного видео, достаточно программно инициировать щелчок на гиперссылке `download` вызовом метода `click()`.

13. Добавим код, привязывающий к кнопке **Сохранить** обработчик события `click`, который выполнит сохранение видеофайла на локальном диске:

```
btnSave.addEventListener('click', () => {
    aDownload.click();
});
```

После нажатия кнопки **Сброс** следует убрать из правого видеопроигрывателя записанное видео, удалить data URL, сформированный в памяти видеофайл, массив `chunks`, сделать кнопки **Сохранить** и **Сброс** недоступными, а кнопку **Старт** — доступной. В общем, подготовить приложение к записи нового видео.

14. Добавим код, привязывающий к кнопке **Сброс** обработчик события `click`, который выполнит сброс:

```
btnReset.addEventListener('click', () => {
    oRecorded.src = '';
    URL.revokeObjectURL(durl);
    durl = undefined;
    oBlob = undefined;
    chunks = [];
    btnReset.disabled = true;
    btnSave.disabled = true;
    btnStart.disabled = false;
});
```

Откроем страницу 11.3.html в веб-обозревателе и разрешим доступ к камере. Попробуем записать несколько видео подряд, приостанавливая и возобновляя запись, просмотреть и сохранить эти видео на локальном диске (рис. 11.2). Проверим это приложение в Google Chrome и Mozilla Firefox.

Рис. 11.2. Веб-приложение для записи видео со встроенной камеры

11.4. Упражнение. Реализуем выгрузку записанного видео на сервер

Добавим веб-приложению, написанному при выполнении *упражнения 11.3*, возможность выгружать записанное видео на сервер.

На странице приложения уже присутствует изначально недоступная кнопка **Выгрузить** с якорем `upload` — она-то и станет выполнять выгрузку. Файл с выгруженным видео будет сохраняться под своим изначальным именем в папке uploads, вложенной в корневую папку сайта. Сразу после выгрузки приложение будет выполнять сброс (как если бы пользователь нажал кнопку **Сброс**).

1. Найдем в папке 11\ex11.3 сопровождающего книгу электронного архива (см. *приложение*) файлы 11.3.html, 11.3.css и 11.3.js, содержащие код ранее написанного приложения. Скопируем их куда-либо на локальный диск.

2. Найдем в папке 11\!sources сопровождающего книгу электронного архива (см. *приложение*) файл 11.4.php с кодом бэкенда. Скопируем его в ту же папку, где хранятся ранее созданные копии файлов 11.3.html, 11.3.css и 11.3.js.

   Бэкенд 11.4.php сохранит выгруженный файл, взяв его из POST-параметра `video`, в папку uploads, находящуюся в корневой папке сайта. Если эта папка не существует, бэкенд создаст ее. Код бэкенда достаточно тривиален, и мы не будем его здесь рассматривать.

3. Откроем в текстовом редакторе копию файла 11.3.js и добавим выражение, получающее доступ к кнопке **Выгрузить**:

   ```
   . . .
   const btnSave = document.getElementById('save');
   const btnUpload = document.getElementById('upload');
   const btnReset = document.getElementById('reset');
   . . .
   ```

   После остановки записи и формирования видеофайла следует сделать эту кнопку доступной, чтобы пользователь смог выгрузить видео.

4. Добавим в обработчик события `stop` объекта `oRec` выражение, делающее кнопку **Выгрузить** доступной:

```
oRec.addEventListener('stop', () => {
    . . .
    btnSave.disabled = false;
    btnUpload.disabled = false;
    btnReset.disabled = false;
});
```

После сброса приложения эту кнопку нужно сделать, наоборот, недоступной.

5. Добавим в обработчик события `click` кнопки **Сброс** выражение, делающее кнопку **Выгрузить** недоступной:

```
btnReset.addEventListener('click', () => {
    . . .
    btnReset.disabled = true;
    btnUpload.disabled = true;
    btnSave.disabled = true;
    btnStart.disabled = false;
});
```

Выгружать файл будем с помощью загрузчика данных AJAX, рассмотренного на *уроке 6*. При его вызове следует указать метод загрузки данных POST. Сам выгружаемый файл поместим в объект класса `FormData`, содержимое файла возьмем из переменной `oBlob`, а его имя — из атрибута `download` тега скрытой гиперссылки `download`. После выгрузки файла выполним сброс, программно инициировав щелчок на кнопке **Сброс** вызовом метода `click()`. Поскольку операция выгрузки файла асинхронна, функцию-обработчик события `click`, которая и выгрузит файл, сделаем асинхронной.

6. Добавим в файл 11.3.js обработчик события `click` кнопки **Выгрузить**, который выполнит выгрузку записанного видеофайла:

```
btnUpload.addEventListener('click', async function () {
    const oFD = new FormData();
    oFD.append('video', oBlob, aDownload.download);
    await fetch('/11.4.php', { method: 'post', body: oFD });
    btnReset.click();
});
```

Для эксперимента попробуем войти в наше приложение по протоколу HTTPS.

7. Скопируем файлы 11.3.html, 11.3.css, 11.3.js и 11.4.php в корневую папку веб-сервера, запустим сервер, откроем веб-обозреватель и перейдем по интернет-адресу **https://localhost/11.3.html**. Появится страница с предупреждением о незащищенном подключении (рис. 11.3).

   Эта страница выводится потому, что мы не настроили веб-сервер для постоянной эксплуатации с применением защищенного протокола HTTPS, в частности, не указали сертификат безопасности. Впрочем, для отладки это необязательно.

8. Нажмем на появившейся странице кнопку **Дополнительные**.

   Появится другая страница — с более детальным описанием возникшей проблемы (рис. 11.4).

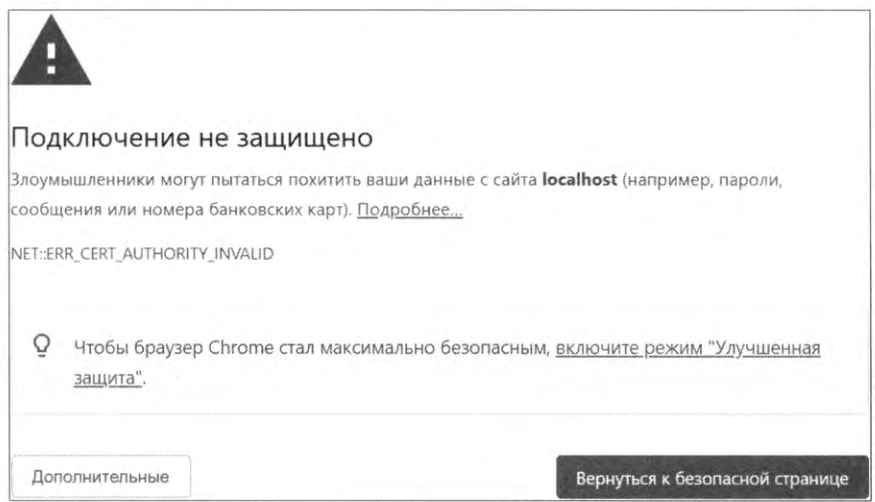


Рис. 11.3. Веб-страница с предупреждением о незащищенном подключении

Не удалось подтвердить, что это сервер **localhost**. Операционная система компьютера не доверяет его сертификату безопасности. Возможно, сервер настроен неправильно или кто-то пытается перехватить ваши данные.

Перейти на сайт localhost (небезопасно)

Скрыть подробности

Вернуться к безопасной странице

Рис. 11.4. Веб-страница с более детальным описанием возникшей проблемы

9. Щелкнем на гиперссылке **Перейти на сайт <*интернет-адрес*> (небезопасно)**.

   После этого будет выполнен переход на наше веб-приложение.

   Запишите какое-либо видео и выгрузите его на сервер. Проверьте, появилась ли в корневой папке сайта папка uploads, а в ней — выгруженный файл.

   Опять же, ради эксперимента, можно попробовать открыть веб-приложение с другого компьютера, смартфона или планшета. Для этого понадобится узнать IP-адрес текущего компьютера, на котором работает веб-сервер.

10. На текущем компьютере — запустим командную строку и отдадим в ней команду `ipconfig`.

    Эта команда выведет основные сведения о сетевой конфигурации текущего компьютера, в том числе его IP-адрес (подчеркнут):

```
DNS-суффикс подключения . . . . . :
Локальный IPv6-адрес канала . . . : fe80::589a:395d:67c1:3314%4
IPv4-адрес. . . . . . . . . . . . : 192.168.1.2
Маска подсети . . . . . . . . . . : 255.255.255.0
Основной шлюз. . . . . . . . . : 192.168.1.1
```

11. На другом компьютере — откроем веб-обозреватель и перейдем по интернет-адресу **https://<*полученный ранее IP-адрес*>/11.3.html** (например, **https://192.168.1.2/11.3.html**).

    Веб-обозреватель также выведет страницу с сообщением о незащищенном подключении, и для перехода на приложение придется выполнить описанные ранее действия.

> ***Полезно знать...***
>
> - При повторном заходе на локальный хост по протоколу HTTPS веб-обозреватель уже не будет предупреждать о незащищенном подключении. Он посчитает, что если пользователь явно признал сайт защищенным, значит, так оно и есть, и отметит это в своей конфигурации.
> - При повторном открытии веб-приложения, использующего камеру, по протоколу HTTPS также не понадобится снова давать разрешение на ее использование.
> - IP-адреса, начинающиеся цифрами **192.168**, служат для идентификации компьютеров во внутренней локальной сети. Извне по этим IP-адресам получить доступ к компьютерам невозможно.

11.5. Захват и запись звука

Захват и запись звука с микрофона (встроенного в камеру или отдельного) выполняется почти так же, как и захват и запись видео, за следующими исключениями:

- при попытке получить доступ к микрофону — в вызове метода `getUserMedia()` класса `MediaDevices` в объекте с параметрами дать свойству `video` значение `false`;
- при создании объекта, записывающего звук, — в вызове конструктора класса `MediaRecorder` в объекте с параметрами дать свойству `mediaType` значение `'audio/webm;codecs=opus'` (формат файла WebM, алгоритм кодирования звука Opus). Как показывает практика, эти параметры успешно работают и в Chrome, и в Firefox.

Пример кода, выполняющего захват звука:

```
<audio id="audio" autoplay></audio>
. . .
const oAudio = document.getElementById('audio');
(async function () {
    const oStream = await navigator
                          .mediaDevices
                          .getUserMedia({ audio: true, video: false });
    oAudio.srcObject = oStream;
})();
```

Пример кода, запускающего запись звука:

```
const oStream = await navigator.mediaDevices
                        .getUserMedia({ audio: true, video: false});
oRec = new MediaRecorder(oStream, { mimeType: 'audio/webm;codecs=opus' });
```

11.6. Захват видео с экрана

Видео можно захватывать не только с камеры, но и с экрана компьютера — записывая все, что происходит на нем. Можно захватывать видео как со звуком, так и без него.

Для получения доступа к подсистеме, записывающей видео с экрана, служит метод `getDisplayMedia()` класса `MediaDevices`. Формат его вызова аналогичен таковому у метода `getUserMedia()` (см. *разд. 11.1*).

Если нужно захватить видео с экрана вместе со звуком, свойству `audio` объекта с параметрами, передаваемому методу `getDisplayMedia()`, нужно дать значение `true`. В этом случае будут захватываться все звуки, издаваемые операционной системой и отдельными приложениями, но не звук с микрофона (зачем это сделано, непонятно). Если же звуковое сопровождение не требуется, следует дать свойству `audio` значение `false`.

Метод `getDisplayMedia()` также возвращает промис и выводит на экран окно с запросом, с какого источника следует захватывать видео. В Google Chrome оно выглядит так, как показано на рис. 11.5.

В этом окне присутствуют три вкладки:

- **Весь экран** — содержит список мониторов, подключенных к компьютеру (на рис. 11.5 выбрана эта вкладка, и список показывает единственный монитор, ко-

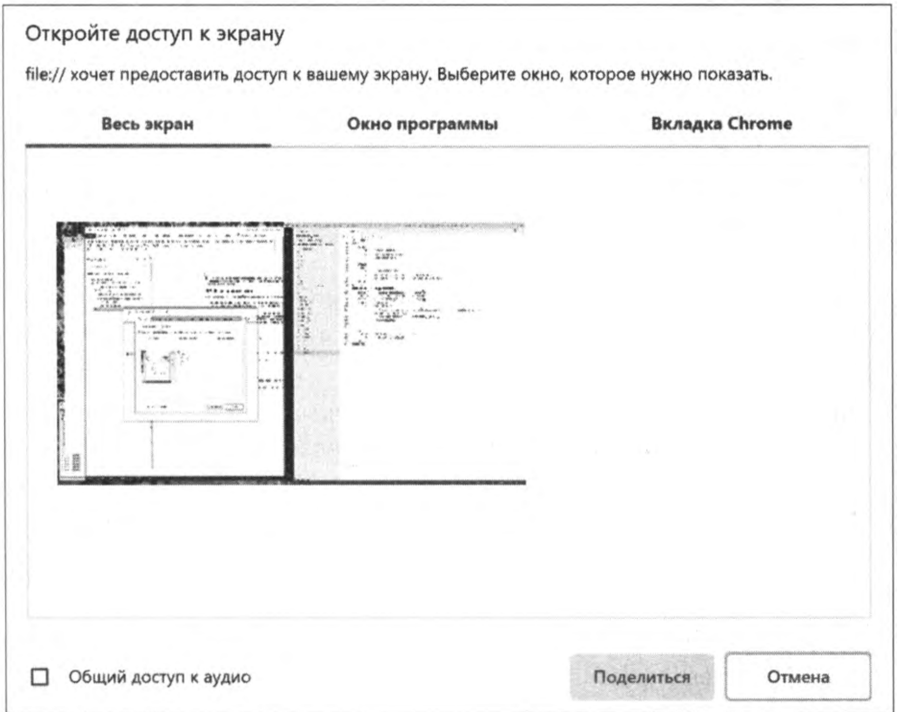


Рис. 11.5. Окно с запросом источника в Google Chrome: вкладка **Весь экран**

торый имеется у автора). Захватываемое видео будет показывать все, что происходит на выбранном в этом списке мониторе;

- **Окно программы** — содержит список окон, выведенных запущенными приложениями (рис. 11.6). Захватываемое видео будет содержать все, что происходит в выбранном окне;

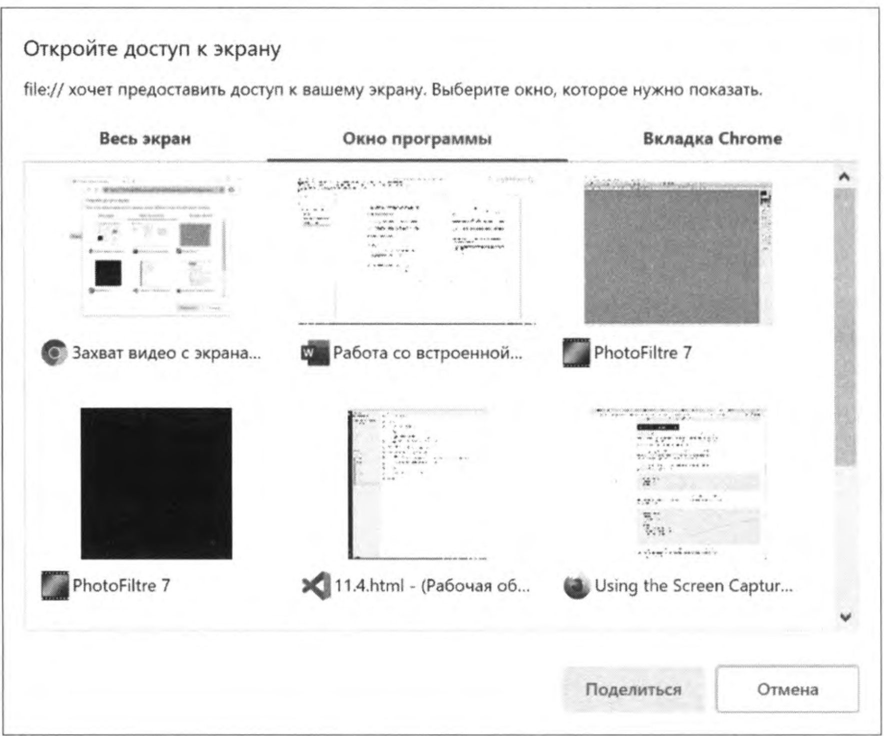


Рис. 11.6. Окно с запросом источника в Google Chrome: вкладка **Окно программы**

- **Вкладка Chrome** — содержит список вкладок текущего веб-обозревателя (рис. 11.7). Захватываемое видео будет содержать все, что происходит на выбранной вкладке.

Флажок **Общий доступ к аудио** присутствует лишь в том случае, если также выполняется захват звука. Его установка разрешает захват звука (если флажок сброшен, звук захватываться не будет).

Кнопка **Поделиться**, собственно, разрешает захват видео с экрана. Она становится доступной лишь при выборе нужного монитора, окна или вкладки веб-обозревателя. Кнопка **Отмена** отклоняет запрос на захват видео.

В Mozilla Firefox окно с запросом источника выглядит так, как показано на рис. 11.8. Источник выбирается в раскрывающемся списке (на рис. 11.8 он раскрыт). Здесь вариантов меньше — так, из всех вкладок, открытых в веб-обозревателе, видео можно захватывать лишь с активной в текущий момент. Кнопка

Разрешаю разрешает захват видео, кнопка **Не сейчас** отклоняет запрос (на рис. 11.8 обе кнопки скрыты развернутым списком).

Захваченное с экрана видео можно вывести в видеопроигрывателе (теге `<video>`) и записать, использовав программные инструменты, описанные в *разд. 11.2*.

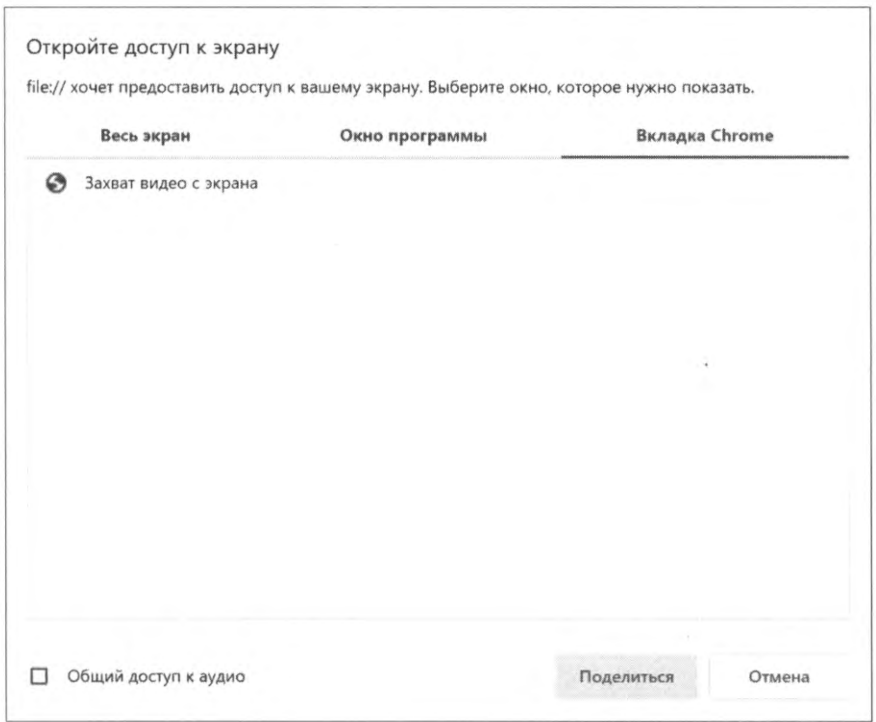


Рис. 11.7. Окно с запросом источника в Google Chrome: вкладка **Вкладка Chrome**

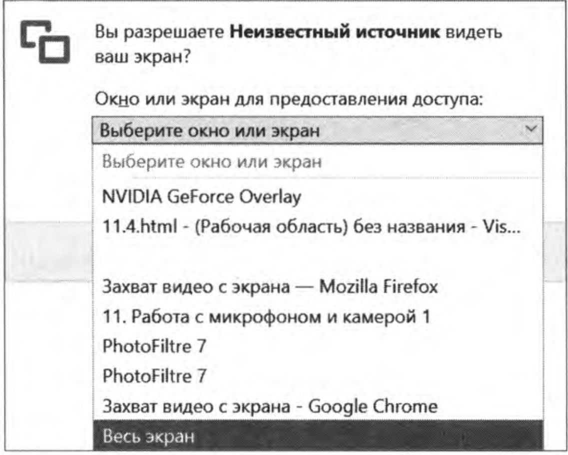


Рис. 11.8. Окно с запросом источника в Mozilla Firefox

11.7. Захват статичных изображений

И наконец, с камеры, равно как и с экрана, можно захватывать статичные изображения — фото.

> **ВНИМАНИЕ!**
> Захват статичных изображений в настоящее время не поддерживается Mozilla Firefox.

Чтобы захватить статичное изображение, следует выполнить действия, перечисленные далее.

1. Получить медиапоток с камеры или экрана — средствами, описанными в *разд. 11.1* и *11.6* соответственно.
2. Извлечь из медиапотока видеодорожку.

   Сначала следует вызвать метод `getVideoTracks()` класса `MediaStream`. Он вернет коллекцию всех видеодорожек, что присутствуют в текущем медиапотоке, и практически во всех случаях эта коллекция будет содержать одну видеодорожку.

   Видеодорожка будет представлена объектом класса `MediaStreamTrack`.
3. Создать кодировщик изображений — объект класса `ImageCapture`. Формат вызова конструктора этого класса: `ImageCapture(<видеодорожка>)`. Видеодорожка, из которой будут захватываться изображения, должна быть представлена объектом класса `MediaStreamTrack`.
4. Захватить изображение — вызвав метод `takePhoto([<параметры>=undefined])` класса `ImageCapture`.

   Параметры задаются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Например, чтобы включить функцию коррекции «красных глаз», в служебный объект следует добавить свойство `redEyeReduction`, имеющее значение `true`. Более подробно параметры захватываемых статичных изображений будут рассмотрены на *уроке 12*.

   Метод возвращает промис, который подтверждается в момент получения захваченного изображения. Это изображение, представленное объектом класса `Blob`, заносится в промис в качестве нагрузки.

   Полученный объект класса `Blob` представляет собой хранящийся в памяти файл с изображением формата PNG (указать другой формат для изображения, судя по всему, в настоящее время невозможно). Его можно без изменений отправить серверу (см. *упражнение 11.4*), или преобразовать в data URL, чтобы вывести на экран в теге `<img>`, или занести в гиперссылку для последующего сохранения на локальном диске.

Пример захвата изображения и вывода его в теге `<img>`:

```
<input type="button" id="capture" value="Сделай фото!">
<img src="#" id="output">
. . .
```

```
const btnCapture = document.getElementById('capture');
const oOutput = document.getElementById('output');
let oIC;
(async function () {
    // Получаем доступ к камере для захвата только видео
    // (звук нам ни к чему)
    const oStream = await navigator.mediaDevices
                        .getUserMedia({ audio: false, video: true});
    // Извлекаем видеодорожку из медиапотока
    const oTrack = oStream.getVideoTracks()[0];
    // Создаем кодировщик изображений
    oIC = new ImageCapture(oTrack);
})();
btnCapture.addEventListener('click', async function() {
    // Делаем фото
    const oBlob = await oIC.takePhoto();
    // Преобразуем его в data URL и выводим в теге <img>
    const durl = URL.createObjectURL(oBlob);
    oOutput.src = durl;
});
```

11.8. Самостоятельные упражнения

Используя веб-приложение для видеозаписи, написанное при выполнении *упражнения 11.3*, как основу, напишите два веб-приложения, соответственно, для:

- записи звука с микрофона. Как и приложение видеозаписи, оно должно позволять приостанавливать и возобновлять запись звука, воспроизводить записанный звук, сохранять его на локальном диске и выгружать на сервер. Сохраните код этого приложения в файлах 11.8.1.html, 11.8.1.css, 11.8.1.js и 11.8.1.php;
- фотосъемки. Оно должно выводить сделанное фото, сохранять его на локальном диске и выгружать на сервер. Сохраните код этого приложения в файлах 11.8.2.html, 11.8.2.css, 11.8.2.js и 11.8.2.php.

Урок 12
Работа со встроенной камерой, часть 2

Параметры видео и звука, доступные для указания

Получение сведений о возможностях камеры и микрофона

Параметры статичных изображений

12.1. Задание параметров видео и звука

Параметры записываемых видео и звука задаются в служебном объекте, передаваемом методу `getUserMedia()` (см. *разд. 11.1*) или `getDisplayMedia()` (см. *разд. 11.6*) при вызове. Можно указать, в частности, что именно требуется захватить — только видео, только звук или видео со звуком, характеристики захватываемых видео и звука (разрешение видео, количество каналов звука и др.), а также активизировать различные дополнительные функции камеры и микрофона (например, коррекцию «красных глаз» или подавление шума).

12.1.1. Параметры, доступные для указания

Служебный объект, передаваемый методам `getUserMedia()` и `getDisplayMedia()`, должен содержать два свойства:

- `video` — может принимать три значения:
  - `true` — захватить видео наилучшего качества, какое может обеспечить камера;
  - служебный объект с параметрами — захватить видео, максимально соответствующее заданным в этом объекте параметрам;
  - `false` — вообще не захватывать видео;
- `audio` — может принимать три значения:
  - `true` — захватить звук наилучшего качества, какое может обеспечить микрофон;
  - служебный объект с параметрами — захватить звук, максимально соответствующий заданным в этом объекте параметрам;
  - `false` — вообще не захватывать звук.

Внимание!

Параметры видео и звука, указываемые при захвате, носят лишь рекомендательный характер. Если камера и микрофон не могут выдать видео и звук с заданными параметрами, веб-обозреватель выполнит захват с характеристиками, максимально близкими к заданным.

Например, если указать разрешение видео 1920×1080 пикселов (Full HD), а камера имеет разрешение 640×480 пикселов (VGA), захватываемое видео будет иметь разрешение 640×480.

Как правило, камеры и микрофоны поддерживают предопределенный набор комбинаций параметров видео и звука — *пресетов*. Так, камера, имеющаяся у автора, поддерживает два пресета видео: с разрешением 640×480 и 320×240 пикселов.

Далее будут рассмотрены параметры видео и звука, которые можно указать в служебных объектах, присваиваемых свойствам `video` и `audio`.

12.1.1.1. Параметры видео

- `width` — ширина видео. Может быть задана в виде:
  - целого числа в пикселах — будет захвачено видео с шириной, равной или максимально близкой к указанной:

    ```
    const oParams = { video: { width: 1280 } };
    ```

  - служебного объекта со свойствами, значения которых задаются в виде целых чисел в пикселах:
    - `exact` — строго заданная ширина видео. Если камера не может захватывать видео указанной ширины, попытка подключиться к камере не увенчается успехом. Пример:

      ```
      const oParams2 = { video: { width: { exact: 1280 } } };
      ```

      Можно указать либо свойство `exact`, либо следующие три свойства:
    - `min` — минимальная ширина видео;
    - `ideal` — желаемая ширина видео;
    - `max` — максимальная ширина видео.

      В этом случае веб-обозреватель выберет пресет с шириной видео, указанной в свойстве `ideal`, а если такой не поддерживается, подберет другой — с шириной, расположенной между указанными в свойствах `min` и `max` величинами. Пример:

      ```
      const oParams3 = { video: { width: { min: 640, ideal: 1280,
                                           max: 1920 } } };
      ```

      Эти три свойства являются необязательными для указания. Пример задания только минимальной ширины видео:

      ```
      const oParams4 = { video: { width: { min: 640 } } };
      ```

- `height` — высота видео. Может быть задана в виде:
  - целого числа в пикселах — будет захвачено видео с высотой, равной или максимально близкой к указанной;

- служебного объекта со свойствами exact, min, ideal и max, описанными ранее, — будет захвачено видео с высотой, соответствующей заданным в этом объекте параметрам.

  Пример:

```
const oParams5 = { video: { width:  { min: 640, ideal: 1280; max: 1920 },
                            height: { min: 480, ideal: 720, max: 1080 } } };
```

- ♦ aspectRatio — соотношение ширины к высоте видео. Может быть указано в виде:
  - вещественного числа — будет захвачено видео с соотношением сторон, максимально близком с указанному:

```
const oParams6 = { video: { aspectRatio: 16 / 9 } };
```

  - служебного объекта со свойствами exact, min, ideal и max, описанными ранее, значения которых являются вещественными числами, — будет захвачено видео с соотношением сторон, соответствующим указанному в этом объекте:

```
const oParams7 = { video: { aspectRatio: { ideal: 16 / 9 } } };
```

- ♦ frameRate — частота кадров видео. Может быть задана в виде:
  - вещественного числа в Гц — будет захвачено видео с частотой кадров, максимально приближенной к указанной:

```
const oParams8 = { video: { frameRate: 25 } };
```

  - служебного объекта со свойствами exact, min, ideal и max, описанными ранее, значения которых задаются в виде вещественных чисел в Гц, — будет захвачено видео с частотой кадров, соответствующей указанной в этом объекте:

```
const oParams9 = { video: { frameRate: { min: 12.5, max: 15 } } };
```

- ♦ facingMode — обозначение используемой камеры. Применяется лишь на мобильных устройствах, имеющих несколько камер. Значение может быть указано в виде:
  - строки — по возможности будет использована камера с указанным обозначением (если таковой нет, то первая доступная):
    - 'user' — фронтальная камера (расположенная на стороне экрана);
    - 'environment' — тыловая камера (расположенная на тыльной стороне).

    Следующие два значения применяются на носимых устройствах, оборудованных двумя фронтальными камерами:

    - 'left' — левая камера;
    - 'right' — правая камера.

    Пример указания фронтальной камеры в качестве предпочтительной:

```
const10 = { video: { facingMode: 'user' } };
```

  - служебного объекта со свойством exact, в котором указывается обозначение используемой камеры, — будет задействована камера с указанным обозначе-

нием (если таковой нет, попытка захвата видео не увенчается успехом). Пример указания строго тыловой камеры:

```
const oParams11 = { video: { facingMode: { exact: 'environment' } } };
```

Внимание!

Параметр `aspectRatio` в настоящее время игнорируется Mozilla Firefox.

12.1.1.2. Параметры звука

- `channelCount` — количество каналов звука. Указывается в виде:
  - целого числа — будет выбран источник звука с количеством каналов, максимально близким к указанному:

    ```
    const oParams11 = { audio: { channelCount: 1 } };
    ```

  - служебного объекта со свойствами `exact`, `min`, `ideal` и `max`, описанными в *разд. 12.1.1.1*, значениями которых являются целые числа, — будет выбран источник звука с количеством каналов, соответствующим указанному в этом объекте:

    ```
    const oParams12 = { audio: { channelCount: { max: 2 } } };
    ```

- `sampleRate` — частота дискретизации звука.

  > *Частота дискретизации* — частота, с которой выполняется измерение уровня аналогового сигнала при его оцифровке.

  Может быть задана в виде:
  - целого числа в Гц — будет выбран источник звука с частотой дискретизации, максимально близкой к указанной;
  - служебного объекта со свойствами `exact`, `min`, `ideal` и `max`, описанными в *разд. 12.1.1.1*, значения которых задаются в виде целых чисел в Гц, — будет выбран источник звука с частотой дискретизации, соответствующей указанной в этом объекте;
- `sampleSize` — разрядность дискретизации звука.

  > *Разрядность дискретизации* — количество битов, отводимое под запись уровня аналогового сигнала при его оцифровке.

  Может быть указана в виде:
  - целого числа в битах — будет выбран источник звука с разрядностью дискретизации, максимально близкой к указанной;
  - служебного объекта со свойствами `exact`, `min`, `ideal` и `max`, описанными в *разд. 12.1.1.1*, значения которых задаются в виде целых чисел в битах, — будет выбран источник звука с разрядностью дискретизации, соответствующей указанной в этом объекте;

- `latency` — допустимая задержка звука. Может быть указана в виде:
  - вещественного числа в секундах — будет выбран источник звука с допустимой задержкой, максимально близкой к указанной;
  - служебного объекта со свойствами `exact`, `min`, `ideal` и `max`, описанными в *разд. 12.1.1.1*, значения которых указываются в виде вещественных чисел в секундах, — будет выбран источник звука с допустимой задержкой, соответствующей указанной в этом объекте;
- `autoGainControl` — наличие или отсутствие автоматической регулировки усиления.

  > *Автоматическая регулировка усиления* — поддержание постоянного уровня звукового сигнала, выдаваемого микрофоном, вне зависимости от колебания громкости поступающего на микрофон звука.

  Может быть указан в виде:
  - логической величины — будет выбран источник звука, по возможности, с автоматической регулировкой усиления (если задано значение `true`) или без таковой (если задано `false`);
  - служебного объекта со свойством `exact`, в котором указывается логическая величина, — будет использован источник звука строго с автоматической регулировкой усиления (если задано `true`) или без таковой (если задано `false`);
- `echoCancellation` — наличие или отсутствие подавления эха в виде:
  - логической величины — будет выбран источник звука, по возможности, с подавлением эха (если задано значение `true`) или без такового (если задано `false`);
  - служебного объекта со свойством `exact`, в котором указывается логическая величина, — будет использован источник звука строго с подавлением эха (если задано `true`) или без такового (если задано `false`);
- `noiseSuppression` — наличие или отсутствие подавления шума в виде:
  - логической величины — будет выбран источник звука, по возможности, с подавлением шума (если задано значение `true`) или без такового (если задано `false`);
  - служебного объекта со свойством `exact`, в котором указывается логическая величина, — будет использован источник звука строго с подавлением шума (если задано `true`) или без такового (если задано `false`).

Внимание!

Параметры `channelCount`, `sampleRate` и `sampleSize` в настоящее время игнорируются Mozilla Firefox.

Пример указания параметров видео и звука минимального качества, подходящего для видеотелефонии:

```
const oParams13 = { video: { width:  { min: 240, ideal: 320; max: 640 },
                             height: { min: 160, ideal: 240, max: 480 }},
                    audio: { channelCount: { exact: 1 },
                             sampleRate: 8000, sampleSize: 8,
                             noiseSuppression: true }
            };
```

12.1.2. Получение списка поддерживаемых параметров

Не все параметры, приведенные в *разд. 12.1.1*, поддерживаются всеми камерами и микрофонами. Получить набор параметров, поддерживаемых используемыми в настоящий момент камерой и микрофоном, можно вызовом метода `getSupportedConstraints()` класса `MediaDevices`. Этот метод возвращает служебный объект со свойствами, одноименными с соответствующими им параметрами видео и звука, поддерживаемыми камерой и микрофоном. Все эти свойства хранят значение `true`.

Пример выяснения, поддерживает ли микрофон подавление эха:

```
const oCts = navigator.mediaDevices.getSupportedConstraints();
if (oCts.echoCancellation) {
    // Подавление эха поддерживается
}
```

12.1.3. Смена параметров видео и звука во время захвата

Если необходимо сменить параметры видео и звука (например, переключиться на другую камеру мобильного устройства) в процессе захвата, сначала следует остановить медиапоток, получаемый с камеры, и отключиться от камеры. Для этого нужно выполнить шаги, перечисленные далее.

1. Получить все видео- и аудиодорожки текущего медиапотока — вызвав метод `getTracks()` класса `MediaStream`. Этот метод вернет в качестве результата массив объектов класса `MediaStreamTrack`, представляющих отдельные дорожки видео и звука.
2. Остановить воспроизведение всех дорожек — вызвав метод `stop()` класса `MediaStreamTrack`.
3. Удалить объекты кодировщика и медиапотока, присвоив всем переменным и свойствам других объектам, хранящим ссылки на них, какие-либо другие значения — например, `undefined`.

После этого можно снова попытаться обратиться к камере и получить новый медиапоток, уже с новыми параметрами видео и звука. Пример будет рассмотрен далее.

12.2. Упражнение. Реализуем переключение между фронтальной и тыловой камерами

Доработаем веб-приложение записи видео, написанное при выполнении *упражнений 11.3* и *11.4*, добавив в него поддержку переключения между фронтальной и тыловой камерами. Эта возможность пригодится для пользователей, работающих с мобильными устройствами.

Переключение между камерами реализуем с помощью двух переключателей, которые поместим на страницу приложения. Изначально будет выбираться фронтальная камера. Если текущая платформа не поддерживает переключение между камерами (например, содержит только одну камеру), переключатели будут скрываться.

1. Найдем в папке 11\ex11.4 сопровождающего книгу электронного архива (см. *приложение*) файлы 11.3.html, 11.3.css, 11.3.js и 11.4.php, содержащие код написанного ранее приложения. Скопируем их куда-либо на локальный диск.

   Переключатели, выбирающие камеру, поместим в блок (тег `<div>`), которому дадим якорь `selcam` — нам понадобится получать к нему доступ из программного кода, чтобы при необходимости скрыть. Переключателю **Фронтальная** дадим якорь `frontal`, а переключателю **Тыловая** — `rear`.

   Переключатели разделим тегом `<br>` — чтобы вывести их друг за другом. Блок с переключателями отделим от кнопок «пустым» встроенным контейнером (тегом `<span>`) — чтобы вставить просвет (необходимый стиль для этого контейнера записан в таблице стилей 11.3.css).

2. Откроем в текстовом редакторе копию страницы 11.3.html и вставим в блок, содержащий все кнопки, код, создающий переключатели **Фронтальная** и **Тыловая**:

```
<div>
    <div id="selcam">
        <input type="radio" name="camera" id="frontal" checked>
        Фронтальная<br>
        <input type="radio" name="camera" id="rear">
        Тыловая
    </div>
    <span></span>
    <input type="button" id="start" value="Старт" disabled>
    . . .
</div>
```

3. Откроем в текстовом редакторе копию файла 11.3.js и в начале кода запишем выражения, получающие доступ к только что созданному блоку с переключателями и к самим переключателям:

```
. . .
const aDownload = document.getElementById('download');
const oSelCam = document.getElementById('selcam');
```

```
const rdbFrontal = document.getElementById('frontal');
const rdbRear = document.getElementById('rear');
. . .
```

Теперь самое время выяснить, поддерживает ли текущая платформа переключение между фронтальной и тыловой камерами, и, если не поддерживает, скрыть блок с переключателями.

4. Добавим под вставленными ранее выражениями код, выполняющий эти действия:

```
const isFacingMode = navigator.mediaDevices
                              .getSupportedConstraints().facingMode;

if (!isFacingMode)
    oSelCam.style.display = 'none';
```

Значение свойства `facingMode` служебного объекта, возвращаемого методом `getSupportedConstraints()` класса `MediaDevices`, сохраняем в константе `isFacingMode`. Если значение этой константы — `false` (т. е. платформа не поддерживает переключение между камерами), скрываем блок `selcam`.

При выборе другого переключателя нужно повторно обратиться к камере, получить формируемый ею медиапоток и создать на его основе объект кодировщика. Код, выполняющий эти действия, у нас записан в асинхронном замыкании. Поскольку он будет вызываться в качестве обработчиков события `change` переключателей, это замыкание нужно преобразовать в обычную именованную функцию, также асинхронную, которую назовем `init()`.

5. Переделаем объявление асинхронного замыкания в объявление именованной асинхронной функции `init()`:

```
async function init() {
(async function () {
    . . .
})();
}
```

Если функция `init()` была вызвана при выборе другого переключателя, перед обращением к камере следует остановить медиапоток и удалить объекты кодировщика и медиапотока.

6. Добавим в самое начало тела функции `init()` код, выполняющий эти действия:

```
async function init() {
    oRec = undefined;
    if (oControl.srcObject)
        oControl.srcObject.getTracks().forEach(track => {
            track.stop();
        });
    oControl.srcObject = undefined;
```

```
    const oStream = await navigator
                        .mediaDevices
                        .getUserMedia({ audio: true, video: true });
    . . .
}
```

Объект кодировщика хранится в глобальной переменной `oRec`. Чтобы удалить его, присвоим этой переменной значение `undefined` — и ранее хранившийся в ней объект будет удален подсистемой сборщика мусора.

Объект медиапотока хранится только в свойстве `srcObject` видеопроигрывателя `control`. Останавливаем медиапоток и удаляем его аналогичным образом.

Теперь надо сформировать служебный объект с параметрами захватываемых видео и звука, указав в нем обозначение требуемой камеры, которая идентифицируется установленным переключателем. И выполнить обращение к камере.

7. Вставим в указанное место тела функции `init()` необходимый код:

```
async function init() {
    . . .
    oControl.srcObject = undefined;
    const oParams = { audio: true };
    if (isFacingMode)
        if (rdbFrontal.checked)
            oParams.video = { facingMode: 'user' };
        else
            oParams.video = { facingMode: 'environment' };
    else
        oParams.video = true;
    const oStream = await navigator.mediaDevices
                                   .getUserMedia(oParams);
    . . .
}
```

Сделаем так, чтобы при изменении состояния переключателей **Фронтальная** и **Тыловая** выполнялось повторное обращение к камере.

8. После кода функции `init()` добавим выражения, привязывающие эту функцию к событиям `change` переключателей в качестве обработчика:

```
rdbFrontal.addEventListener('change', init);
rdbRear.addEventListener('change', init);
```

И наконец, сделаем так, чтобы обращение к камере выполнялось сразу после запуска приложения.

9. Добавим в самом конце кода вызов функции `init()`:

```
init();
```

Скопируем файлы 11.3.html, 11.3.css, 11.3.js и 11.4.php в корневую папку веб-сервера и запустим сервер. Откроем веб-обозреватель и перейдем по интернет-

адресу **https://localhost/11.3.html**. Проверим, выводятся ли переключатели **Фронтальная** и **Тыловая** (рис. 12.1), и, если выводятся, попробуем переключиться на тыловую камеру.

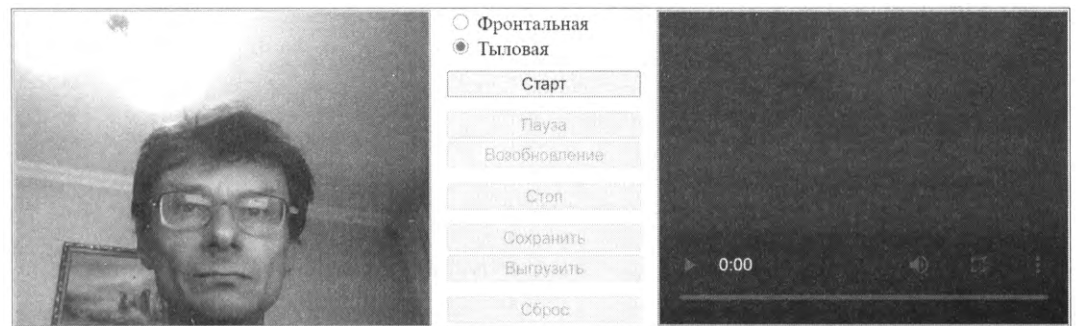


Рис. 12.1. Веб-приложение для записи видео с возможностью переключения между камерами

На традиционных компьютерах, даже оснащенных одной камерой, возможность выбора камеры, скорее всего, будет поддерживаться (по крайней мере, у автора дело обстоит именно так). Просто при установке другого переключателя будет выполняться подключение к той же самой камере.

Попробуем открыть это приложение на каком-либо мобильном устройстве, имеющем две встроенные камеры. Переключимся на тыловую камеру и проверим, действительно ли видео захватывается именно с нее.

12.3. Задание параметров статичных изображений

12.3.1. Параметры, доступные для указания

Параметры статичных изображений, захватываемых с камеры, указываются в вызове метода `takePhoto()` класса `ImageCapture` (см. *разд. 11.7*). Они задаются в виде служебного объекта с одноименными этим параметрам свойствами. Вот параметры, поддерживаемые в настоящее время:

- `imageWidth` — желаемая ширина изображения в виде целого числа в пикселах;
- `imageHeight` — желаемая высота изображения в виде целого числа в пикселах.

  Веб-обозреватель подберет пресет с размерами изображения, наиболее близкими к указанным в этих свойствах;
- `fillLightMode` — обозначение режима работы вспышки (если таковая имеется) в виде строки:
  - `'auto'` — вспышка включится автоматически, если освещенность для съемки недостаточна;
  - `'flash'` — включить вспышку в любом случае;
  - `'off'` — не включать вспышку в любом случае;

- `redEyeReduction` — если `true`, коррекция «красных глаз» будет включена (если она поддерживается), если `false` — будет отключена.

Пример указания размера захватываемых фото 2160×1536 пикселов, постоянно включенной вспышки и коррекции «красных глаз»:

```
const oParams = { imageWidth: 2160, imageHeight: 1536,
                  fillLightMode: 'flash', redEyeReduction: true };
const oBlob = await oIC.takePhoto(oParams);
```

12.3.2. Получение допустимых значений параметров

Есть возможность узнать, какие значения допустимо указывать у тех или иных параметров захватываемых статичных изображений (например, у ширины и высоты).

Выяснить это поможет метод `getPhotoCapabilities()` класса `ImageCapture`. Он вернет промис, который после подтверждения получит в качестве нагрузки объект класса `PhotoCapabilities`, хранящий сведения о допустимых значениях параметров камеры.

Класс `PhotoCapabilities` поддерживает следующие доступные только для чтения свойства:

- `imageWidth` — диапазон значений ширины статичных изображений, допустимых для указания. Представляется объектом класса `MediaSettingsRange`, который поддерживает такие доступные только для чтения свойства:
  - `min` — минимальное значение в виде целого числа в пикселах;
  - `max` — максимальное значение в виде целого числа в пикселах;
  - `step` — минимальная величина, на которую могут отличаться следующие друг за другом допустимые значения, в виде целого числа в пикселах.

    Если свойства `min` и `max` хранят одинаковые величины, значит, для указания доступно лишь одно значение ширины. Тогда величина, хранящаяся в свойстве `step`, не принимается во внимание;
- `imageHeight` — диапазон значений высоты статичных изображений, допустимых для указания. Представляется объектом класса `MediaSettingsRange` (см. ранее);
- `fillLightMode` — обозначение режима работы вспышки в виде одной из строк: `'auto'`, `'flash'` или `'off'` (см. *разд. 12.3.1*);
- `redEyeReduction` — обозначение режима коррекции «красных глаз» в виде одной из строк:
  - `'controllable'` — коррекция поддерживается, и ее режим можно установить в свойстве `redEyeReduction` служебного объекта, передаваемого методу `takePhoto()`;
  - `'always'` — коррекция всегда включена, и отключить ее программно нельзя;

* `'never'` — либо коррекция не поддерживается, либо всегда отключена, и включить ее программно нельзя.

Пример выяснения, можно ли включить коррекцию «красных глаз»:

```
(async function () {
    const oPC = await oIC.getPhotoCapabilities()
    if (oPC.redEyeReduction == 'controllable') {
        // Коррекция «красных глаз» поддерживается и может управляться
        // программно
    }
})();
```

12.4. Получение текущих параметров

Приведенные ранее параметры видео и звука — не более чем рекомендация веб-обозревателю. Захватывая видео и звук, веб-обозреватель принимает во внимание, в первую очередь, возможности камеры и микрофона, а уже во вторую — наши пожелания. Так, если потребовать от камеры выдать видео разрешения 1920×1080 пикселов, а камера поддерживает максимум 640×480, мы вполне предсказуемо получим видео разрешением 640×480 пикселов...

12.4.1. Получение параметров видео и звука

Есть возможность получить текущие параметры захватываемых видео и звука. Для этого нужно выполнить действия, перечисленные далее.

1. Извлечь из медиапотока отдельные видео- и аудиодорожки — вызовом одного из следующих методов:

   * `getTracks()` — возвращает массив всех дорожек, что есть в медиапотоке, — и видео, и звуковых;
   * `getVideoTracks()` — возвращает массив только видеодорожек;
   * `getAudioTracks()` — возвращает массив только аудиодорожек.

   Если выполняется захват видео с обычной камеры и микрофона, медиапоток, скорее всего, будет содержать всего одну дорожку видео и одну дорожку звука.

2. Получить объект класса `MediaStreamTrack`, представляющий нужную дорожку, из извлеченного ранее массива.

   Доступное только для чтения свойство `kind` класса `MediaStreamTrack` хранит строку `'video'`, если текущая дорожка содержит видео, и `'audio'` — если она содержит звук.

3. Получить служебный объект с текущими параметрами видео (если это видеодорожка) или звука (если это дорожка звука) — вызовом метода `getSettings()` класса `MediaStreamTrack`.

Полученный служебный объект содержит следующие свойства:

- если он был получен у видеодорожки:
  - `width` — ширина видео в виде целого числа в пикселах;
  - `height` — высота видео в виде целого числа в пикселах;
  - `aspectRatio` — соотношение ширины к высоте видео в виде вещественного числа;
  - `frameRate` — частота кадров в виде вещественного числа в Гц;
  - `facindMode` — обозначение камеры, с которой захватывается видео, в виде одной из строк: `'user'` (фронтальная), `'environment'` (тыловая), `'left'` (левая фронтальная) или `'right'` (правая фронтальная);
- если он был получен у звуковой дорожки:
  - `sampleSize` — разрядность дискретизации в виде целого числа в битах;
  - `autoGainControl` — `true`, если автоматическая регулировка усиления активна, `false` — в противном случае;
  - `echoCancellation` — `true`, если подавление эха активно, `false` — в противном случае;
  - `noiseSuppression` — `true`, если подавление шума активно, `false` — в противном случае.

> **Внимание!**
>
> Свойства `aspectRatio` и `sampleSize` в настоящее время не поддерживаются Mozilla Firefox.

Пример выяснения, ведется ли съемка тыловой камерой:

```
const oStream = await navigator.mediaDevices
                    .getUserMedia({ audio: true, video: true});
const oVT = oStream.getVideoTracks()[0];
if (oVT.facingMode == 'environment') {
    // Съемка ведется тыловой камерой
}
```

12.4.2. Получение параметров статичных изображений

Текущие параметры захватываемых статичных изображений можно получить вызовом метода `getPhotoSettings()` класса `ImageCapture`. Он вернет промис, который после подтверждения получит в качестве нагрузки служебный объект со следующими свойствами:

- `imageWidth` — текущая ширина изображения в виде целого числа в пикселах;
- `imageHeight` — текущая высота изображения в виде целого числа в пикселах;

- `fillLightMode` — обозначение режима работы вспышки в виде одной из строк: `'auto'` (включается автоматически), `'on'` (включается всегда) или `'off'` (не включается никогда);
- `redEyeReduction` — если `true`, коррекция «красных глаз» выполняется, `false` — если не выполняется.

Пример получения текущих размеров изображений:

```
(async function () {
    const oPS = await oIC.getPhotoSettings()
    const photoWidth = oPS.imageWidth;
    const photoHeight = oPS.imageHeight;
    // Выводим полученные размеры на экран
})();
```

12.5. Самостоятельные упражнения

- Добавьте в приложение для записи звука (результат выполнения самостоятельного *упражнения 11.8*) возможность включить подавление шума. Для включения подавления шума поместите на страницу приложения соответствующий флажок.
- Добавьте в приложение для фотосъемки (результат выполнения упражнения 11.8) возможность выбора качества фото: максимального или минимального. Максимальное качество должно соответствовать максимальному разрешению, поддерживаемому камерой, а минимальное качество — минимальному разрешению. Для указания качества поместите на страницу приложения соответствующий флажок: его установка выберет максимальное качество, сброс — минимальное.

Урок 13
Обработка звука

Источники, обработчики и получатели звука

Звуковой контекст, цепочка и узлы

Панорамирование

Управление усилением звука

Биквадратный фильтр

HTML API предоставляет весьма развитые инструменты для обработки выводимого звука, в частности наложения на него эффектов (панорамирования, фильтрации и др.).

13.1. Как выполняется обработка звука

Для обработки звука применяется ряд объектов, связанных друг с другом в цепочку (рис. 13.1). Эти объекты передают друг другу обрабатываемый звук подобно эстафетной палочке.

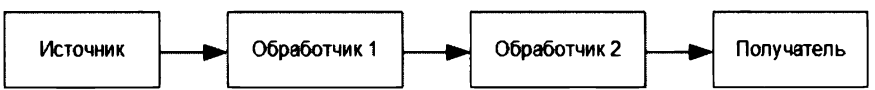


Рис. 13.1. Цепочка взаимосвязанных объектов, применяемых при обработке звука

> *Источник звука* — объект, выдающий изначальный, подлежащий обработке звук.

Источник стоит в самом начале цепочки.

> *Обработчик звука* — объект, принимающий исходный звук от источника (или предыдущего обработчика в цепочке), выполняющий обработку звука согласно заложенному в нем алгоритму и передающий звук далее (получателю или следующему обработчику в цепочке).

Как только что было сказано, в цепочке может быть произвольное количество обработчиков, в совокупности выполняющих сложную обработку звука (на рис. 13.1 представлены два обработчика).

> *Получатель звука* — объект, принимающий звук от последнего обработчика в цепочке и отправляющий его на воспроизведение.

Получатель — последний объект в цепочке.

> *Звуковой контекст* — соединяет отдельные объекты, обрабатывающие звук, в цепочку и организует их совместную работу.

Звуковой контекст — это своего дирижер в программном «оркестре».

> *Узел* — один из объектов, обрабатывающих звук и объединенных в цепочку: звуковой контекст, обработчик или получатель.

13.2. Реализация обработки звука

Для обработки звука следует выполнить действия, перечисленные далее.

1. Создать звуковой контекст.
2. Создать источник звука.
3. Создать обработчик звука и присоединить его к источнику.
4. Если обработчиков должно быть несколько — создать следующий обработчик и присоединить его к предыдущему.
5. Присоединить получатель звука к последнему обработчику.

Рассмотрим эти действия подробнее.

13.2.1. Создание звукового контекста

Звуковой контекст представляется объектом класса `AudioContext`. Конструктор этого класса имеет следующий формат вызова: `AudioContect([<параметры>=undefined])`.

Необязательные *параметры* задаются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Поддерживаются следующие параметры:

- `latencyHint` — желаемая допустимая задержка звука. Чем ниже задержка, тем выше энергопотребление компьютера. Может быть указана в виде:
  - вещественного числа в секундах;
  - одной из строк:
    - `'interactive'` — минимальная задержка, которую может обеспечить звуковая подсистема компьютера. При ее указании возрастает энергопотребление. Используется при программировании игр;
    - `'playback'` — довольно большая задержка, указание которой, тем не менее, позволяет значительно снизить энергопотребление. Используется при воспроизведении музыки;
    - `'balanced'` — средняя задержка, обеспечивающая умеренное энергопотребление.

  Если не указана, используется задержка `'interactive'`;
- `sampleRate` — желаемая частота дискретизации звука в виде целого числа в Гц. Поддерживаются значения от 8000 до 96 000 Гц. Чем выше частота дискретизации, тем выше энергопотребление компьютера.

Если не указана, будет установлена частота дискретизации, заданная в настройках звуковой подсистемы компьютера, обычно — 44 100 Гц.

> **Внимание!**
>
> Параметры `latencyHint` и `sampleRate` в настоящее время игнорируются Mozilla Firefox.

Пример:

```
const oAC = new AudioContext({ latencyHint: 'playback', sampleRate: 22050 });
```

13.2.2. Создание источника звука

Для создания источника звука применяются следующие методы класса `AudioContext`:

- `createMediaElementSource(<проигрыватель>)` — создает и возвращает источник звука на основе заданного объекта аудио- или видео-*проигрывателя* HTML (тега `<audio>` или `<video>`). Возвращаемый источник представляется объектом класса `MediaElementAudioSourceNode`. Пример:

```
<audio src="sound.mp3" id="snd"></audio>
. . .
const oSnd = document.getElementById('snd');
const oAC = new AudioContext();
const oMEAS = oAC.createMediaElementSource(oSnd);
```

> **Внимание!**
>
> Веб-обозреватели не позволяют создавать источники звука на основе аудио- и видеопроигрывателей, которые воспроизводят файлы, загруженные с локального диска. Зачем это сделано, непонятно.

  Класс `MediaElementAudioSourceNode` поддерживает доступное только для чтения свойство `mediaElement`, хранящее ссылку на аудио- или видеопроигрыватель, на основе которого был создан текущий источник;

- `createMediaStreamSource(<медиапоток>)` — создает и возвращает источник звука на основе первой аудиодорожки, которая присутствует в заданном *медиапотоке* (объекте класса `MediaSource`, подробности — в *разд. 11.1*). Возвращаемый источник представляется объектом класса `MediaStreamAudioSourceNode`. Пример:

```
(async function () {
    const oStream = await navigator.mediaDevices
                        .getUserMedia({ audio: true, video: false});
    const oAC = new AudioContext();
    const oMSAS = oAC.createMediaStreamSource(oStream);
})();
```

  Класс `MediaStreamAudioSourceNode` поддерживает доступное только для чтения свойство `mediaStream`, хранящее ссылку на медиапоток, на основе которого был создан текущий источник.

13.2.3. Создание и присоединение обработчиков звука

Отдельные обработчики звука могут быть созданы двумя способами:

- вызовом особых методов класса `AudioContext`, возвращающих объекты этих обработчиков;
- непосредственным созданием объектов этих обработчиков путем вызова конструкторов соответствующих классов.

Какой способ выбрать — дело вкуса.

Методы класса `AudioContext`, создающие обработчики, и классы обработчиков будут рассмотрены далее.

Для присоединения заданного узла к другому узлу применяется метод `connect(<`*присоединяемый узел*`>)`. Этот метод должен вызываться у присоединяющего узла.

Метод `connect()` классы источников и обработчиков наследуют от общего суперкласса `AudioNode`.

13.2.4. Присоединение получателя звука

Для присоединения получателя к последнему обработчику также применяется метод `connect()`, рассмотренный в *разд. 13.2.3*.

В качестве получателя звука можно использовать:

- звуковую подсистему компьютера — при этом звук будет выведен на колонки или наушники.

  Звуковая подсистема представляется объектом класса `AudioDestinationNode`. Единственный объект этого класса, созданный самим веб-обозревателем, хранится в свойстве `destination` класса `AudioContext`.

  Пример:

```
<audio src="sound.mp3" id="audio" controls></audio>
. . .
const oAudio = document.getElementById('audio');
// Создаем звуковой контекст
const oAC = new AudioContext();
// Создаем источник звука на основе аудиопроигрывателя audio
const oMEAS = oAC.createMediaElementSource(oAudio);
// Создаем обработчик, реализующий эффект панорамирования
const oSP = oAC.createStereoPanner();
// Указываем этому обработчику панорамировать звук вправо до максимума
oSP.pan.value = 1;
// Присоединяем обработчик к источнику...
oMEAS.connect(oSP);
// ...а получатель — к обработчику
oSP.connect(oAC.destination);
```

```
// Если запустить воспроизведение в аудиопроигрывателе, звук будет
// воспроизводиться только в правом канале
```

- получатель с «пустым» медиапотоком — для записи обработанного звука с целью преобразования его в файл, сохранения на локальном диске или отправки на сервер.

  Получатель такого рода возвращается методом `createMediaStreamDestination()` класса `AudioContext` и представляется объектом класса `MediaStreamAudioDestinationNode`. Этот класс поддерживает свойство `stream`, хранящее ссылку на автоматически созданный «пустой» медиапоток — объект класса `MediaStream`.

  Пример:

```
const oAudio2 = document.getElementById('audio');
const oAC2 = new AudioContext();
const oMEAS2 = oAC.createMediaElementSource(oAudio);
const oSP2 = oAC.createStereoPanner();
oSP2.pan.value = 1;
oMEAS2.connect(oSP2);
// Создаем получатель с «пустым» медиапотоком для записи в него
// обработанного звука
const oMSAD = oAC.createMediaStreamDestination();
// Присоединяем получатель к обработчику
oSP2.connect(oMSAD);
// На основе «пустого» медиапотока, содержащегося в созданном ранее
// получателе, создаем кодировщик для записи звука
const oRec = new MediaRecorder(oMSAD.stream, 'audio/webm;codecs=opus');
// Далее записываем звук и сохраняем его в файле
```

13.2.5. Решение проблемы с приостановленным звуковым контекстом

В некоторых веб-обозревателях (в частности, в Google Chrome) звуковой контекст, созданный *не* в результате действий пользователя (а, например, при загрузке страницы), автоматически переключается в приостановленное состояние. Такой контекст не «пустит» звук по цепочке, и пользователь ничего не услышит. Вероятно, это сделано с целью не допустить воспроизведения какой-либо назойливой рекламы сразу после загрузки страницы.

Решить эту проблему можно, выполнив перечисленные далее действия.

1. Привязать обработчик к какому-либо событию, возникающему в результате действий пользователя (например, к событию `click` кнопки или событию `play` аудиопроигрывателя).
2. В теле этого обработчика выяснить, хранит ли свойство `state` звукового контекста строку `'suspended'` (т. е. находится ли контекст в приостановленном состоянии).
3. Если это так — запустить контекст, вызвав у него метод `resume()`.

Пример:

```
const oAudio = document.getElementById('audio');
const oAC = new AudioContext();
. . .
oAudio.addEventListener('play', () => {
    if (oAC.state == 'suspended')
        oAC.resume();
});
```

13.3. Обработчики звука

Далее будут описаны наиболее часто применяемые обработчики звука из поддерживаемых веб-обозревателями.

13.3.1. Эффект панорамирования

Этот обработчик позволяет сместить стереобазу звука на заданное значение влево или вправо.

Такой обработчик представляется объектом класса `StereoPannerNode`. Его можно создать двумя способами:

- вызовом метода `createStereoPanner()` класса `AudioContext`:

  ```
  const oSP = oAC.createStereoPanner();
  ```

  Класс `StereoPannerNode` поддерживает свойство `pan`, хранящее величину смещения стереобазы в виде объекта класса `AudioParam`. Последний, в свою очередь, поддерживает свойство `value`, в которое и заносится величина смещения стереобазы в виде вещественного числа от –1 (звук смещен максимально влево и воспроизводится лишь в левом канале) до 1 (звук смещен максимально вправо). Значение 0 помещает стереобазу посередине, это значение по умолчанию. Пример:

  ```
  oSP.pan.value = -0.5;
  ```

- вызовом конструктора класса `StereoPannerNode` — в формате:

  ```
  StereoPannerNode(<звуковой контекст>[, <параметры>=undefined])
  ```

  Звуковой контекст должен быть указан в виде объекта класса `AudioContext`. *Параметры* задаются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Поддерживается параметр `pan`, задающий величину смещения стереобазы в виде вещественного числа от –1 до 1. Пример:

  ```
  const oSP2 = new StereoPannerNode(oAC, { pan: -0.5 });
  ```

> **НА ЗАМЕТКУ**
>
> Класс `AudioParam` также поддерживает ряд методов, позволяющих указать, как величина смещения стереобазы (или иной параметр звука) должна изменяться с течением времени. Подобного рода эффекты реализуются весьма редко, и их создание в этой

книге не описывается. Полное описание класса `AudioParam` можно найти на странице **https://developer.mozilla.org/en-US/docs/Web/API/AudioParam**.

13.3.2. Управление усилением

Этот обработчик позволяет управлять громкостью звука. Он обычно используется для управления уровнем записи звука, захватываемого с микрофона.

Обработчик, управляющий усилением, представляется объектом класса `GainNode`. Его можно создать двумя способами:

- вызовом метода `createGain()` класса `AudioContext`.

  Класс `GainNode` поддерживает свойство `gain`, хранящее уровень усиления в виде объекта класса `AudioParam` (см. *разд. 13.3.1*). В свойство `value` этого класса заносится величина, на которую следует увеличить или уменьшить усиление звука и, соответственно, его громкость, в виде вещественного числа. Отрицательные числа вызывают уменьшение усиления, положительные — увеличение. Число 1 указывает оставить усиление без изменения, это значение по умолчанию. Пример:

  ```
  const oG = oAC.createGain();
  oG.gain.value = 2;
  ```

- вызовом конструктора класса `GainNode` — в формате:

  ```
  GainNode(<звуковой контекст>[, <параметры>=undefined])
  ```

  Звуковой контекст должен быть указан в виде объекта класса `AudioContext`. *Параметры* задаются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Поддерживается параметр `gain`, задающий величину, на которую следует изменить усиление, в виде вещественного числа. Пример:

  ```
  const oG2 = new GainNode(oAC, { gain: 2 });
  ```

13.3.3. Биквадратный фильтр

Этот обработчик позволяет реализовать несколько различных типов фильтров: низких частот, высоких частот, полосно-пропускающий фильтр и др.

Биквадратный фильтр представляется объектом класса `BiquadFilterNode`. Создать его можно двумя способами:

- вызовом метода `createBiquadFilter()` класса `AudioContext`.

  Класс `BiquadFilterNode` поддерживает свойство `type`, в котором указывается строка, обозначающая тип создаваемого фильтра. Еще он поддерживает четыре свойства, служащие для указания параметров фильтра того или иного типа. Значениями этих четырех свойств являются объекты класса `AudioParam` (см. *разд. 13.3.1*). Параметры в свойства `value` этих объектов заносятся в виде вещественных чисел.

Далее будут рассмотрены поддерживаемые значения свойства `type`, соответствующие им типы создаваемых фильтров и назначение остальных четырех свойств класса `BiquadFilterNode`:

- `'lowpass'` — фильтр нижних частот.

  > *Фильтр нижних частот* (ФНЧ) — пропускает частотный спектр звука ниже заданной частоты (*частоты среза*) и подавляющий более высокие частоты.

  Степень подавления высоких частот составляет 12 дБ на октаву.

  Назначение свойств:

  - `frequency` — частота среза (в Гц);
  - `Q` — высота пика на частоте среза (в дБ);

- `'highpass'` — фильтр верхних частот.

  > *Фильтр верхних частот* (ФВЧ) — пропускает частотный спектр звука выше заданной частоты среза и подавляющий более низкие частоты.

  Степень подавления низких частот составляет 12 дБ на октаву.

  Назначение свойств:

  - `frequency` — частота среза в Гц;
  - `Q` — высота пика на частоте среза в дБ;

- `'bandpass'` — полосно-пропускающий фильтр.

  > *Полосно-пропускающий фильтр* (ППФ) — пропускает частоты, находящиеся в указанном диапазоне (*полосе пропускания*), остальные подавляет.

  Назначение свойств:

  - `frequency` — частота, обозначающая середину полосы пропускания, в Гц;
  - `Q` — ширина полосы пропускания. Чем больше указанное в свойстве значение, тем шире полоса;

- `'notch'` — полосно-задерживающий фильтр.

  > *Полосно-задерживающий фильтр* (режекторный фильтр, ПЗФ) — подавляет частоты, находящиеся в полосе пропускания, остальные пропускает.

  Назначение свойств:

  - `frequency` — частота, обозначающая середину полосы пропускания, в Гц;
  - `Q` — ширина полосы пропускания. Чем больше указанное в свойстве значение, тем шире полоса;

- `'lowshelf'` — фильтр, усиливающий или ослабляющий (в зависимости от заданных настроек) частоты, которые располагаются ниже частоты среза.

Назначение свойств:

- frequency — частота среза в Гц;
- gain — степень усиления или ослабления низких частот в дБ. Положительное значение указывает усилить низкие частоты, отрицательное — ослабить;

• 'highshelf' — фильтр, усиливающий или ослабляющий (в зависимости от заданных настроек) частоты, которые располагаются выше частоты среза.

  Назначение свойств:

  - frequency — частота среза в Гц;
  - gain — степень усиления или ослабления высоких частот в дБ. Положительное значение указывает усилить высокие частоты, отрицательное — ослабить;

• 'peaking' — фильтр, усиливающий или ослабляющий (в зависимости от заданных настроек) частоты, которые располагаются в полосе пропускания.

  Назначение свойств:

  - frequency — частота, обозначающая середину полосы пропускания, в Гц;
  - Q — ширина полосы пропускания. Чем больше указанное в свойстве значение, тем шире полоса;
  - gain — степень усиления или ослабления частот в дБ. Положительное значение указывает усилить частоты, отрицательное — ослабить;

• 'allpass' — фазовый фильтр.

  > *Фазовый фильтр* — пропускает все частоты и меняет фазу звукового сигнала. Степень изменения фазы зависит от частоты.

  Назначение свойств:

  - frequency — частота, на которой изменение фазы достигает 90° (*центральная частота*), в Гц;
  - Q — резкость фазового перехода на центральной частоте.

Наконец, свойство detune класса BiquadFilterNode служит для указания отстройки частоты, измеряемой в центах (1 цент — $^1/_{100}$ равномерно темперированного строя).

Значения свойств по умолчанию: type — 'lowpass', frequency — 350, Q — 1, gain — 0, detune — 0.

Пример создания фильтра, подавляющего высокочастотные помехи, которые могут возникнуть при записи звука со встроенного микрофона:

```
const oBF = oAC.createBiquadFilter();
oBF.type = 'highpass';
oBF.frequency.value = 10000;
oBF.Q.value = 2;
```

- вызовом конструктора класса `BiquadFilterNode` — в формате:

  `BiquadFilterNode(<звуковой контекст>[, <параметры>=undefined])`

  Звуковой контекст должен быть указан в виде объекта класса `AudioContext`. *Параметры* задаются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Поддерживаются параметры `type`, `frequency`, `Q`, `gain` и `detune`, соответствующие описанным ранее свойствам. Пример:

```
const oBF2 = new BiquadFilterNode(oAC, { type: 'highpass',
                                          frequency: 10000, Q: 2 });
```

> **НА ЗАМЕТКУ**
>
> Веб-обозреватели поддерживают и другие обработчики звука, реализующие более сложные и менее распространенные на практике фильтры: звуковой компрессор, конвольвер и др. Информацию о них можно найти на странице описания класса `BaseAudioContext`, от которого наследует класс `AudioContext`: **https://developer.mozilla.org/en-US/docs/Web/API/BaseAudioContext**.

13.4. Упражнение. Пишем аудиопроигрыватель с усилением басов

Создадим страницу со стандартным аудиопроигрывателем HTML и добавим ему возможность усиления басов.

Включать и отключать усиление басов будем установкой или сбросом флажка (изначально он будет сброшен), а устанавливать степень усиления — регулятором HTML. Степень усиления будет задаваться в диапазоне от 0 до 10 дБ с шагом 1 (изначально регулятор будет установлен на 0). При сброшенном флажке будем делать регулятор недоступным для пользователя. Частоту, ниже которой начинаются басы, положим равной 1000 Гц.

1. Найдем в папке 13\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы 13.4.html (веб-страница с аудиопроигрывателем) и 13.4.css (таблица стилей). Скопируем их куда-либо на локальный диск.

   Аудиопроигрыватель, присутствующий на странице 13.4.html, имеет якорь `audio` и воспроизводит файл sound.mp3. Сам этот файл в состав электронного архива не входит — вы, уважаемые читатели, можете использовать любой аудиофайл формата MP3, дав ему имя sound.mp3.

2. Откроем в текстовом редакторе копию страницы 13.4.html и вставим под аудиопроигрывателем код, создающий флажок, который включает усиление басов, и регулятор степени усиления:

```
<main>
    <audio id="audio" src="sound.mp3" controls></audio>
    <div>
        <div><input type="checkbox" id="boost_low">
            Усиление басов</div>
```

```
        <input type="range" id="level_low" max="10" value="0"
               disabled>
    </div>
</main>
```

Поскольку флажок изначально сброшен, сразу делаем регулятор недоступным.

3. Добавим в самый конец HTML-кода страницы привязку файла сценария 13.4.js, который скоро создадим и в который запишем весь программный код:

```
<html>
    . . .
</html>
<script src="13.4.js" type="text/javascript"></script>
```

4. Создадим в той же папке, где находятся файлы 13.4.html и 13.4.css, файл 13.4.js и добавим в него выражения, получающие доступ к аудиопроигрывателю, флажку и регулятору:

```
const oAudio = document.getElementById('audio');
const chkBoostLow = document.getElementById('boost_low');
const rbdLevelLow = document.getElementById('level_low');
```

Для усиления басов требуется создать биквадратный фильтр и переключить его в режим фильтра, усиливающего частоты ниже заданной.

5. Добавим код, создающий цепочку из звукового контекста, соответствующим образом настроенного биквадратного фильтра и получателя звука:

```
const oAC = new AudioContext();
const oMEAS = oAC.createMediaElementSource(oAudio);
const oBFLow = new BiquadFilterNode(oAC,
                    { type: 'lowshelf', frequency: 1000 });
oMEAS.connect(oBFLow);
oBFLow.connect(oAC.destination);
oAudio.addEventListener('play', () => {
    if (oAC.state == 'suspended')
        oAC.resume();
});
```

Не забываем о том, что во многих веб-обозревателях звуковой контекст, созданный не в результате действий пользователя, изначально приостанавливается, и его нужно явно запустить.

При установке флажка следует указать биквадратному фильтру степень усиления басов, заданную регулятором, и сделать этот регулятор доступным. А при сбросе флажка нужно указать степень усиления, равной 0 (чтобы убрать усиление), и сделать регулятор, наоборот, недоступным.

6. Добавим код, привязывающий к событию `change` флажка обработчик, который будет выполнять указанные действия:

```
chkBoostLow.addEventListener('change', () => {
    rbdLevelLow.disabled = !chkBoostLow.checked;
```

```
        if (chkBoostLow.checked)
            oBFLow.gain.value = rbdLevelLow.value;
        else
            oBFLow.gain.value = 0;
    });
```

При изменении положения регулятора следует занести заданное в нем значение степени усиления в биквадратный фильтр.

7. Добавим код, привязывающий к событию `change` регулятора обработчик, который выполнит это действие:

```
rbdLevelLow.addEventListener('change', () => {
    oBFLow.gain.value = rbdLevelLow.value;
});
```

8. Найдем какой-либо аудиофайл в формате MP3, скопируем его в корневую папку веб-сервера и дадим копии имя sound.mp3. Скопируем в ту же папку файлы 13.4.html, 13.4.css и 13.4.js. Запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/13.4.html** (рис. 13.2).

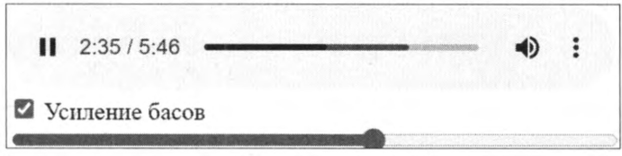


Рис. 13.2. Аудиопроигрыватель с усилением басов

Запустим воспроизведение. Установим флажок **Усиление басов** и увеличим степень усиления посредством регулятора. Отметим, как изменяется при этом звук.

13.5. Дополнительные инструменты для обработки звука

Отсоединить заданный *узел* от другого узла, убрав его тем самым из цепочки, можно вызовом метода `disconnect(<отсоединяемый узел>)` у узла, к которому этот *узел* присоединен. Пример отсоединения обработчика от источника звука:

```
oMEAS.disconnect(oSP);
```

Класс звукового контекста `AudioContext` поддерживает два доступных только для чтения свойства, которые могут оказаться полезными:

- `state` — строковое обозначение состояния, в котором находится текущий контекст:
  - `'running'` — контекст работает;
  - `'suspended'` — контекст находится в приостановленном состоянии;
  - `'closed'` — контекст закрыт (как его закрыть, будет описано чуть позже).

- `currentTime` — время, прошедшее после создания текущего контекста, в виде вещественного числа в секундах.

Также этот класс поддерживает ряд полезных методов:

- `suspend()` — приостанавливает текущий звуковой контекст;
- `resume()` — переводит ранее приостановленный текущий контекст в рабочее состояние;
- `getOutputTimestamp()` — позволяет узнать, в течение какого времени существует текущий звуковой контекст. Метод возвращает служебный объект со следующими свойствами:
  - `contextTime` — время, прошедшее после создания текущего контекста, в виде вещественного числа в секундах (то же, что и свойство `currentTime`);
  - `performanceTime` — время, прошедшее после вывода страницы, которая создала текущий контекст, в виде вещественного числа в миллисекундах;
- `close()` — закрывает текущий контекст, прекращая обработку звука и освобождая все отведенные под нее системные ресурсы. Возвращает промис, подтверждаемый после закрытия контекста и получающий в качестве нагрузки значение `undefined`.

13.6. Самостоятельное упражнение

Добавьте аудиопроигрывателю на странице, написанной при выполнении *упражнения 13.4*, функцию усиления высоких частот, аналогичную функции усиления басов. Частота, выше которой начинаются высокие частоты, пусть будет равна 4000 Гц.

Урок 14
Визуализация звука

Анализатор звука

Спектр

Осциллограмма

Многие программы аудиопроигрывателей при воспроизведении визуализируют звук, показывая различные графики и диаграммы. Современные веб-обозреватели содержат встроенные инструменты для создания подобного рода визуализации.

14.1. Анализатор звука

> *Анализатор звука* — обработчик, предназначенный для получения сведений о воспроизводящемся в текущий момент звуке. В сам звук никаких изменений он не вносит.

Анализатор создается так же, как и остальные обработчики, и точно так же присоединяется к источнику или предыдущему обработчику в цепочке (см. *разд. 13.2.3*).

Для формирования данных, на основе которых будет выводиться график или диаграмма сигнала, анализатор применяет быстрое преобразование Фурье.

14.1.1. Создание анализатора звука

Анализатор представляется объектом класса `AnalyserNode`. Его можно создать двумя способами:

- вызовом метода `createAnalyser()` класса `AudioContext`.

  Класс `AnalyserNode` поддерживает ряд свойств, с помощью которых можно указать параметры анализатора. Эти свойства будут рассмотрены далее.

  Пример:

  ```
  const oAC = new AudioContext();
  const oMEAS = oAC.createMediaElementSource(oAudio);
  const oA = oAC.createAnalyser();
  oMEAS.connect(oA);
  oA.connect(oAC.destination);
  ```

- вызовом конструктора класса `AnalyserNode` — в формате:

  ```
  AnalyserNode(<звуковой контекст>[, <параметры>=undefined])
  ```

`Звуковой контекст` должен быть представлен в виде объекта класса `AudioContext`. Параметры указываются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Эти параметры будут рассмотрены далее. Пример:

```
const oA2 = new AnalyserNode(oAC);
```

14.1.2. Вывод осциллограммы

> *Осциллограмма* — график зависимости амплитуды сигнала от времени.

Для вывода осциллограммы следует выполнить действия, перечисленные далее.

1. Получить размер кадра, выдаваемого анализатором.

   > *Кадр* — массив значений, которые характеризуют воспроизводящийся в текущий момент фрагмент звука, выдаваемый анализатором.

   В случае осциллограммы кадр будет содержать значения амплитуды звукового сигнала в различные моменты времени.

   > *Размер кадра* — количество значений в кадре.

   Получить размер кадра можно, обратившись к доступному только для чтения свойству `frequencyBinCount` класса `AnalyserNode`.

2. Создать массив, в котором будут храниться очередные кадры.

   Такой массив представляется объектом класса `Float32Array`. Он способен хранить только вещественные числа, зато обрабатывается быстрее обычного массива JavaScript, представляемого объектом класса `Array`.

   Конструктор класса `Float32Array` вызывается в формате:

   ```
   Float32Array(<количество элементов в массиве>)
   ```

   В качестве `количества элементов` следует указать размер кадра, полученный на *шаге 1*.

3. Заполнить созданный ранее `массив` очередным кадром — вызвав метод `getFloatTimeDomainData(<массив>)` класса `AnalyserNode`.

   Значения амплитуды, содержащиеся в кадре, представляются вещественными числами от –1.0 (экстремум отрицательных полуволн) до 1.0 (экстремум положительных полуволн), значение 0 обозначает тишину.

4. Вывести данные из полученного кадра на экран в виде графика.

   Извлечь отдельный элемент массива, представленного объектом класса `Float32Array`, можно, применив оператор `[]`.

5. Для вывода осциллограммы следующего звукового фрагмента — перейти на *шаг 3*.

   Для рисования осциллограммы, отображаемой в процессе воспроизведения звука, удобно использовать синхронный вывод на экран (см. *разд. 8.2*).

Метод `getFloatTimeDomainData()` формирует результаты в виде вещественных чисел, вследствие чего позволяет вывести довольно точную осциллограмму. Однако он весьма медленный. Если при выводе осциллограммы не нужна высокая точность, можно заметно повысить быстродействие, затребовав у анализатора значения амплитуды в виде байтов — целых чисел от 0 (экстремум отрицательных полуволн) до 255 (экстремум положительных полуволн), соответственно, целое число 128 представляет 0 — тишину. При этом следует использовать:

- для хранения значений амплитуды — массив, представленный объектом класса `Uint8Array`, хранящий байты. Конструктор этого класса вызывается в том же формате, что и конструктор класса `Float32Array`;
- для получения очередного кадра — метод `getByteTimeDomainData()` класса `AnalyserNode`, имеющий тот же формат вызова, что и метод `getFloatTimeDomainData()`.

Класс `AnalyserNode` поддерживает следующие свойства, позволяющие указать различные параметры формируемой осциллограммы:

- `fftSize` — размер окна в выборках, используемый при выполнении быстрого преобразования Фурье. Задается в виде одного из следующих целых чисел: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 или 32768. Чем больше заданное значение, тем менее детальной получается формируемая осциллограмма. Значение по умолчанию: 2048;
- `minDecibels` — минимальный уровень звука в диапазоне осциллограммы в виде вещественного числа в дБ. Звук заданного в этом свойстве или меньшего уровня будет отображаться как 0 (при выводе в виде вещественных чисел) или 128 (при выводе в виде байтов). Значение по умолчанию: –100;
- `maxDecibels` — максимальный уровень звука в диапазоне осциллограммы в виде вещественного числа в дБ. Звук заданного в этом свойстве или большего уровня будет отображаться как ±1.0 (при выводе в виде вещественных чисел) или 255 (при выводе в виде байтов). Значение по умолчанию: –30;
- `smoothingTimeConstant` — коэффициент сглаживания осциллограммы. Фактически, это среднее между значением очередного элемента текущего массива и значением того же элемента предыдущего массива. Задается в виде вещественного числа от 0,0 (сглаживание не выполняется) до 1,0 (максимально сглаживание). Значение по умолчанию: 0,8.

Если анализатор звука создается вызовом конструктора класса `AnalyserNode`, в передаваемом конструктору служебном объекте можно указать свойства, содержащие значения этих параметров и одноименные им.

14.1.3. Вывод спектра

> *Спектр* — диаграмма распределения интенсивности звукового сигнала по частотам. Представляет собой набор столбцов, каждый из которых показывает интенсивность сигнала в одной из частот.

Для вывода спектра нужно выполнить действия, перечисленные далее.

1. Получить размер кадра.
2. Создать массив, в котором будет храниться очередной кадр.

   Эти действия выполняются так же, как и при выводе осциллограммы (см. *разд. 14.1.2*). Для хранения кадра можно использовать объект класса `Float32Array` или `Uint8Array`.
3. Заполнить ранее созданный кадр данными — в зависимости от типа массива, отведенного под хранение кадра:
   - `Float32Array` — вызовом метода `getFloatFrequencyData()` класса `AnalyserNode`.

     Полученный кадр будет содержать значения интенсивности звука в виде отрицательных вещественных чисел в дБ;
   - `Uint8Array` — вызовом метода `getByteFrequencyData()` того же класса.

     Полученный кадр будет содержать целые числа, показывающие интенсивность звука, от 0 (тишина) до 255 (максимальная интенсивность).
4. Вывести данные из полученного кадра на экран в виде графика.
5. Для вывода спектра следующего звукового фрагмента — перейти к *шагу 3*.

При получении спектра звукового сигнала класс `AnalyserNode` игнорирует значения свойств `minDecibels`, `maxDecibels` и `smoothTimeConstant`, описанных в *разд. 14.1.2*. Значение свойства `fftSize` при увеличении, наоборот, делает спектр более детализированным.

14.2. Упражнение. Пишем аудиопроигрыватель, выводящий осциллограмму

Создадим страницу со стандартным аудиопроигрывателем HTML и реализуем в нем визуализацию звука в виде осциллограммы.

Выводить осциллограмму будем над аудиопроигрывателем черными линиями толщиной 2 пиксела на светло-сером фоне (стандартный CSS-цвет `gainsboro`).

1. Найдем в папке 14\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы 14.2.html (веб-страница с аудиопроигрывателем) и 14.2.css (таблица стилей). Скопируем их куда-либо на локальный диск.

   Аудиопроигрыватель, присутствующий на странице 14.2.html, имеет якорь `audio` и воспроизводит файл sound.mp3. Сам этот файл в состав электронного архива не входит — вы, уважаемые читатели, можете использовать любой аудиофайл формата MP3, дав ему имя sound.mp3.

   Для вывода осциллограммы применим холст HTML, который сначала придется создать в HTML-коде. Дадим этому холсту якорь `graph` и размеры 400×150 пикселов.

2. Откроем в текстовом редакторе копию страницы 14.2.html, найдем в ее коде тег `<audio>`, создающий аудиопроигрыватель `audio`, и вставим над ним код, который создаст холст `graph`:

   ```
   <main>
       <canvas id="graph" width="400" height="150"></canvas>
       <audio id="audio" src="sound.mp3" controls></audio>
   </main>
   ```

3. Поместим в конце HTML-кода привязку файла 14.2.js, который скоро создадим:

   ```
   <html>
       . . .
   </html>
   <script src="14.2.js" type="text/javascript"></script>
   ```

4. Создадим в той же папке, где находятся файлы 14.2.html и 14.2.css, файл 14.2.js, откроем его в текстовом редакторе и запишем в него выражения, получающие доступ к аудиопроигрывателю `audio` и холсту `graph`:

   ```
   const oAudio = document.getElementById('audio');
   const oGraph = document.getElementById('graph');
   ```

5. Добавим код, создающий цепочку из источника звука (в качестве которого выступает аудиопроигрыватель `audio`), анализатора и получателя (которым станет звуковая подсистема компьютера):

   ```
   const oAC = new AudioContext();
   const oMEAS = oAC.createMediaElementSource(oAudio);
   const oA = oAC.createAnalyser();
   oMEAS.connect(oA);
   oA.connect(oAC.destination);
   ```

   Значения амплитуды звука будем получать в виде байтов — это повысит быстродействие.

6. Добавим код, создающий массив под хранение кадра:

   ```
   const arrayLength = oA.frequencyBinCount;
   const oDA = new Uint8Array(arrayLength);
   ```

   Выполним подготовительные действия: получим графический контекст холста `graph`, укажем у него цвет фона `gainsboro`, толщину линий 2 пиксела и черный цвет линий.

   Осциллограмма будет рисоваться отдельными отрезками. Понадобится вычислить длину каждого такого отрезка по оси абсцисс (которая пройдет по середине холста) — она будет равна частному от деления ширины холста на размер кадра.

7. Добавим код, выполняющий указанные подготовительные действия:

   ```
   const oCtx = oGraph.getContext('2d');
   oCtx.fillStyle = 'gainsboro';
   oCtx.lineWidth = 2;
   ```

```
oCtx.strokeStyle = 'black';
const sliceWidth = oGraph.width / arrayLength;
```

Код, рисующий осциллограмму, поместим в функцию `drawGraph()`. Он будет выполнять следующие действия:

- получит очередной кадр с осциллограммой из анализатора;
- создаст переменную `x` — для хранения текущей позиции по оси абсцисс, из которой начнется рисование очередного отрезка осциллограммы (изначально — 0);
- закрасит холст заданным ранее фоновым цветом `gainsboro`;
- запустит рисование сложной фигуры;
- в цикле — переберет все элементы кадра и выполнит на каждой итерации такие действия:
  - вычислит позицию очередного отрезка осциллограммы по оси ординат — разделив значение очередного элемента кадра на 128 и умножив получившееся частное на половину высоты холста. Результат будет присвоен переменной `y`;
  - если обрабатывается первый элемент кадра — поместит графическое перо в точку с координатами [x, y]. Здесь будет располагаться начальная точка рисуемой осциллограммы;
  - если обрабатывается какой-либо из последующих элементов кадра — проведет отрезок от начальной точки осциллограммы или конечной точки предыдущего отрезка;
  - увеличит координату по оси абсцисс (хранится в переменной `x`) на длину, вычисленную на *шаге 7*;
- проведет последний отрезок осциллограммы — в точку с координатами [<*ширина холста*>, 0];
- нарисует сложную фигуру без заливки;
- если звук еще не закончил воспроизводиться — зарегистрирует функцию `drawGraph()` в качестве задачи синхронного вывода, чтобы вывести осциллограмму следующего фрагмента воспроизводимого звука.

8. Добавим объявление описанной ранее функции `drawGraph()`:

```
function drawGraph() {
    oA.getByteTimeDomainData(oDA);
    let x = 0;
    oCtx.fillRect(0, 0, oGraph.width, oGraph.height);
    oCtx.beginPath();
    for (let i = 0; i < arrayLength; i++) {
        const y = oDA[i] / 128 * oGraph.height / 2;
        if (i == 0)
            oCtx.moveTo(x, y);
```

```
        else
            oCtx.lineTo(x, y);
        x += sliceWidth;
    }
    oCtx.lineTo(oGraph.width, oGraph.height / 2);
    oCtx.stroke();
    if (!oAudio.ended)
        requestAnimationFrame(drawGraph);
}
```

Как только звук начнет воспроизводиться, следует перевести звуковой контекст в рабочее состояние и начать рисование осциллограммы, вызвав функцию `drawGraph()`.

9. Добавим обработчик события `play` аудиопроигрывателя, выполняющий эти действия:

```
oAudio.addEventListener('play', () => {
    if (oAC.state == 'suspended')
        oAC.resume();
    drawGraph();
});
```

10. Найдем какой-либо аудиофайл в формате MP3, скопируем его в корневую папку веб-сервера и дадим копии имя sound.mp3. Скопируем в ту же папку файлы 14.2.html, 14.2.css и 14.2.js. Запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/14.2.html**.
11. Запустим воспроизведение и посмотрим на рисуемую осциллограмму (рис. 14.1).

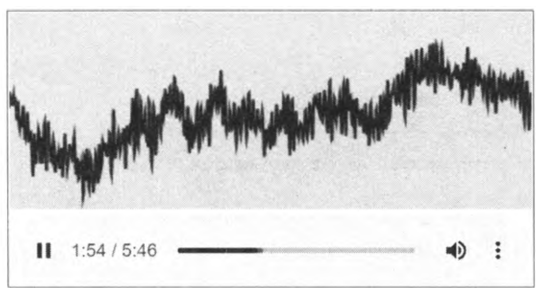


Рис. 14.1. Аудиопроигрыватель, выводящий осциллограмму звука

14.3. Упражнение. Пишем аудиопроигрыватель, выводящий спектр

Напишем другую редакцию аудиопроигрывателя, созданного в *упражнении 14.2*, которая будет выводить спектр.

Выводить спектр будем на черном фоне столбцами красного цвета, причем интенсивность цвета будет тем больше, чем выше столбец.

1. Найдем в папке 14\ex14.2 сопровождающего книгу электронного архива (см. *приложение*) файлы 14.2.html, 14.2.css и 14.2.js. Скопируем их куда-либо на локальный диск и переименуем, соответственно, в 14.3.html, 14.3.css и 14.3.js.

2. Откроем в текстовом редакторе копию файла 14.3.html, найдем в его коде ссылки на файлы 14.2.css и 14.2.js и исправим их на, соответственно, 14.3.css и 14.3.js.

   У свойства `fftSize` анализатора звука значение по умолчанию — 2048, в результате чего спектр получится слишком детализированным — в виде множества тонких линий. Чтобы он стал менее детализированным и выводился в виде относительно широких столбцов, уменьшим значение этого свойства до минимально допустимого — 32.

3. Откроем в текстовом редакторе копию файла 14.3.js и исправим код, создающий объект анализатора звука:

```
. . .
const oMEAS = oAC.createMediaElementSource(oAudio);
const oA = oAC.createAnalyser();
const oA = new AnalyserNode(oAC, { fftSize: 32 });
oMEAS.connect(oA);
. . .
```

   Вычислим ширину каждого столбца, составляющего спектр, для чего разделим ширину холста на размер кадра.

4. Вставим под выражением, получающим доступ к графическому контексту, выражение, вычисляющее ширину столбца, и удалим ненужный код:

```
const oCtx = oGraph.getContext('2d');
oCtx.fillStyle = 'gainsboro';
oCtx.lineWidth = 2;
oCtx.strokeStyle = 'black';
const sliceWidth = oGraph.width / arrayLength;
const barWidth = oGraph.width / arrayLength;
```

   Код, рисующий спектр, поместим в функцию `drawGraph()`. Он будет выполнять следующие действия:

   - получит очередной кадр с данными спектра из анализатора;
   - закрасит холст черным цветом;
   - сбросит на 0 позицию очередного столбца по оси абсцисс;
   - в цикле переберет все элементы кадра и выполнит на каждой итерации такие действия:
     - вычислит высоту очередного столбца спектра — разделив значение очередного элемента кадра на 256 и умножив получившееся частное на половину высоты холста (это сделано, чтобы холст занимал не более половины высоты холста — так он будет выглядеть эффектнее);

- установит цвет очередного столбца — тем более интенсивный, чем выше столбец;
- нарисует столбец. Причем ширина рисуемого столбца будет на 1 пиксел меньше вычисленной на *шаге 4* — так мы создадим просветы между отдельными столбцами;
- увеличит позицию очередного столбца по оси абсцисс на ширину, вычисленную на *шаге 4*;

• если звук еще не закончил воспроизводиться — зарегистрирует функцию `drawGraph()` в качестве задачи синхронного вывода, чтобы вывести спектр следующего фрагмента воспроизводимого звука.

5. Перепишем код функции, чтобы она выполняла указанные действия:

```
function drawGraph() {
    oA.getByteFrequencyData(oDA);
    oCtx.fillStyle = 'black';
    oCtx.fillRect(0, 0, oGraph.width, oGraph.height);
    let barHeight;
    let x = 0;
    for (let i = 0; i < arrayLength; i++) {
        barHeight = oDA[i] / 256 * oGraph.height / 2;
        oCtx.fillStyle = 'rgb(' + (oDA[i] + 100) + ', 50, 50)';
        oCtx.fillRect(x, oGraph.height - barHeight, barWidth - 1,
                                                     barHeight);
        x += barWidth;
    }
    if (!oAudio.ended)
        requestAnimationFrame(drawGraph);
}
```

6. Найдем какой-либо аудиофайл в формате MP3, скопируем его в корневую папку веб-сервера и дадим копии имя sound.mp3. Скопируем в ту же папку файлы 14.3.html, 14.3.css и 14.3.js. Запустим веб-сервер, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/14.3.html**.
7. Запустим воспроизведение и посмотрим на рисуемый спектр (рис. 14.2).

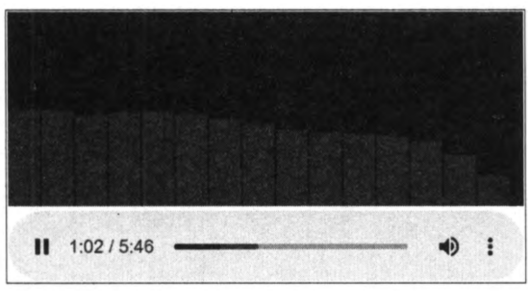


Рис. 14.2. Аудиопроигрыватель, выводящий спектр звука

14.4. Самостоятельные упражнения

- Сделайте осциллограмму, выводимую аудиопроигрывателем из *упражнения 14.2*, «бегущей». Подсказка: поэкспериментируйте с увеличением значения свойства `fftSize` анализатора звука.
- Сделайте так, чтобы спектр, выводимый аудиопроигрывателем из *упражнения 14.3*, рисовался более тонкими столбцами. Подсказка: попробуйте увеличить значение свойства `fftSize` анализатора звука.

ЧАСТЬ V
Работа с сетью

- Протокол WebSocket
- Временные и постоянные соединения
- Технология WebRTC
- Одноранговые сети
- Вещание

Урок 15
WebSocket

Протокол WebSocket

Временные и постоянные соединения

Клиенты и сервер WebSocket

Сигналы

Библиотека Workerman

Протокол HTTP (и его защищенная разновидность HTTPS) для обмена данными между веб-обозревателем и веб-сервером подходит наилучшим образом. Он позволяет пересылать данные в обе стороны (как от веб-сервера веб-обозревателю, так и в обратном направлении) и, поскольку использует временные соединения, весьма устойчив к сетевым сбоям.

> *Временное соединение* — сетевое соединение между двумя приложениями, устанавливаемое только на момент пересылки одного блока данных (например, файла).

Так как временное соединение существует в течение весьма короткого промежутка времени, вероятность того, что в момент его существования произойдет сетевой сбой, очень мала.

Однако для пересылки данных в обоих направлениях (например, при проведении интернет-конференций или чатов) протокол HTTP подходит плохо, и вот почему:

- для пересылки очередного блока данных требуется устанавливать отдельное соединение, что отнимает определенное время и, таким образом, снижает производительность;
- инициировать обмен данными может лишь веб-обозреватель, отправив веб-серверу клиентский запрос. Веб-сервер отправить данные веб-обозревателю по собственной инициативе не может.

Протокол WebSocket лишен этих недостатков.

> *WebSocket* — компактный быстродействующий протокол для двунаправленного обмена произвольными данными между веб-обозревателем и веб-сервером.

Основные отличия WebSocket от HTTP:

- обмен данными может инициировать как клиент, так и сервер;
- для пересылки данных используется постоянное соединение, что значительно увеличивает производительность.

> *Постоянное соединение* — сетевое соединение между двумя приложениями (обычно клиентским и серверным), поддерживаемое до окончания сеанса работы.

Вместо клиентских запросов и серверных ответов в WebSocket приложения обмениваются блоками данных, называемыми *сигналами*. Каждый такой сигнал несет какие-либо данные: строковые (в том числе представленные в формате JSON) или двоичные.

Чтобы использовать в веб-приложении протокол WebSocket, необходимо:

- в состав фронтенда — встроить клиент WebSocket, использовав встроенные инструменты HTML API, поддерживаемые современными веб-обозревателями;
- в состав бэкенда — поместить отдельное приложение сервера WebSocket, которое будет *запускаться и работать отдельно от веб-сервера*.

  Сервер WebSocket может быть написан на языке PHP, Python, JavaScript (выполняемом в среде Node.js), C++, C#, Java и др.

Далее будут рассмотрены сначала реализация клиента WebSocket, а потом — программирование сервера на языке PHP.

> **Полезно знать...**
>
> Протокол WebSocket основан на протоколе TCP.

15.1. Клиент WebSocket

15.1.1. Установление соединения с сервером

Для того чтобы установить соединение с сервером WebSocket, следует создать клиент WebSocket, представляемый объектом класса `WebSocket`. Конструктор этого класса вызывается в следующем формате:

```
WebSocket(<интернет-адрес сервера>)
```

`Интернет-адрес сервера` указывается в виде строки и записывается в том же формате, что и обычный интернет-адрес в WWW. Единственное отличие: в качестве обозначения протокола в нем указывается **ws://** (обычный WebSocket) или **wss://** (его защищенная разновидность).

> **Внимание!**
>
> Веб-страницы, загруженные по протоколу HTTPS, могут устанавливать соединения только с применением защищенной редакции протокола WebSocket. Соединения по обычной, незащищенной, редакции установить не получится. Это сделано ради повышения безопасности.

Операция установления соединения с сервером является асинхронной. Как только соединение будет установлено, в объекте клиента WebSocket возникнет событие `open`. Пример:

```
const oWS = new WebSocket('ws://somesite.ru:2345');
oWS.addEventListener('open', () => {
    // Соединение с сервером WebSocket установлено.
    // Можем начинать обмен данными.
});
```

Класс `WebSocket` поддерживает следующие полезные свойства:

- `binaryType` — тип, используемый для представления пересылаемых двоичных данных (например, файлов), в виде строкового обозначения:
  - `'blob'` — объект класса `Blob` (см. *разд. 11.2*). Подходит для отправки видео и звука, записанных с камеры;
  - `'arraybuffer'` — объект класса `ArrayBuffer`. Подходит для отправки файлов, выбранных пользователем.

  Значение по умолчанию — `'blob'`;
- `readyState` — доступно только для чтения — состояние, в котором находится клиент, в виде целочисленного обозначения:
  - 0 — соединение еще не установлено;
  - 1 — соединение установлено;
  - 2 — соединение закрывается;
  - 3 — соединение закрыто;
- `url` — доступно только для чтения — полный интернет-адрес сервера WebSocket, с которым установлено соединение.

15.1.2. Отправка данных серверу

Для отправки серверу WebSocket сигнала, содержащего указанные *данные*, применяется метод `send(<данные>)` класса `WebSocket`. *Данные* можно указать в виде строки, объекта класса `Blob` или `ArrayBuffer`.

Примеры:

```
// Отправка строки
oWS.send('Привет, сервер!')

// Отправка данных в формате JSON
const oMessage = { type: 'message', content: 'Как дела?' };
const encoded = JSON.stringify(oMessage);
oWS.send(encoded);
```

Отправляемые данные предварительно помещаются в буфер, находящийся в оперативной памяти, и отправляются, когда веб-обозреватель входит в состояние простоя.

Доступное только для чтения свойство `bufferedAmount` класса `WebSocket` хранит объем информации, помещенной в буфер, но еще не отправленной, в виде целого числа в байтах.

15.1.2.1. Отправка файлов серверу

Файлы удобнее всего пересылать в виде data URL. Пример:

```
Выберите файл: <input type="file" id="file">
<input type="button" id="send" value="Отправить ">
. . .
const txtFile = document.getElementById('file');
const btnSend = document.getElementById('send');
btnSend.addEventListener('click', () => {
    const oFR = new FileReader();
    oFR.addEventListener('load', () => {
        const oMessage = { type: 'message',
                           content: 'Получите файл ' + txtFile.files[0].name,
                           file: oFR.result };
        oWS.send(JSON.stringify(oMessage));
    });
    oFR.readAsDataURL(txtFile.files[0]);
});
```

Также файл можно отправить серверу в виде объекта одного из следующих классов:

- `Blob` — в таком виде можно отправить файл с видео или звуком, записанными с камеры и микрофона:

  ```
  const oStream = await navigator
                        .mediaDevices
                        .getUserMedia({ audio: true, video: true });
  oRec = new MediaRecorder(oStream,
                           { mimeType: 'video/webm;codecs=vp9,opus' });
  oRec.addEventListener('dataavailable', (evt) => {
      const oBlob = new Blob([evt.data]);
      // Отправляем записанный файл серверу
      oWS.send(oBlob);
  });
  oRec.start();
  . . .
  oRec.stop();
  ```

- `ArrayBuffer` — в таком виде удобно отправлять файлы, выбранные пользователем с помощью поля выбора файла:

  ```
  <input type="file" id="file">
  <input type="button" id="get" value="Получить">
  . . .
  const txtFile = document.getElementById('file');
  const btnGet = document.getElementById('get');
  btnGet.addEventListener('click', () => {
      const oFR = new FileReader();
  ```

```
    oFR.addEventListener('load', () => {
        // Указываем в качестве типа отправляемых файлов класс
        // ArrayBuffer
        oWS.binaryType = 'arraybuffer';
        // Отправляем выбранный файл
        oWS.send(oFR.result);
    });

    // Загружаем выбранный пользователем файл в виде объекта класса
    // ArrayBuffer, вызвав у считывателя (объекта класса FileReader)
    // метод readAsArrayBuffer(<считываемый файл>)
    oFR.readAsArrayBuffer(txtFile.files[0]);
});
```

15.1.3. Получение данных от сервера

При получении от сервера очередного сигнала в клиенте WebSocket возникает событие `message`. Обработчику этого события передается объект класса `MessageEvent`, хранящий сведения о событии. В их числе присутствуют и данные, полученные в составе сигнала, — их можно извлечь из свойства `data` класса `MessageEvent`.

Пример:

```
// Получение простой строки
oWS.addEventListener('message', (evt) => {
    const message = evt.data;
});

// Получение данных в формате JSON
oWS.addEventListener('message', (evt) => {
    const oMessage = JSON.parse(evt.data);
    const type = oMessage.type;
    const content = oMessage.content;
});
```

15.1.3.1. Получение файлов от сервера

Если файл был отправлен не в виде data URL, а в виде объекта класса `Blob` или `ArrayBuffer`, для его получения можно использовать приведенный далее универсальный код.

```
oWS.addEventListener('message', (evt) => {
    // Извлекаем полученный от сервера файл
    let oFile = evt.data;
    // Если он представлен объектом класса ArrayBuffer, преобразуем его
    // в объект класса Blob
    if (oFile instanceof ArrayBuffer)
        oFile = new Blob(oFile);
```

```
    // Далее делаем что-либо с этим файлом (например, преобразуем в
    // data URL и заносим в гиперссылку)
});
```

15.1.4. Разрыв соединения

Когда обмен данными с сервером завершится, следует явно закрыть установленное ранее соединение. Для этого применяется метод `close()` класса `WebSocket`:

```
close([<код статуса>=1005[, <описание причины>='']])
```

Можно указать целочисленный *код статуса*, описывающий причину, по которой соединение было закрыто. Обычно указывают код 1000, означающий корректное закрытие соединения. Значение параметра по умолчанию — 1005 — ничего не обозначает и является зарезервированным на будущее.

> **На заметку**
>
> Полный список поддерживаемых кодов статуса можно найти на странице **https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent**.

Также можно указать строковое *описание причины*, по которой было закрыто соединение. Оно должно занимать не более 123 байтов. С учетом того, что один символ кириллицы в кодировке Unicode занимает 2 байта, это дает 61 символ.

При закрытии соединения — неважно, вызовом метода `close()` или при возникновении ошибки, — в клиенте WebSocket возникает событие `close`. Его обработчику передается объект класса `CloseEvent`, содержащий сведения о событии. Класс `CloseEvent` поддерживает следующие свойства:

- `code` — целочисленный код статуса, обозначающий причину закрытия соединения;
- `reason` — строковое описание причины закрытия соединения;
- `wasClean` — `true`, если соединение было закрыто корректно, вызовом метода `close()`, и `false` — в противном случае.

15.1.5. Обработка ошибок

При возникновении ошибки в процессе обмена сигналами в клиенте WebSocket возникает событие `error`. В его обработчике можно выполнить соответствующие действия — например, уведомить пользователя о возникшей проблеме.

15.2. Сервер WebSocket

При программировании сервера WebSocket на языке PHP удобно использовать какую-либо готовую библиотеку, реализующую базовую функциональность сервера. Это позволит существенно упростить программирование.

15.2.1. Библиотека Workerman

Одной из таких библиотек является *Workerman*. Она проста в использовании, легко устанавливается и не требует для работы никаких других библиотек.

Для установки Workerman нужно выполнить действия, перечисленные далее.

1. Перейти на «домашнюю» страницу библиотеки: **https://github.com/walkor/Workerman**.
2. Щелкнуть на большой зеленой кнопке **Code** (рис. 15.1).

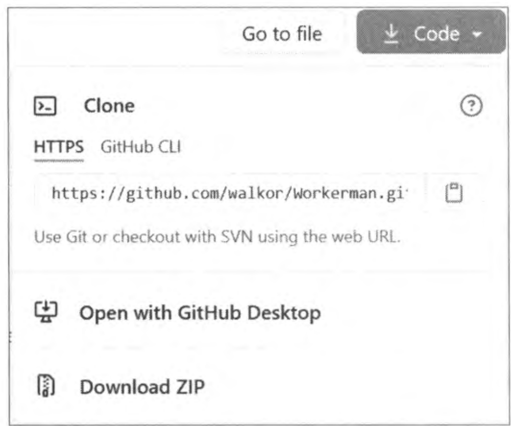


Рис. 15.1. Кнопка **Code**, открывающая всплывающее меню с вариантами установки

3. Щелкнуть на пункте **Download ZIP** появившегося на экране всплывающего меню (это самый нижний пункт).
4. Сохранить загруженный ZIP-архив с библиотекой на локальном диске.
5. Распаковать содержимое этого архива в папку, в которой будет храниться код создаваемого сервера WebSocket.

   Содержимым архива будет папка Workerman-master, хранящая PHP-модули с кодом библиотеки.
6. Для удобства переименовать папку Workerman-master в Workerman.

В самое начало каждого PHP-модуля с кодом сервера WebSocket следует добавить выражение, выполняющее включение модуля Workerman\autoloader.php. Он содержит код обработчика автозагрузки классов, составляющих библиотеку, — без него ничего не заработает. Например, в начале модуля, хранящегося в той же папке, что и папка Workerman, нужно добавить выражение:

```
require_once './Workerman/autoloader.php';
```

15.2.2. Класс слушателя

> *Слушатель* — объект, являющийся частью сервера WebSocket и принимающий сигналы от клиентов.

Слушатель в библиотеке Workerman представляется объектом класса `Worker` из пространства имен `Workerman`. Его нужно создать явно вызовом конструктора класса:

```
Worker(<интернет-адрес для приема сигналов>[, <настройки>=[]])
```

Интернет-адрес, с которого создаваемый слушатель будет принимать сигналы, должен задаваться в виде строки и содержать:

- обозначение протокола — **websocket://** (не **ws://** и не **wss://**, как при создании клиента!);
- адрес хоста — **0.0.0.0**, чтобы принимать сигналы с любого компьютера, или **127.0.0.1** (не **localhost**!), для приема сигналов лишь с локального хоста;
- номер TCP-порта, через который будут приниматься сигналы, — можно выбрать любой незанятый порт.

Настройки указываются лишь в том случае, если сервер должен работать по защищенной редакции протокола WebSocket, и будут рассмотрены позже.

Пример создания слушателя, получающего сигналы с любого компьютера в сети через TCP-порт 2345:

```
$oWorker = new Worker('websocket://0.0.0.0:2345');
```

Как только в объекте слушателя что-либо происходит (устанавливается или разрывается соединение с очередным клиентом, приходит сигнал от клиента, происходит сетевой сбой и др.), в нем возникает соответствующее событие. Обработчики этих событий, задаваемые в виде анонимных функций, присваиваются следующим свойствам класса `Worker`:

- `onConnection` — вызывается при установлении соединения с новым клиентом. В качестве параметра должен принимать объект вновь установленного соединения (класс соединения будет описано далее);
- `onMessage` — вызывается при получении сигнала от клиента. Должен принимать два параметра: объект соединения, с которого были получены данные, и сами данные, полученные в составе клиента. Полученные данные представлены в том виде, в котором они были отправлены клиентом;
- `onClose` — вызывается при закрытии соединения. В качестве параметра должен принимать объект закрытого соединения;
- `onError` — вызывается при возникновении сетевого сбоя. Должен принимать три параметра: объект соединения, код статуса и строковое описание возникшей проблемы;
- `onWorkerStart` — вызывается при запуске сервера. В качестве параметра должен принимать объект слушателя;
- `onWorkerStop` — вызывается при остановке сервера. В качестве параметра должен принимать объект слушателя.

Весь код, реализующий взаимодействие с клиентами, помещается в анонимные функции-обработчики этих событий.

Изначально ни один обработчик не задан, и соответствующие свойства хранят значения `null`.

Класс `Worker` поддерживает следующие полезные свойства:

- `connections` — массив всех установленных сервером соединений (разговор о них пойдет позже);
- `logFile` — статическое — полный путь к файлу журнала, в который будут записываться сведения о работе сервера, в виде строки. Если хранит «пустую» строку, файл журнала с именем workerman.log будет создан в папке, в которой находится папка с библиотекой Workerman. Значение по умолчанию — «пустая» строка;
- `count` — количество процессов, запускаемых слушателем для работы с клиентами, в виде целого числа. Чем больше процессов, тем быстрее будет отклик сервера, но тем выше потребление системных ресурсов серверного компьютера. Значение по умолчанию: 1;
- `name` — наименование сервера в виде строки. Носит информационный характер и может, например, отсылаться клиенту при подключении. Значение по умолчанию — «пустая» строка.

И полезные статические методы:

- `runAll()` — запускает все созданные к настоящему моменту слушатели;
- `stopAll()` — останавливает все созданные к настоящему моменту слушатели.

15.2.2.1. Настройка слушателя для работы по защищенной редакции WebSocket

Все данные, пересылаемые посредством защищенного WebSocket, обязательно шифруются с применением пароля, генерируемого клиентом при установлении соединения. Этот пароль пересылается серверу также в зашифрованном виде, причем для его шифрования и дешифрования применяются разные пароли, называемые *ключами*.

Процесс установления защищенного соединения включает следующие шаги:

1. Клиент — получает от сервера открытый ключ.

   > *Открытый ключ* — пароль, применяемый только для шифрования данных при асимметричном шифровании.
   >
   > *Асимметричное шифрование* — использует один пароль (открытый ключ) для шифрования данных, а другой (закрытый ключ) — для дешифрования.
   > *Закрытый ключ* — пароль, применяемый только для дешифрования данных при асимметричном шифровании.

   Открытый ключ сервер рассылает всем клиентам, которые хотят установить с ним соединение. Закрытый ключ всегда хранится в секрете и не отсылается никому.

И открытый, и закрытый ключи хранятся в особых файлах, которые записываются на серверный компьютер администратором.

2. Клиент — генерирует произвольный ключ сеанса, шифрует его с применением полученного на *шаге 1* открытого ключа и отсылает серверу.

   > *Ключ сеанса* — пароль, которым будут шифроваться сигналы, пересылаемые клиентом и сервером друг другу.

3. Сервер — подтверждает получение ключа сеанса.

Далее клиент и сервер применяют симметричное шифрование данных с использованием ключа сеанса, ранее сгенерированного клиентом.

> *Симметричное шифрование* — использует один пароль (например, ключ сеанса) и для шифрования, и для дешифрования данных.

Чтобы сервер получил поддержку защищенного WebSocket, ему следует сообщить пути к файлам открытого и закрытого ключей и указать некоторые дополнительные параметры. Они задаются во втором параметре конструктора класса `Worker` (см. *разд. 15.2.2*) в виде ассоциативного массива с элементом `ssl`, значением которого также должен быть ассоциативный массив, чьи элементы зададут конкретные параметры:

- `local_cert` — путь к файлу сертификата открытого ключа.

  > *Сертификат открытого ключа* — файл, хранящий сам открытый ключ, сведения о его владельце и, возможно, закрытый ключ.

  Сертификат открытого ключа может как выдаваться особой организацией (*сертификационным центром* — такие сертификаты считаются наиболее надежными), так и генерироваться непосредственно пользователями (*самоподписанный сертификат* — не считается сколь-нибудь надежным).

  Сертификат, выданный сертификационным центром, стоит довольно дорого — несколько тысяч рублей. Поэтому такие сертификаты применяют лишь при эксплуатации сайтов. Во время разработки используют самоподписанные сертификаты, которые (вместе с соответствующими им закрытыми ключами) можно сгенерировать с использованием программного пакета OpenSSL (**https://www.openssl.org/**).

  В составе пакета XAMPP поставляется самоподписанный сертификат и соответствующий ему закрытый ключ, которые можно использовать для отладки серверов WebSocket. Пути к этим файлам будут приведены далее, в примере кода;

- `local_pk` — путь к файлу закрытого ключа (указывается, если закрытый ключ хранится в отдельном файле);

- `allow_self_signed` — значение `false` запрещает использование самоподписанных сертификатов, `true` — разрешает (значение по умолчанию — `false`);

- `verify_peer` — если `false`, будет осуществляться обязательная проверка сертификата, и, если сертификат не пройдет ее, сервер не заработает. Если `true`, сер-

вер будет работать даже в том случае, если сертификат не пройдет проверку. Значение по умолчанию — `true`.

Если используется самоподписанный сертификат (и параметру `allow_self_signed` дано значение `true`), параметру `verify_peer` обязательно следует дать значение `false`.

Кроме параметров защищенного соединения, объекту слушателя следует указать, что для связи используется защищенный протокол. Для этого свойству `transport` слушателя следует присвоить обозначение защищенного протокола — строку `'ssl'`.

Пример кода, создающего сервер WebSocket, который использует сертификат из состава XAMPP:

```
$aContext = ['ssl' => ['local_cert'  =>
                 'c:/xampp/apache/conf/ssl.crt/server.crt',
                       'local_pk' =>
                 'c:/xampp/apache/conf/ssl.key/server.key',
                       'verify_peer' => false,
                       'allow_self_signed' => true] ];
$oWorker = new Worker('websocket://0.0.0.0:2345', $aContext);
$oWorker->transport = 'ssl';
```

> **ПОЛЕЗНО ЗНАТЬ...**
>
> Строка `'ssl'`, присваиваемая свойству `transport` объекта слушателя, обозначает протокол *SSL* (Secure Socket Layer, уровень защищенных сокетов). Этот протокол лежит в основе ряда протоколов повышенной защищенности, включая защищенный WebSocket и HTTPS.

15.2.3. Класс соединения

Соединение с клиентом в библиотеке Workerman представляется объектом класса `TcpConnection`. Объекты этого класса создаются самой библиотекой, добавляются в массив из свойства `connections` слушателя и передаются функциям-обработчикам его событий (см. *разд. 15.2.2*).

Взаимодействие с клиентом выполняется путем вызова следующих методов класса `TcpConnection`:

♦ `send()` — отправляет указанные *данные* клиенту, установившему текущее соединение, в составе сигнала:

  `send(<данные>[, <отправлять данные как есть?>=false])`

  По умолчанию отправляемые *данные* преобразуются в формат, подходящий для пересылки по протоколу WebSocket. Если их нужно отправить без преобразования (что может понадобиться, например, при пересылке файлов), следует дать параметру *отправлять данные как есть* значение `true`.

  Передаваемые *данные* сначала заносятся в особый буфер, формируемый в оперативной памяти и принадлежащий текущему соединению. Если сервер не очень

сильно загружен, данные из буфера отправляются сразу же, в противном случае объект соединения ждет, когда нагрузка на сервер снизится.

Метод возвращает в качестве результата:

- `true` — если данные были успешно отправлены;
- `null` — если данные в текущий момент находятся в буфере и ожидают отправки;
- `false` — если данные не были отправлены, что может случиться при переполнении буфера, закрытии или внезапном разрыве соединения;

- `getRemoteIp()` — возвращает IP-адрес клиента, установившего текущее соединение, в виде строки;
- `close()` — корректно закрывает текущее соединение с отсылкой клиенту кода статуса 1000 (соединение было закрыто корректно). Перед закрытием соединения будут отправлены данные, находящиеся в буфере. Формат вызова:

```
close([<данные>=null[, <отправлять данные как есть?>=false]])
```

  Можно указать *данные*, которые будут оправлены с последним сигналом перед закрытием соединения. Назначение параметра *отправлять данные как есть* аналогично таковому у метода `send()`;
- `destroy()` — немедленно разрывает текущее соединение.

Еще класс `TcpConnection` поддерживает следующие полезные свойства:

- `maxSendBufferSize` — размер буфера, в котором предварительно сохраняются отправляемые данные, в виде целого числа в байтах (по умолчанию — 1048576 байтов, или 1 Мбайт);
- `maxPackageSize` — статическое, предельный размер сигнала, который способно принять соединение, в виде целого числа в байтах (по умолчанию — 10485760 байтов, или 10 Мбайт);
- `id` — уникальный целочисленный идентификатор текущего соединения;
- `worker` — слушатель, создавший текущее соединение, в виде объекта класса `Worker`.

> **На заметку**
>
> Полное описание библиотеки Workerman находится по интернет-адресу **https://github.com/walkor/Workerman**.

15.2.4. Запуск и остановка сервера

Сервер WebSocket — отдельная программа, запускающаяся и работающая независимо от веб-сервера. Запустить ее можно из командной строки, перейдя в папку, в которой хранится PHP-файл с кодом сервера, и набрав команду:

```
<путь к файлу php.exe> <имя файла с кодом сервера>
```

Например, если сервер хранится в папке c:\work, нужно набрать команды:

```
cd c:\work
c:\xampp\php\php.exe server.php
```

> ***Внимание!***
>
> После запуска сервера WebSocket потребуется разрешить ему доступ в сеть, положительно ответив на появившееся на экране предупреждение установленного в системе брандмауэра. Если этого не сделать, ничего не заработает.
>
> Если сервер работает через защищенный протокол WebSocket, и для его запуска использовался самоподписанный сертификат открытого ключа (например, поставляемый в составе пакета XAMPP), при установлении каждого соединения сервер будет выводить в командной строке сообщение об ошибке. Однако на работу сервера это влиять не будет.

Чтобы остановить сервер WebSocket, достаточно переключиться в окно командной строки, в котором он был запущен, и нажать комбинацию клавиш <Ctrl>+<Break> или <Ctrl>+<C>.

15.3. Упражнение. Пишем веб-чат

Напишем службу веб-чата, работающую по протоколу WebSocket и выполняющую следующие функции:

- вход в чат под произвольно заданным именем;
- написание сообщений, содержащих текст и (или) файл;
- отправка сообщений всем пользователям, что присутствуют в чате;
- отправка сообщений пользователю, выбранному в списке (будет реализована читателями самостоятельно);
- выход из чата.

Веб-чат будет состоять из клиента, реализованного в виде веб-приложения, и сервера, написанного на PHP с применением библиотеки Workerman.

15.3.1. Веб-чат: технические детали

Клиент и сервер службы будут располагаться на одном компьютере. Клиент, представляющий собой веб-приложение, будет обслуживаться обычным веб-сервером и взаимодействовать с сервером службы по протоколу WebSocket через TCP-порт 2345.

Клиент будет выводить два экрана:

- экран входа — с полем ввода **Имя** (имя пользователя для входа) и кнопкой **Войти**. После нажатия кнопки **Войти**, при условии, что указанное имя пользователя еще не «занято», будет выполнен переход на экран собственно чата;
- экран собственно чата — с именем пользователя, под которым был выполнен вход в чат, перечнем сообщений (как полученных от других пользователей, так и написанных текущим пользователем), списком **Отправить** (сообщение), в ко-

тором выводятся пункт **Всем** (изначально выбранный) и все пользователи чата (кроме текущего), областью редактирования **Текст сообщения**, полем выбора файла **Файл**, кнопками **Отправить** и **Выход**. При нажатии кнопки **Выход** будет выполняться возврат на экран входа.

Клиент и сервер будут пересылать друг другу сигналы с данными в формате JSON. Каждый сигнал будет содержать обязательное свойство `type`, хранящее строковое обозначение типа этого сигнала (попытка входа в чат, отправка сообщения, оповещение о входе нового пользователя и др.). Набор остальных свойств будет варьироваться в зависимости от выполняемой операции.

Чтобы войти в чат, пользователь, находясь на экране входа, занесет желаемое имя в поле ввода **Имя** и нажмет кнопку **Войти**. При этом:

- клиент:
  - установит соединение с сервером;
  - сразу же отправит ему сигнал со свойствами:
    - `type` — `'login'` (попытка подключения);
    - `userName` — имя нового пользователя;
- сервер — если заданное пользователем имя «свободно»:
  - отправит клиенту, выполняющему вход, сигнал со свойствами:
    - `type` — `'ok'` (вход выполнен успешно);
    - `users` — массив с именами пользователей чата, за исключением только что вошедшего;
  - отправит остальным клиентам сигнал со свойствами:
    - `type` — `'userLoggedIn'` (в чат вошел новый пользователь);
    - `userName` — имя нового пользователя чата;
- клиент — если заданное им имя «свободно»:
  - переключится на экран чата;
- остальные клиенты:
  - добавят нового пользователя в списки **Отправить**;
- сервер — если заданное пользователем имя «занято»:
  - отправит клиенту, пытающемуся выполнить вход, сигнал со свойством:
    - `type` — `'userNameExists'` (в чате уже есть пользователь с заданным именем, и нужно указать другое);
  - разорвет соединение с клиентом.

Чтобы отправить сообщение, пользователь, находясь на экране чата, запишет текст сообщения в область редактирования **Текст сообщения** и (или) выберет файл посредством поля **Файл**, после чего нажмет кнопку **Отправить**.

При этом:

- клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'message'` (пересылается сообщение чата);
    - `userName` — имя пользователя, отправляющего сообщение;
    - `content` — текст сообщения (если он был введен);
    - `fileName` — имя файла, пересылаемого в сообщении (если он был выбран);
    - `file` — содержимое файла, пересылаемого в сообщении, в виде data URL (если файл был выбран);
    - `sent` — временна́я отметка отправки сообщения, представленная в виде строки;
  - выведет отправленное сообщение в перечне сообщений, пометив его как написанное текущим пользователем;
- сервер:
  - перешлет полученное от клиента сообщение без изменения остальным клиентам;
- остальные клиенты:
  - выведут полученное сообщение в перечне сообщений с указанием имени автора.

Чтобы отключиться от чата, пользователь, находясь на экране чата, нажмет кнопку **Выйти**. При этом:

- клиент:
  - разорвет соединение с сервером;
  - переключится на экран входа;
- сервер:
  - отправит всем оставшимся клиентам объект со свойствами:
    - `type` — `'userLoggedOut'` (пользователь вышел из чата);
    - `userName` — имя вышедшего пользователя;
- остальные клиенты:
  - уберут вышедшего пользователя из списков **Отправить**.

15.3.2. Веб-чат: сервер

Файлу, хранящему код сервера веб-чата, дадим имя server.php.

1. Загрузим библиотеку Workerman (как это сделать, было описано в *разд. 15.2.1*), распакуем ее куда-либо и переименуем папку Workerman-master в Workerman.

2. Создадим в той же папке, куда поместили библиотеку, файл server.php и откроем его в текстовом редакторе.

   Сначала нужно создать объект слушателя и функции, которые будут вызываться при получении нового сигнала от клиента и разрыве соединения, пока «пустые».

   Помимо того, нам потребуется хранить на сервере список имен пользователей, вошедших в чат. Это нужно, чтобы при подключении нового пользователя проверить, «свободно» ли указанное им имя. Также желательно связать каждого пользователя с соответствующим ему подключением (это понадобится для реализации личных сообщений, позже, при выполнении самостоятельных упражнений). Такой список удобно представить в виде ассоциативного массива, ключами элементов которого выступают имена пользователей, а значениями элементов — объекты соответствующих им подключений.

3. Запишем в файл server.php код, выполняющий указанные действия:

```
<?php
require_once './workerman/autoloader.php';

use Workerman\Worker;

$aUsers = [];

$oWorker = new Worker('websocket://0.0.0.0:2345');

$oWorker->onMessage = function ($connection, $data) {
};

$oWorker->onClose = function ($connection) {
};

Worker::runAll();
```

   Массив с именами пользователей мы сохранили в переменной `aUsers`.

   Клиент может прислать серверу сигнал, свойство `type` которого хранит значение `'login'` (выполняется попытка входа) или `'message'` (отправляется сообщение). Соответствующие реакции сервера будут реализованы в функции, вызываемой при получении сигнала от клиента (задается в свойстве `onMessage` объекта слушателя).

4. Вставим в функцию из свойства `onMessage` слушателя код, извлекающий значение свойства `type` принятого объекта и пока «пустое» выражение выбора:

```
$oWorker->onMessage = function ($connection, $data) {
    global $aUsers, $oWorker;
    $aData = json_decode($data);
    switch ($aData->type) {
        case 'login':
        case 'message':
    }
};
```

Для декодирования данных, представленных в формате JSON, использовалась встроенная в PHP функция `json_decode()`. Она преобразует строковые JSON-данные, переданные ей в качестве параметра, в объект встроенного класса `stdClass`, аналогичный служебному объекту JavaScript.

Действия, выполняемые сервером при попытке пользователя войти в чат, были описаны в *разд. 15.3.1*. Помимо этого, при успешном входе следует добавить имя нового пользователя в массив из переменной `aUsers`.

5. Добавим в выражение выбора, записанное ранее в тело функции из свойства `onMessage`, код, обрабатывающий вход нового пользователя в чат:

```
. . .
switch ($aData->type) {
    case 'login':
        $userName = $aData->userName;
        if (key_exists($userName, $aUsers)) {
            $aR = ['type' => 'userNameExists'];
            $connection->close(json_encode($aR));
        } else {
            $aR = ['type' => 'ok', 'users' => array_keys($aUsers)];
            $aUsers[$userName] = $connection;
            $connection->send(json_encode($aR));
            $aR = ['type' => 'userLoggedIn', 'userName' => $userName];
            $oR = json_encode($aR);
            foreach ($oWorker->connections as $oCon)
                if ($oCon != $connection)
                    $oCon->send($oR);
        }
        break;
    case 'message':
}
. . .
```

Чтобы получить список имен всех пользователей чата, достаточно извлечь ключи всех элементов массива пользователей (он хранится в переменной `aUsers`) и объединить их в массив. Это выполняет встроенная в PHP функция `array_keys()`.

Функцию `array_keys()` мы вызываем до того, как добавить в массив пользователей элемент, представляющий нового пользователя. Благодаря этому, мы получим массив из имен всех пользователей, кроме вновь присоединившегося к чату.

6. Добавим в выражение выбора, записанное ранее в тело функции из свойства `onMessage`, код, рассылающий сообщения:

```
. . .
switch ($aData->type) {
    case 'login':
        . . .
```

```
        case 'message':
            foreach ($oWorker->connections as $oCon)
                if ($oCon != $connection)
                    $oCon->send($data);
    }
    . . .
```

При обработке разрыва соединения следует иметь в виду, что соединение разрывается в двух случаях: при выходе пользователя из чата и при неудачной попытке пользователя войти в чат, возникающей, если заданное им имя уже «занято». Все действия, сопутствующие выходу пользователя из чата (они были описаны в *разд. 15.3.1* и выполняются в теле функции из свойства `onClose` объекта слушателя), нужно выполнить только в первом случае.

7. Добавим в функцию из свойства `onClose` слушателя код, выполняющий необходимые действия в случае разрыва соединения:

```
$oWorker->onClose = function ($connection) {
    global $aUsers, $oWorker;
    $userName = array_search($connection, $aUsers);
    if ($userName !== FALSE) {
        $aR = ['type' => 'userLoggedOut', 'userName' => $userName];
        $oR = json_encode($aR);
        unset($aUsers[$userName]);
        foreach ($oWorker->connections as $oCon)
            if ($oCon != $connection)
                $oCon->send($oR);
    }
};
```

Сначала ищем в массиве пользователей (из переменной `aUsers`) объект соединения, которое было разорвано (с помощью функции `array_search()` PHP). Если этого соединения в массиве нет, значит, имела место неудачная попытка подключения, и удалять ушедшего пользователя из массива пользователей и рассылать уведомления остальным пользователям не нужно.

На этом разработка сервера чата закончена.

15.3.3. Веб-чат: клиент

Клиент чата будет выводить сообщения в виде блоков (тегов `<div>`). Если выводится сообщение, написанное текущим пользователем, к этому блоку будет привязан стилевой класс `my`. Сообщения будут выводиться в несколько строк:

- имя пользователя-автора сообщения с двоеточием — в виде абзаца со стилевым классом `author`, — только если автором сообщения *не* является текущий пользователь;
- текст сообщения — в виде абзаца, — только если автор ввел этот текст;

- сведения об отправленном файле в формате «Файл: <*имя файла*>[(сохранить)]» — в виде абзаца, — если автор сообщения выбрал файл для отправки.

  Надпись **(сохранить)** станет гиперссылкой, указывающей на полученный файл, и появится на экране, если автором сообщения *не* является текущий пользователь;

- временна́я отметка отправки сообщения — в виде абзаца со стилевым классом `datetime`.

Все необходимые стили уже содержатся в таблице стилей, привязанной к странице, на основе которой будет писаться клиент чата.

1. Найдем в папке 15\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы index.html (веб-страница с интерфейсом чата) и styles.css (таблица стилей). Скопируем их куда-либо на локальный диск.

   Страница index.html содержит следующие элементы:

   - экран входа — тег `<section>` с якорем `login`. В нем находится:
     - веб-форма `frmLogin` — для выполнения входа на сайт. Она содержит поле ввода `txtUserName` (имя пользователя) и кнопку отправки данных `btnLogin` (вход в чат), изначально недоступную;
   - экран чата тег `<section>` с якорем `chat`. В нем находятся:
     - очень важный текст (тег `<strong>`) с якорем `username` — для вывода имени, под которым пользователь подключился к чату;
     - блок `messages` — для вывода перечня полученных сообщений;
     - веб-форма `frmChat` — для отправки сообщений. Содержит область редактирования `txtMessage` (текст сообщения), поле выбора файла `txtFile` (файл, отправляемый в сообщении) и кнопку отправки данных `btnSend` (отправка).

       Еще эта веб-форма содержит список `lstUsers`, содержащий всех присоединившихся к чату пользователей, кроме текущего, и, в самом начале, пункт **Всем**. Пока что этот список служит только для целей информирования, но при выполнении самостоятельных упражнений вы используете его для выбора пользователя, которому будет отправлено личное сообщение;
     - кнопка `btnLogout` — для выхода из чата.

2. Откроем в текстовом редакторе копию страницы index.html и вставим в конце HTML-кода привязку файла сценария script.js, который вскоре создадим:

```
<html>
    . . .
</html>
<script src="script.js" type="text/javascript"></script>
```

3. Создадим файл script.js в той же папке, где находятся файлы index.html и styles.css, откроем его в текстовом редакторе и запишем выражения, получающие доступ к приведенным ранее элементам страницы:

```
const oLogin = document.getElementById('login');
const oChat = document.getElementById('chat');
const frmLogin = document.getElementById('frmLogin');
const txtUserName = document.getElementById('txtUserName');
const btnLogin = document.getElementById('btnLogin');
const oUserName = document.getElementById('username');
const oMessages = document.getElementById('messages');
const frmChat = document.getElementById('frmChat');
const lstUsers = document.getElementById('lstUsers');
const txtMessage = document.getElementById('txtMessage');
const txtFile = document.getElementById('txtFile');
const btnSend = document.getElementById('btnSend');
const btnLogout = document.getElementById('btnLogout');
```

Объявим константу для хранения интернет-адреса сервера службы. Его будем формировать программно, взяв из текущего интернет-адреса адрес хоста, добавив к нему спереди обозначение протокола **ws://**, а сзади — номер TCP-порта 2345.

4. Добавим выражение, объявляющее константу для хранения интернет-адреса сервера службы чата:

```
const serverURL = 'ws://' + location.hostname + ':2345';
```

Также нужно объявить переменные `oWS` и `userName` — для хранения объекта клиента WebSocket и имени текущего пользователя соответственно.

5. Добавим выражение, объявляющее эти переменные:

```
let oWS, userName;
```

В веб-форме `frmLogin` кнопка входа `btnLogin` изначально недоступна, и доступной она должна стать только в том случае, если в поле ввода `txtUserName` что-либо занесено.

6. Добавим обработчик события `input` поля ввода `txtUserName`, который будет делать кнопку `btnLogin` доступной, если это поле не «пусто»:

```
txtUserName.addEventListener('input', () => {
    btnLogin.disabled = txtUserName.value == '';
});
```

После нажатия кнопки `btnLogin` следует создать клиент WebSocket и привязать к нему обработчики событий `open` и `message` (их мы напишем чуть позже).

7. Добавим обработчик события `submit` веб-формы `frmLogin`, выполняющий эти действия:

```
frmLogin.addEventListener('submit', (evt) => {
    evt.preventDefault();
    oWS = new WebSocket(serverURL);
    oWS.addEventListener('open', tryLogin);
    oWS.addEventListener('message', tryLoginAnswer);
});
```

Функция `tryLogin()`, обработчик события `open` клиента WebSocket, сразу же после установления соединения отправит серверу сигнал `login` с именем, введенным пользователем.

8. Добавим объявление функции `tryLogin()`:

```
function tryLogin() {
    const oM = { type: 'login', userName: txtUserName.value };
    oWS.send(JSON.stringify(oM));
}
```

Функция `tryLoginAnswer()`, обработчик события `message` клиента WebSocket, обработает полученный от сервера сигнал-ответ на отправленное «предложение».

Если сервер послал сигнал `ok`, следует скрыть экран `login`, вывести экран `chat`, сохранить введенное в поле `txtUserName` имя пользователя в переменной `userName`, вывести его на страницу в элементе `username`, очистить перечень сообщений `messages`, сбросить веб-форму `frmChat`, сделать кнопку `btnSend` недоступной, удалить из списка `lstUsers` все пункты, кроме первого (**Всем**), и добавить в этот список все имена пользователей, приведенные в массиве из свойства `users` полученного от сервера сигнала. Для добавления в список очередного пользователя применим функцию `addUser()`, которую напишем позже.

А еще нужно убрать у клиента WebSocket ранее привязанные обработчики событий `open` и `messages` (функции `tryLogin()` и `tryLoginAnswer()`) и привязать к событию `message` новый обработчик — еще не написанную функцию `getMessage()`. Эта функция и будет теперь «отвечать» за обработку получаемых от сервера сигналов.

9. Добавим объявление функции `tryLoginAnswer()`:

```
function tryLoginAnswer(evt) {
    const oM = JSON.parse(evt.data);
    if (oM.type == 'userNameExists') {
        txtUserName.focus();
        window.alert('Заданное имя уже занято');
    } else {
        oLogin.style.display = 'none';
        oChat.style.display = 'block';
        userName = txtUserName.value;
        oUserName.textContent = userName;
        oMessages.innerHTML = '';
        frmChat.reset();
        btnSend.disabled = true;
        while (lstUsers.length > 1)
            lstUsers.remove(1);
        for (let userName of oM.users)
            addUser(userName);
        oWS.removeEventListener('open', tryLogin);
        oWS.removeEventListener('message', tryLoginAnswer);
```

```
        oWS.addEventListener('message', getMessage);
    }
}
```

Для удаления пользователей из списка `lstUsers` применим простой прием: будем удалять пункт № 2, пока в списке присутствует больше одного пункта.

Функция `getMessage()`, новый обработчик события `message` клиента WebSocket, должна обрабатывать вывод полученного сообщения в перечень `messages`, добавление вошедшего в чат пользователя в список `lstUsers` и удаление оттуда ушедшего пользователя. Для этого она будет вызывать функции, соответственно, `showMessage()`, `addUser()` и `removeUser()`, которыми мы займемся позже.

10. Добавим объявление функции `getMessage()`:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        case 'message':
            showMessage(oM);
            break;
        case 'userLoggedIn':
            addUser(oM.userName);
            break;
        case 'userLoggedOut':
            removeUser(oM.userName);
    }
}
```

Функция `showMessage()`, выводящая заданное *сообщение* в перечень `messages`, будет иметь следующий формат вызова:

```
showMessage(<сообщение>[, <мое сообщение?>=false])
```

Сообщение указывается в виде служебного объекта. Если параметру *мое сообщение* дать значение `true`, выведенное сообщение будет оформлено как написанное текущим пользователем (как именно — было описано в начале этого раздела).

11. Добавим объявление функции `showMessage()`:

```
function showMessage(message, myMessage=false) {
    const oDiv = document.createElement('div');
    if (myMessage)
        oDiv.classList.add('my');
    let oP, oTxt, oA;
    if (!myMessage) {
        oP = document.createElement('p');
        oP.classList.add('author');
        oP.textContent = message.userName + ':';
        oDiv.appendChild(oP);
    }
```

```
    oP = document.createElement('p');
    if (message.content) {
        oP.textContent = message.content;
        oDiv.appendChild(oP);
    }
    if (message.file && message.fileName) {
        oP = document.createElement('p');
        oTxt = document.createTextNode('Файл: ' +
                                        message.fileName + ' ');
        oP.appendChild(oTxt);
        if (!myMessage) {
            oA = document.createElement('a');
            oA.href = message.file;
            oA.download = message.fileName;
            oA.textContent = '(сохранить)';
            oP.appendChild(oA);
        }
        oDiv.appendChild(oP);
    }
    oP = document.createElement('p');
    oP.classList.add('datetime');
    oP.textContent = message.sent;
    oDiv.appendChild(oP);
    oMessages.appendChild(oDiv);
    oMessages.scrollTop = oMessages.scrollHeight;
}
```

Функция `addUser(<имя>)` добавит заданное *имя* пользователя в список `lstUsers`. Это *имя* будет указано в качестве как текста, так и значения добавляемого пункта (значение пункта списка указывается в атрибуте `value` тега `<option>`) — это упростит дальнейшее программирование.

12. Добавим объявление функции `addUser()`:

```
function addUser(userName) {
    const oOpt = document.createElement('option');
    oOpt.value = userName;
    oOpt.textContent = userName;
    lstUsers.add(oOpt);
}
```

Функция `removeUser(<имя>)` удалит заданное *имя* пользователя из списка `lstUsers`.

13. Добавим объявление функции `removeUser()`:

```
function removeUser(userName) {
    const oOpt = lstUsers.querySelector('option[value=' + userName + ']');
    lstUsers.removeChild(oOpt);
}
```

Поскольку при добавлении пункта в список в атрибут `value` тега `<option>` заносится имя пользователя, для поиска удаляемого пункта мы можем использовать метод `querySelector()`, записав в нем селектор атрибута `value`.

В веб-форме `frmChat` кнопка отправки сообщения `btnSend` изначально недоступна, и доступной она должна стать только в том случае, если в область редактирования `txtMessage` что-либо занесено или если в поле `txtFile` выбран какой-либо файл.

14. Добавим обработчик события `input` области редактирования `txtMessage` и события `change` поля выбора файла `txtFile`, который будет делать кнопку `btnSend` доступной, если занесен текст сообщения или выбран отправляемый файл:

```
function enableSendButton() {
    btnSend.disabled = !txtMessage.value &&
                       (txtFile.files.length == 0);
}

txtMessage.addEventListener('input', enableSendButton);
txtFile.addEventListener('change', enableSendButton);
```

Отправку введенного сообщения реализуем в обработчике события `submit` веб-формы `frmChat`. Однако при занесении в служебный объект, хранящий сообщение, выбранного пользователем файла возникнет проблема. Операция чтения файла посредством объекта класса `FileReader()` является асинхронной, и она вполне может завершиться лишь тогда, когда сообщение уже будет отправлено. В результате остальные пользователи получат сообщение без файла.

Решить эту проблему можно, вынеся код, считывающий файл, в отдельную функцию, которая в качестве результата возвращает промис. Этот промис будет подтвержден по окончании считывания файла и получит в качестве нагрузки содержимое файла в виде data URL. Функцию, читающую файл, назовем `getSelectedFile()`.

15. Добавим объявление функции `getSelectedFile()`:

```
function getSelectedFile() {
    return new Promise((resolve, reject) => {
        const oFR = new FileReader();
        oFR.addEventListener('load', (evt) => {
            resolve(evt.target.result);
        });
        oFR.readAsDataURL(txtFile.files[0]);
    });
}
```

Теперь можно заняться обработчиком события `submit` веб-формы `frmChat`, отправляющим введенное сообщение. Поскольку в нем для получения выбранного пользователем файла будет вызываться только что объявленная функция `getSelectedFile()`, и для получения самого файла будет использован оператор ожидания, реализуем обработчик в виде асинхронной функции.

16. Добавим обработчик события `submit` веб-формы `frmChat`, отправляющий введенное сообщение:

```
frmChat.addEventListener('submit', async function (evt) {
    evt.preventDefault();
    const oM = { type: 'message', userName: userName };
    if (txtMessage.value)
        oM.content = txtMessage.value;
    if (txtFile.files.length > 0) {
        oM.fileName = txtFile.files[0].name;
        oM.file = await getSelectedFile();
    }
    oM.sent = (new Date()).toLocaleString();
    oWS.send(JSON.stringify(oM));
    showMessage(oM, true);
    frmChat.reset();
    btnSend.disabled = true;
    txtMessage.focus();
});
```

После отправки сообщения не забываем вывести это сообщение в перечне `messages` (вызвав объявленную ранее функцию `showMessage()` и указав в ней вторым параметром значение `true`), сбросить веб-форму `frmChat`, сделать кнопку `btnSend` недоступной и установить фокус на область редактирования `txtMessage`. Так мы подготовим клиент для ввода нового сообщения.

После нажатия кнопки `btnLogout` нужно:

- закрыть соединение и удалить объект клиента WebSocket;
- переключиться на экран входа;
- подготовить приложение для выполнения нового входа в чат.

17. Добавим обработчик события `click` кнопки `btnLogout`, который выполнит все эти действия:

```
btnLogout.addEventListener('click', () => {
    oWS.close(1000);
    oWS = undefined;
    oLogin.style.display = 'block';
    oChat.style.display = 'none';
    frmLogin.reset();
    btnLogin.disabled = true;
});
```

При разрыве соединения укажем код статуса 1000 (корректный разрыв соединения). Объект клиента WebSocket удалим, присвоив хранящей его переменной новое значение, — например, `undefined`. И не забудем сбросить веб-форму `frmLogin` и сделать кнопку `btnLogin` недоступной.

18. Откроем командную строку, перейдем в папку, в которой находится сервер чата, и запустим его с помощью команды, описанной в *разд. 15.2.4*.

19. Скопируем файлы index.html, styles.css и script.js в корневую папку веб-сервера и запустим веб-сервер. Откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/**. Мы увидим экран входа нашего веб-чата (рис. 15.2).

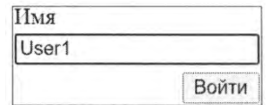


Рис. 15.2. Веб-чат: экран входа

20. Введем какое-либо имя пользователя в поле **Имя** и нажмем кнопку **Войти**. Приложение переключится на экран чата (рис. 15.3).

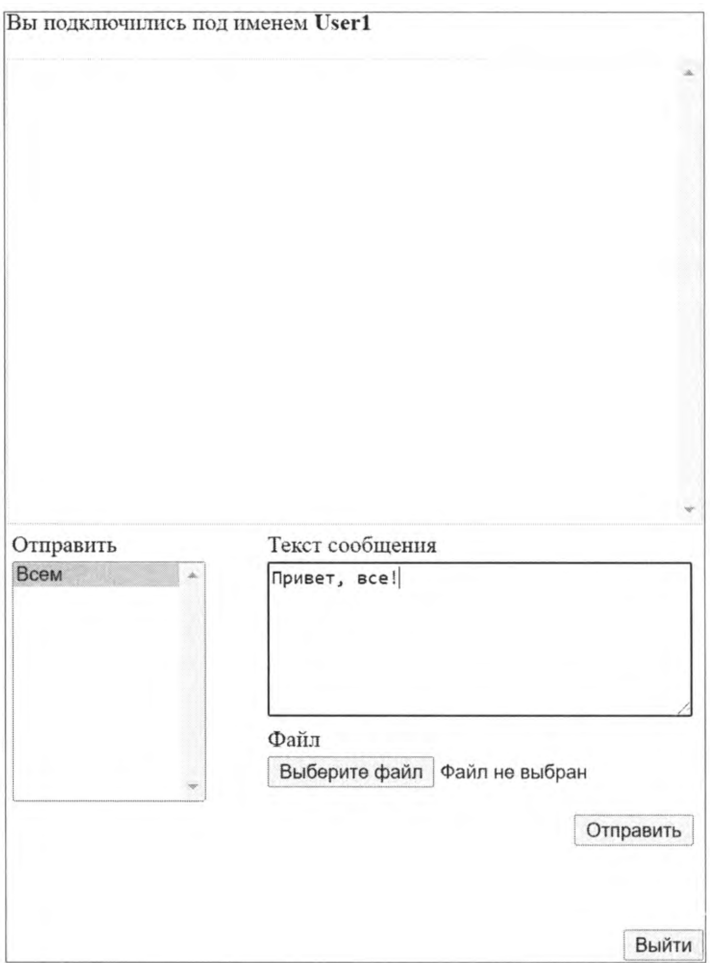


Рис. 15.3. Веб-чат: экран чата

21. Откроем либо другой веб-обозреватель, либо другую вкладку того же веб-обозревателя и выполним вход в чат от имени другого пользователя. Проверим, как работает пересылка сообщений, в том числе включающих файлы (рис. 15.4).

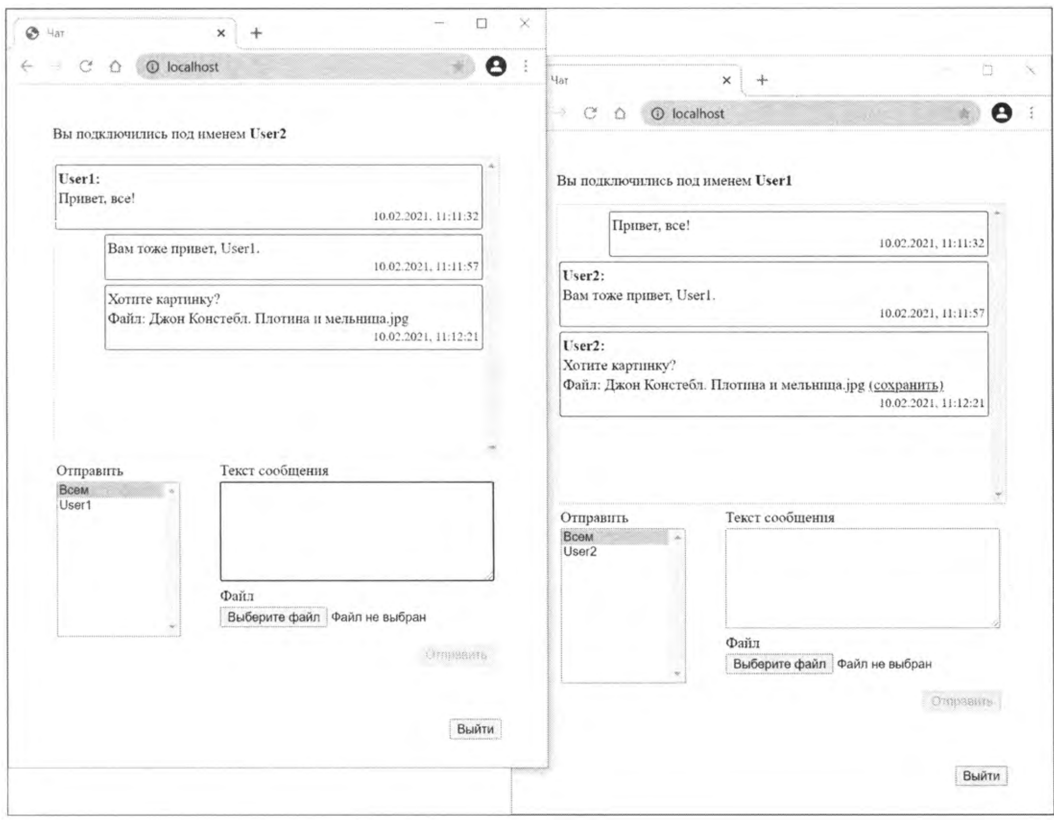


Рис. 15.4. Два экземпляра веб-чата

Напоследок выйдем из обоих экземпляров чата, нажав кнопки **Выйти**, и завершим работу сервера чата (как это сделать, было описано в *разд. 15.2.4*).

15.4. Самостоятельное упражнение

Реализуйте в службе веб-чата (см. *упражнение 15.3*) отправку личных сообщений. Такое сообщение должно отправляться пользователю, выбранному в списке **Отправить** (при выборе пункта **Всем** сообщение должно отправляться всем пользователям).

При отправке личного сообщения клиент занесет имя пользователя-адресата в свойство `toUser` сообщения. Наличие этого свойства укажет серверу, что отправляется именно личное сообщение, и его следует переслать только пользователю с указанным именем.

На стороне сервера объект подключения, соответствующий пользователю-адресату, можно извлечь из массива `aUsers`, использовав полученное имя адресата в качестве ключа.

Урок 16
WebRTC

WebRTC
Двухранговые и одноранговые сети
Вещание
Приглашение и согласие
Претенденты ICE
Каналы данных

Протокол WebSocket имеет существенный недостаток — организуемая им сеть является двухранговой и, соответственно, требует наличия сервера.

> *Двухранговая сеть* — компьютерная сеть, в которой программы-клиенты взаимодействуют друг с другом посредством специальной программы-сервера.

Вследствие этого WebSocket плохо подходит для интенсивного обмена данными (например, интернет-телефонии) — в этом случае значительно возрастает нагрузка на сервер, который в определенный момент просто не сможет обработать все сигналы от клиентов и «повиснет».

Устранить этот недостаток можно, дав клиентам возможность пересылать данные друг другу напрямую, без участия сервера, с применением WebRTC.

> *WebRTC* (Web Real-Time Communication, соединение реального времени через веб) — семейство протоколов и технологий, предназначенных для организации одноранговой сети.
>
> *Одноранговая сеть* — компьютерная сеть, в которой программы-клиенты взаимодействуют друг с другом напрямую, без участия программы-сервера

Современные веб-обозреватели предоставляют удобные программные инструменты для обмена данными на основе WebRTC. С их помощью можно пересылать видео, звук и произвольные данные (например, сообщения чата).

Внимание!

При установке соединения посредством WebRTC клиентам потребуется пересылать друг другу всевозможные служебные данные. Их пересылка выполняется по протоколу WebSocket, соответственно, для установления соединения все-таки требуется сервер. Однако, как только соединение WebRTC установлено, обмен данными идет непосредственно между клиентами, без участия сервера.

16.1. Организация вещания посредством WebRTC

> *Вещание* — пересылка по сети видео или звука в реальном времени.

В частности, вещание используется для передачи видео и звука при интернет-телефонии.

Организация вещания средствами WebRTC выполняется в несколько шагов. Далее перечислены шаги, выполняемые вызывающим клиентом, который собирается соединиться с другим клиентом — вызываемым.

1. Создание соединения WebRTC.
2. Указание в соединении WebRTC медиапотока, который будет отправляться вызываемому клиенту.
3. Создание приглашения, регистрация его в соединении WebRTC и отправка вызываемому клиенту.

   > *Приглашение* — объект, сообщающий вызываемому клиенту о желании вызывающего связаться с ним посредством WebRTC и описывающий характеристики вызывающего клиента (в частности, интернет-адрес).

   Отправка приглашения выполняется по протоколу WebSocket через сервер в составе отдельного сигнала.
4. Получение от вызываемого согласия и регистрация его в соединении WebRTC.

   > *Согласие* — объект, сообщающий вызывающему клиенту о готовности вызываемого клиента связаться с ним и описывающий характеристики вызываемого клиента.

   Согласие также пересылается по протоколу WebSocket через сервер в составе отдельного сигнала.
5. Обмен претендентами по технологии ICE.

   > *Претендент* (candidate) — объект, отсылаемый одним клиентом другому клиенту и содержащий предлагаемые параметры устанавливаемого соединения.
   >
   > *ICE* (Interactive Connectivity Establishment, установление интерактивного соединения) — технология, описывающая формат пересылаемых клиентами друг другу претендентов.

   Претенденты пересылается по протоколу WebSocket через сервер, каждый — отдельным сигналом.

   Как только будет получен претендент, содержащий приемлемые параметры соединения, клиент сам установит соединение и начнет вещание.
6. Присоединение к вещанию, осуществляемому вызываемым клиентом, и вывод получаемого медиапотока на экран.
7. Закрытие соединения WebRTC.

Далее описаны шаги, которые должен предпринять вызываемый клиент.

1. Получение приглашения от вызывающего клиента.
2. Создание соединения WebRTC и регистрация в нем полученного приглашения.
3. Указание в соединении WebRTC медиапотока, который будет отправляться вызывающему клиенту.
4. Создание согласия, регистрация его в соединении WebRTC и отправка вызывающему клиенту.
5. Обмен претендентами.
6. Присоединение к вещанию, осуществляемому вызывающим клиентом, и вывод получаемого медиапотока на экран.
7. Закрытие соединения WebRTC.

16.1.1. Создание соединения WebRTC

Соединение WebRTC должны создать оба клиента: вызывающий — в ответ на действие пользователя, желающего начать сеанс связи, вызываемый — после получения от вызывающего приглашения и согласия пользователя «поднять трубку».

Соединение WebRTC представляется объектом класса `RTCPeerConnection`. Конструктор этого класса вызывается в формате: `RTCPeerConnection([`<*параметры*>`])`.

Параметры представляются в виде служебного объекта со свойствами, одноименными с соответствующими параметрами. Поддерживается параметр `iceServers`, хранящий массив интернет-адресов серверов ICE.

> *Сервер ICE* — используется для установления соединения WebRTC в том случае, если клиенты находятся в разных локальных сетях, разделенных сетевыми экранами или брандмауэрами.

Каждый элемент задаваемого массива должен быть служебным объектом со следующими свойствами:

- `url` — должно содержать одно из двух:
  - интернет-адрес в виде строки — если сервер ICE имеет всего один интернет-адрес;
  - массив интернет-адресов, заданных в виде строк, — если сервер ICE имеет несколько интернет-адресов;

  Это единственное обязательное для указания свойство;
- `username` — регистрационное имя, необходимое для подключения к серверу;
- `credential` — пароль для подключения к серверу.

  Эти два свойства указываются, если для доступа к серверу ICE нужны имя и пароль.

Если параметр `iceServers` не указан, можно будет установить соединение только между клиентами, находящимися в одной локальной сети.

> ***На заметку***
>
> Списки бесплатных серверов ICE можно найти по интернет-адресу **https://gist.github.com/yetithefoot/7592580/**, а также выполнив поиск в Интернете по ключевым словам: `ice server list`.

Пример:

```
// Для обмена служебными данными понадобится клиент WebSocket
const oWS = new WebSocket('ws://somesite.ru:2345');
const oPs = { iceServers: [ { url: 'stun:stun.l.google.com:19302' },
                            { url: 'stun:stun1.l.google.com:19302' },
                            { url: 'stun:stun2.l.google.com:19302' },
                            { url: 'stun:stun3.l.google.com:19302' },
                            { url: 'stun:stun4.l.google.com:19302' } ] };
const oRTC = new RTCPeerConnection(oPs);
```

16.1.2. Указание вещаемого медиапотока

После создания соединения WebRTC следует указать в нем вещаемый медиапоток. Для этого нужно выполнить действия, перечисленные далее.

1. Получить сам медиапоток (с камеры или экрана, подробности — в *разд. 11.1* и *11.6*).
2. Извлечь из полученного медиапотока массив всех видео- и звуковых дорожек — вызвав метод `getTracks()` медиапотока (см. *разд. 12.4.1*).
3. Занести каждую из этих дорожек в соединение WebRTC — вызвав метод `addTrack()` класса `RTCPeerConnection`:

   ```
   addTrack(<дорожка>, <медиапоток>)
   ```

   Дорожка должна быть представлена объектом класса `MediaStreamTrack`, а медиапоток — объектом класса `MediaStream`.

Пример:

```
let oLocalStream;
(async function () {
    oLocalStream = await navigator
                         .mediaDevices
                         .getUserMedia({ video: true, audio: true });
    for (let oTr of oLocalStream.getTracks())
        oRTC.addTrack(oTr, oLocalStream);
})();
```

16.1.3. Обмен приглашением и согласием

16.1.3.1. Вызывающий: создание приглашения

Приглашение должно создаваться вызывающим клиентом. Обычно это выполняется в результате действий пользователя (например, нажатия кнопки вызова).

Для создания приглашения следует выполнить действия, перечисленные далее.

1. Создать приглашение — вызовом метода `createOffer()` объекта соединения.

   Метод возвращает промис, который подтверждается после создания приглашения и в качестве нагрузки получает хранящий его служебный объект.

2. Зарегистрировать приглашение в соединении WebRTC — вызовом метода `setLocalDescription()` объекта соединения:

   `setLocalDescription(<служебный объект с приглашением>)`

   Здесь следует указать *объект*, полученный в результате вызова метода `createOffer()`.

   Метод возвращает промис, который подтверждается после регистрации приглашения и получает в качестве нагрузки значение `undefined`.

3. Извлечь зарегистрированное приглашение — из свойства `localDescription` объекта соединения WebRTC.

   Приглашение представляется объектом класса `RTCSessionDescription`.

4. Отправить приглашение вызываемому — отдельным сигналом WebSocket.

Пример:

```
btnCall.addEventListener('click', async function () {
    const oOffer = await oRTC.createOffer();
    await oRTC.setLocalDescription(oOffer);
    // Отправляем приглашение вызываемому
    const oS = { type: 'offer', data: oRTC.localDescription };
    oWS.send(JSON.stringify(oS));
}
```

16.1.3.2. Вызываемый: получение приглашения и создание согласия

Согласие создается вызываемым клиентом после получения от вызывающего по протоколу WebSocket сигнала с приглашением, а от пользователя — согласия начать вещание (что может, например, выполняться нажатием кнопки **Принять вызов**).

Вызываемому клиенту следует выполнить действия, перечисленные далее.

1. Создать объект соединения WebRTC (см. *разд. 16.1.1*).

2. Зарегистрировать полученное приглашение в соединении WebRTC — вызвав метод `setRemoteDescription(<объект приглашения>)` класса у объекта соединения. В вызове следует указать *объект*, полученный от вызывающего абонента.

Метод вернет промис, который будет подтвержден после регистрации приглашения и получит в качестве нагрузки значение `undefined`.

3. Указать вещаемый медиапоток в созданном ранее объекте соединения WebRTC (как это сделать, рассказывалось в *разд. 16.1.2*).
4. Создать согласие — вызовом метода `createAnswer()` у объекта соединения.

   Метод возвращает промис, который подтверждается после создания согласия и в качестве нагрузки получает хранящий его служебный объект.
5. Зарегистрировать согласие в соединении WebRTC — вызовом метода `setLocalDescription()` (см. *разд. 16.1.3.1*) объекта соединения с передачей этому методу полученного ранее служебного объекта с согласием.
6. Извлечь зарегистрированное согласие — из свойства `localDescription` объекта соединения WebRTC.

   Согласие также представляется объектом класса `RTCSessionDescription`.
7. Отправить согласие вызывающему — отдельным сигналом WebSocket.

Пример:

```
// Предполагается, что объект соединения WebRTC уже создан и хранится
// в переменной oRTC
let oLocalStream;
oWS.addEventListener('message', async function (evt) {
    const oDS = JSON.parse(evt.data);
    switch (oDS.type) {
        case 'offer':
            await oRTC.setRemoteDescription(oDS.data);
            oLocalStream = await navigator
                              .mediaDevices
                              .getUserMedia({ video: true, audio: true });
            for (let oTr of oLocalStream.getTracks())
                oRTC.addTrack(oTr, oLocalStream);
            const oAnswer = await oRTC.createAnswer();
            await oRTC.setLocalDescription(oAnswer);
            // Отправляем согласие вызывающему
            const oS = { type: 'answer', data: oRTC.localDescription };
            oWS.send(JSON.stringify(oS));
            break;
        . . .
    }
});
```

16.1.3.3. Вызывающий: получение согласия

Получив согласие, вызывающий клиент должен зарегистрировать его в соединении WebRTC, вызвав метод `setRemoteDescription()` (см. *разд. 16.1.3.2*) объекта соединения. Этому методу нужно передать объект согласия, полученный от вызываемого.

Пример:

```
oWS.addEventListener('message', async function (evt) {
    const oDS = JSON.parse(evt.data);
    switch (oDS.type) {
        . . .
        case 'answer':
            await oRTC.setRemoteDescription(oDS.data);
            break;
        . . .
    }
});
```

16.1.4. Обмен претендентами

Сразу после обмена приглашением и согласием клиенты начинают «переговоры» о приемлемых характеристиках соединения. В процессе этого каждый из них посылает другому клиенту объекты-претенденты с описанием предлагаемых характеристик.

16.1.4.1. Отправка претендента

Претенденты, предназначенные к отправке другому клиенту, создаются объектом соединения WebRTC автоматически. Получить очередной претендент можно в обработчике события `icecandidate`, возникающего в соединении после создания этого претендента. Обработчику события передается объект класса `RTCPeerConnectionIceEvent`, содержащий сведения о событии. Из свойства `candidate` этого объекта можно извлечь сам претендент, представленный объектом класса `RTCIceCandidate`.

Как только клиенты «договорятся» о взаимно подходящих параметрах сетевого соединения, событие `icecandidate` возникнет в последний раз, в этом случае свойство `candidate` объекта со сведениями о событии будет хранить `null`.

Каждый претендент отсылается другому клиенту отдельным сигналом WebSocket.

Пример:

```
function sendCandidate(evt) {
    // Если есть претендент, подлежащий отправке...
    if (evt.candidate) {
        // ...отправляем его
        const oS = { type: 'candidate', data: evt.candidate };
        oWS.send(JSON.stringify(oS));
    }
}

oRTC.addEventListener('icecandidate', sendCandidate);
```

16.1.4.2. Получение и регистрация претендента

Для регистрации полученного *претендента* в соединении WebRTC применяется метод `addIceCandudate(<претендент>)` класса `RTCPeerConnection`. Метод возвращает промис, подтверждаемый после регистрации претендента и получающий в качестве нагрузки значение `undefined`.

Пример:

```
oWS.addEventListener('message', async function (evt) {
    const oDS = JSON.parse(evt.data);
    switch (oDS.type) {
        . . .
        case 'candidate':
            await oRTC.addIceCandidate(oDS.data);
            break;
        . . .
    }
});
```

16.1.5. Присоединение к вещанию

Как только «договоренность» о соединении будет достигнута, каждый из клиентов автоматически начнет вещание медиапотока, ранее зарегистрированного в объекте соединения (см. *разд. 16.1.2*). Однако присоединение к вещанию, выполняемому клиентом-собеседником, автоматически не выполняется, и его следует реализовать самостоятельно.

Присоединение к вещанию реализуется в обработчике события `track`, возникающего, когда клиент-собеседник добавляет в вещаемый медиапоток очередную дорожку видео или звука. В качестве параметра обработчику события передается объект класса `RTCTrackEvent`, хранящий сведения о событии. Свойство `streams` этого объекта хранит массив медиапотоков, в которые была добавлена дорожка, представленных объектами класса `MediaStream`. Нас будет интересовать первый медиапоток из этого массива.

Пример:

```
let oRemoteStream;
function join(evt) {
    oRemoteStream = evt.streams[0];
    // Выводим получаемый от собеседника медиапоток в видеопроигрывателе
    oReceivedVideo.srcObject = oRemoteStream;
}

oRTC.addEventListener('track', join);
```

16.1.6. Завершение вещания

Чтобы завершить вещание, клиенту следует выполнить шаги, перечисленные далее.

1. Убрать все обработчики, ранее привязанные к событиям объекта соединения WebRTC.

   Это нужно для того, чтобы они не сработали в процессе завершения вещания и не вызвали неполадки в работе приложения.

2. Остановить все видео- и звуковые дорожки в обоих медиапотоках — и собственном, и получаемым от собеседника.

   Остановить дорожки можно, получив массив дорожек вызовом метода `getTracks()` медиапотока и вызвав у каждой дорожки метод `stop()`.

3. Удалить объекты медиапотоков — присвоив хранящим их переменным другие значения, обычно `undefined`.
4. Закрыть соединение — вызовом метода `close()` у объекта соединения WebRTC.
5. Удалить сам объект соединения — присвоив хранящей его переменной любое другое значение, например `undefined`.
6. Отправить другому клиенту требование также завершить вещание — в составе сигнала WebSocket.

Пример:

```
// Функция, останавливающая вещание
function stop() {
    for (let oTr of oRemoteStream.getTracks())
        oTr.stop();
    for (let oTr of oLocalStream.getTracks())
        oTr.stop();
    oReceivedVideo.srcObject = undefined;
    oRemoteStream = undefined;
    oLocalStream = undefined;
    oRTC.removeEventListener('icecandidate', sendCandidate);
    oRTC.removeEventListener('track', join);
    oRTC.close();
    oRTC = undefined;
}

// При получении от другого клиента уведомления об остановке вещания
// с его стороны нам также следует остановить вещание
oWS.addEventListener('message', async function (evt) {
    const oDS = JSON.parse(evt.data);
    switch (oDS.type) {
        . . .
        case 'close':
            stop();
```

```
            break;
        . . .
    }
});

// Нажатие кнопки btnCancel вызовет остановку вещания
btnCancel.addEventListener('click', () => {
    // Останавливаем вещание
    stop();
    // Отправляем другому клиенту уведомление об остановке вещания
    const oM = { type: 'close', toUser: 'User2' };
    oWS.send(JSON.stringify(oM));
});
```

16.2. Упражнение. Пишем службу видеотелефона

Напишем интернет-службу видеотелефонии с применением технологии WebRTC, выполняющую следующие функции:

- вход в службу под произвольно заданным именем;
- выбор абонента для проведения связи;
- пометка абонентов, уже участвующих в сеансе связи, как занятых (чтобы им никто более не смог «позвонить», пока они не завершат сеанс);
- отправка текстовых сообщений абоненту (будет реализована читателями самостоятельно);
- выход из службы.

Видеотелефон будет состоять из клиента и сервера, предназначенного для пересылки служебных сигналов и написанного на PHP с применением библиотеки Workerman.

16.2.1. Видеотелефон: технические детали

Клиент и сервер видеотелефона будут располагаться на одном компьютере. Клиент, представляющий собой веб-приложение, будет обслуживаться обычным веб-сервером и взаимодействовать с сервером через TCP-порт 2345.

Клиент будет выводить пять экранов:

- экран входа — с полем ввода **Имя** и кнопкой **Войти**. После нажатия кнопки **Войти**, при условии, что указанное имя абонента еще не «занято», будет выполнен переход на экран выбора абонента.

  Имя, под которым текущий абонент вошел в службу видеотелефона, будет выведено на всех последующих экранах;

- экран выбора абонента — с именем, под которым было выполнено подключение к видеотелефону, списком **Абонент**, в котором выводятся все свободные абоненты (кроме текущего), кнопками **Связаться** и **Выход**.

  После выбора нужного абонента в списке **Абонент** и нажатия кнопки **Связаться** будет выполнен переход на экран ожидания вызова. При этом вызываемый клиент перейдет на экран получения вызова.

  При нажатии кнопки **Выход** будет выполнен возврат на экран входа;
- экран ожидания вызова — с именем вызываемого абонента, заданного на экране выбора абонента, и кнопкой **Отменить вызов**.

  При нажатии кнопки **Отменить вызов** оба абонента — и текущий, и вызываемый — вернутся на экран выбора абонента;
- экран получения вызова — с именем вызывающего абонента, кнопками **Принять вызов** и **Отменить вызов**.

  При нажатии кнопки **Принять вызов** текущий и вызывающий абоненты перейдут на экран сеанса.

  При нажатии кнопки **Отменить вызов** оба абонента перейдут на экран выбора абонента;
- экран сеанса — с видеопроигрывателем, который выводит видео, полученное от другого абонента, контрольным видеопроигрывателем и кнопкой **Завершить звонок**.

  При нажатии кнопки **Завершить звонок** оба абонента вернутся на экран выбора абонента.

Формат сообщений, пересылаемых клиентом и сервером друг другу, аналогичен применяемому в написанной ранее службе веб-чата (см. *упражнение 15.3*).

Вход в службу видеотелефона будет выполняться аналогично входу в веб-чат.

Чтобы вызвать какого-либо абонента, текущий абонент, находясь на экране выбора абонента, выберет будущего собеседника в списке **Абонент** и нажмет кнопку **Связаться**. При этом:

- вызывающий клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'busy'` (текущий абонент занят — проводит сеанс связи с другим абонентом);
    - `userName` — имя текущего абонента, под которым он вошел в службу.

    Этот сигнал в дальнейшем будет отправляться достаточно часто. Назовем его *сигналом занятости*;
- сервер:
  - уберет абонента, отправившего сигнал занятости, из списка свободных;
  - перешлет этот же сигнал остальным клиентам;

- остальные клиенты:
  - удалят занятого абонента из списка **Абонент** экрана вызова;
- вызывающий клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'invite'` (текущий абонент вызывает другого);
    - `userName` — имя текущего абонента;
    - `toUser` — имя вызываемого абонента (выбранное в списке **Абонент**);
  - переключится на экран ожидания вызова;
- сервер:
  - перешлет полученный сигнал вызываемому клиенту;
- вызываемый клиент:
  - отправит серверу сигнал занятости (был описан ранее);
  - выведет экран получения вызова.

Если вызывающий абонент на экране ожидания вызова нажмет кнопку **Отменить вызов**:

- вызывающий клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'cancel'` (вызывающий абонент отменяет вызов);
    - `userName` — имя текущего абонента;
    - `toUser` — имя вызываемого абонента;
  - переключится на экран выбора абонента;
- сервер:
  - перешлет полученный сигнал вызываемому клиенту (извещая того, что вызов отменен);
- вызываемый клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'free'` (текущий абонент свободен и доступен для следующего вызова);
    - `userName` — имя текущего абонента.

    Этот сигнал будет часто отправляться в дальнейшем. Назовем его *сигналом доступности*;
  - переключится на экран выбора абонента;
- сервер:
  - добавит абонента, отправившего сигнал доступности, в список свободных;
  - перешлет этот же сигнал всем остальным клиентам;

- остальные клиенты:
  - добавят освободившегося абонента в список **Абонент** экрана вызова.

Если вызываемый абонент на экране получения вызова нажмет кнопку **Отменить вызов**:

- вызываемый клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'decline'` (вызываемый абонент отменил вызов);
    - `userName` — имя текущего абонента;
    - `toUser` — имя вызывающего абонента;
  - переключится на экран выбора абонента;
- сервер:
  - перешлет полученный сигнал вызывающему клиенту (извещая того, что вызов отклонен);
- вызывающий клиент:
  - выполнит те же задачи, что и при отмене вызова (см. ранее).

Если же вызываемый абонент на экране получения вызова нажмет кнопку **Принять вызов**:

- вызываемый клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'agree'` (вызываемый абонент принял вызов);
    - `userName` — имя текущего абонента;
    - `toUser` — имя вызывающего абонента;
  - переключится на экран сеанса;
  - создаст соединение WebRTC, присоединит к нему медиапоток, выдаваемый камерой, и создаст приглашение;
- сервер:
  - перешлет полученный сигнал вызывающему клиенту (извещая того, что вызов принят);
- вызывающий клиент:
  - переключится на экран сеанса;
- вызываемый клиент:
  - после создания приглашения WebRTC — отправит серверу сигнал со свойствами:
    - `type` — `'offer'` (отправлено приглашение WebRTC);
    - `userName` — имя текущего абонента;

   - `toUser` — имя вызывающего абонента;
   - `data` — объект приглашения;
- сервер:
  - перешлет полученный сигнал с приглашением вызывающему клиенту;
- вызывающий клиент:
  - создаст соединение WebRTC, зарегистрирует в нем полученное приглашение, присоединит к соединению медиапоток с камеры и создаст согласие;
  - после создания согласия — отправит серверу сигнал со свойствами:
    - `type` — `'answer'` (отправлено согласие WebRTC);
    - `userName` — имя текущего абонента;
    - `toUser` — имя вызываемого абонента;
    - `data` — объект согласия;
- сервер:
  - перешлет полученный сигнал с согласием вызываемому клиенту;
- вызываемый клиент:
  - зарегистрирует полученное согласие в соединении WebRTC.

В процессе обмена претендентами WebRTC:

- один клиент-собеседник:
  - отправит серверу сигнал со свойствами:
    - `type` — `'candidate'` (отправлен претендент WebRTC);
    - `userName` — имя текущего абонента;
    - `toUser` — имя другого абонента;
    - `data` — объект претендента;
- сервер:
  - перешлет полученный сигнал с претендентом другому клиенту;
- другой клиент-собеседник:
  - зарегистрирует полученный претендент в соединении WebRTC.

Когда какой-либо из абонентов на экране сеанса нажмет кнопку **Завершить звонок**:

- текущий клиент:
  - отправит серверу сигнал со свойствами:
    - `type` — `'close'` (следует завершить сеанс);
    - `userName` — имя текущего абонента;
    - `toUser` — имя другого абонента;

  - завершит сеанс связи;
  - переключится на экран выбора абонента;
- сервер:
  - перешлет полученный сигнал клиенту-собеседнику;
- текущий клиент:
  - отправит сигнал доступности;
- клиент-собеседник:
  - отправит сигнал доступности;
  - завершит сеанс связи;
  - переключится на экран выбора абонента.

Инструменты для захвата видео доступны только на страницах, загруженных по протоколу HTTPS (см. *разд. 11.1*). А такие страницы могут устанавливать соединения только по защищенной редакции протокола WebSocket (см. *разд. 15.2.2.1*), что придется учесть при программировании.

16.2.2. Видеотелефон: сервер

Сервер этой веб-службы будет использоваться лишь для пересылки служебных сигналов. Сам сеанс связи пройдет посредством WebRTC, без участия сервера.

Файлу, хранящему код сервера службы, дадим имя server.php.

1. Загрузим библиотеку Workerman (как это сделать, было описано в *разд. 15.2.1*), распакуем ее куда-либо и переименуем папку Workerman-master в Workerman.
2. Создадим в той же папке, куда поместили библиотеку, файл server.php и откроем его в текстовом редакторе.

   Сервер видеотелефона будет аналогичен серверу веб-чата (см. *упражнение 15.3*) за следующими различиями:

   - он будет обрабатывать больше типов сигналов;
   - он будет хранить, помимо списка имен всех абонентов, еще и список имен абонентов, которые в текущий момент свободны, т. е. не заняты «разговорами»;
   - он будет работать по защищенному протоколу WebSocket (для чего мы используем сертификат открытого ключа из комплекта поставки пакета XAMPP).
3. Запишем в файл server.php код, создающий объект слушателя, списки всех и активных абонентов и пока «пустые» функции-обработчики:

```
<?php
require_once './workerman/autoloader.php';

use Workerman\Worker;
```

```
$aAllUsers = [];
$aFreeUsers = [];

$aContext = ['ssl' => ['local_cert'  =>
                         'c:/xampp/apache/conf/ssl.crt/server.crt',
                       'local_pk' =>
                         'c:/xampp/apache/conf/ssl.key/server.key',
                       'verify_peer' => false,
                       'allow_self_signed' => true] ];

$oWorker = new Worker('websocket://0.0.0.0:2345', $aContext);

$oWorker->transport = 'ssl';

$oWorker->onMessage = function ($connection, $data) {
};

$oWorker->onClose = function ($connection) {
};

Worker::runAll();
```

Массив с именами всех абонентов будем хранить в переменной `aAllUsers`, а массив имен только свободных абонентов — в переменной `aFreeUsers`.

Функция из свойства `onMessage` объекта слушателя будет обрабатывать сигналы разных типов. Однако какой-либо специфической обработки будут требовать лишь сигналы подключения `login`, занятости `busy` и доступности `free` (подробности — в *разд. 16.2.1*). Сигналы остальных типов будут обрабатываться единообразно.

4. Вставим в функцию из свойства `onMessage` слушателя код, извлекающий значение свойства `type` принятого объекта, и пока «пустое» выражение выбора:

```
$oWorker->onMessage = function ($connection, $data) {
    global $aAllUsers, $aFreeUsers, $oWorker;
    $aData = json_decode($data);
    switch ($aData->type) {
        case 'login':
        case 'busy':
        case 'free':
        default:
    }
};
```

Вход в службу видеотелефона будет выполняться почти так же, как вход в веб-чат (см. *упражнение 15.3*). За единственным исключением: помимо списка всех абонентов (хранится в переменной `aAllUsers`), новоприбывшего также следует занести в список свободных абонентов (из переменной `aFreeUsers`).

5. Добавим в выражение выбора, записанное ранее в тело функции из свойства onMessage, код, обрабатывающий вход нового абонента в службу:

```
switch ($aData->type) {
    case 'login':
        $userName = $aData->userName;
        if (key_exists($userName, $aAllUsers)) {
            $aR = ['type' => 'userNameExists'];
            $connection->close(json_encode($aR));
        } else {
            $aR = ['type' => 'ok',
                   'users' => array_keys($aFreeUsers)];
            $aAllUsers[$userName] = $connection;
            $aFreeUsers[$userName] = $connection;
            $connection->send(json_encode($aR));
            $aR = ['type' => 'userLoggedIn', 'userName' => $userName];
            $oR = json_encode($aR);
            foreach ($oWorker->connections as $oCon)
                if ($oCon != $connection)
                    $oCon->send($oR);
        }
        break;
    case 'busy':
        . . .
}
```

После получения сигнала занятости следует удалить отправившего его абонента из списка свободных (переменная aFreeUsers) и разослать сигнал остальным клиентам.

6. Добавим в выражение выбора, записанное ранее в тело функции из свойства onMessage, код, обрабатывающий сигнал занятости:

```
switch ($aData->type) {
    . . .
    case 'busy':
        unset($aFreeUsers[$aData->userName]);
        foreach ($oWorker->connections as $oCon)
            if ($oCon != $connection)
                $oCon->send($data);
        break;
    case 'free':
    . . .
}
```

После получения сигнала доступности нужно вновь добавить отправившего его абонента в список свободных и разослать сигнал остальным клиентам.

7. Добавим в выражение выбора, записанное ранее в тело функции из свойства onMessage, код, обрабатывающий сигнал доступности:

```
switch ($aData->type) {
    . . .
    case 'free':
        $aFreeUsers[$aData->userName] = $connection;
        foreach ($oWorker->connections as $oCon)
            if ($oCon != $connection)
                $oCon->send($data);
        break;
    default:
}
```

Остальные сигналы необходимо просто пересылать клиенту, имя которого указано в свойстве `toUser`.

8. Добавим в выражение выбора, записанное ранее в тело функции из свойства `onMessage`, код для обработки остальных сигналов:

```
switch ($aData->type) {
    . . .
    default:
        $toUser = $aData->toUser;
        $aAllUsers[$toUser]->send($data);
}
```

Разрыв соединения будет обрабатываться почти так же, как и у веб-чата (см. *упражнение 15.3*). Опять же, за тем исключением, что выходящего из службы абонента следует удалить из двух списков: всех и свободных абонентов.

9. Добавим в функцию из свойства `onClose` слушателя код, выполняющий необходимые действия в случае разрыва соединения:

```
$oWorker->onClose = function ($connection) {
    global $aAllUsers, $aFreeUsers, $oWorker;
    $userName = array_search($connection, $aAllUsers);
    if ($userName !== FALSE) {
        $aR = ['type' => 'userLoggedOut', 'userName' => $userName];
        $oR = json_encode($aR);
        unset($aFreeUsers[$userName]);
        unset($aAllUsers[$userName]);
        foreach ($oWorker->connections as $oCon)
            if ($oCon != $connection)
                $oCon->send($oR);
    }
};
```

На этом разработка сервера видеотелефонии завершена.

16.2.3. Видеотелефон: клиент

1. Найдем в папке 16\!sources сопровождающего книгу электронного архива (см. *приложение*) файлы index.html (веб-страница с интерфейсом видеотелефона), styles.css

(таблица стилей) и iceservers.js (код, объявляющий переменную `aICEServers` с массивом интернет-адресов пяти общедоступных серверов ICE). Скопируем их куда-либо на локальный диск.

Страница index.html содержит:

- экран входа — полностью аналогичный таковому из клиента веб-чата (см. *упражнение 15.3*);
- экран выбора абонента — тег `<section>` с якорем `invite`. В нем находятся:
  - очень важный текст (тег `<strong>`) с якорем `username1` — для вывода имени, под которым абонент выполнил вход в службу.

    На последующих экранах будут присутствовать аналогичные элементы с якорями, оканчивающимися на числа 2, 3 и 4;
  - веб-форма `frmInvite` — для запуска соединения с выбранным абонентом. Содержит список `lstUsers` (перечень свободных абонентов), изначально пустой и недоступный, и кнопку отправки данных `btnInvite` (запуск соединения с выбранным абонентом), также изначально недоступную;
  - обычную кнопку `btnLogout` — для выхода из службы;
- экран ожидания вызова — тег `<section>` с якорем `calling`, со следующими элементами:
  - очень важный текст `username2`;
  - очень важный текст `touser1` — для вывода имени абонента, с которым устанавливается сеанс связи.

    На последующих экранах будут присутствовать аналогичные элементы с якорями, оканчивающимися на числа 2 и 3;
  - обычная кнопка `btnCancel1` — для отмены вызова на стороне вызывающего абонента;
- экран получения вызова — тег `<section>` с якорем `calling`, с элементами:
  - очень важный текст `username3`;
  - очень важный текст `touser2`;
  - обычная кнопка `btnAgree` — для принятия вызова;
  - обычная кнопка `btnDecline` — для отмены вызова на стороне вызываемого абонента;
- экран сеанса — тег `<section>` с якорем `phone`, на котором присутствуют элементы:
  - очень важный текст `username4`;
  - очень важный текст `touser3`;
  - видеопроигрыватель `video_remote` — для вывода видео, получаемого от другого абонента;

- видеопроигрыватель `video_local` — контрольный.

  Контрольный видеопроигрыватель имеет значительно меньшие размеры, нежели выводящий видео от другого абонента, и располагается в правом нижнем углу последнего. Благодаря этому достигается эффект «картинка в картинке». Стили, оформляющие видеопроигрыватели, записаны в таблице стилей styles.css;

- обычная кнопка `btnCancel2` — для завершения звонка.

На экране сеанса также присутствуют:

- блок `messages` — для вывода текстовых сообщений;
- веб-форма `frmChat` — для набора текстовых сообщений. Содержит область редактирования `txtMessage` (текст сообщения), поле выбора файла `txtFile` (отправляемый файл) и кнопку отправка данных `btnSend` (отправка текстового сообщения), изначально недоступную.

Эти элементы понадобятся при выполнении самостоятельного упражнения.

2. Откроем в текстовом редакторе копию страницы index.html и вставим в конце HTML-кода привязку файла сценария script.js, который вскоре создадим:

```
<html>
    . . .
</html>
<script src="iceservers.js" type="text/javascript"></script>
<script src="script.js" type="text/javascript"></script>
```

3. Создадим файл script.js в той же папке, где находятся файлы index.html, styles.css и iceservers.js, откроем его в текстовом редакторе и запишем выражения, получающие доступ к приведенным ранее элементам страницы:

```
const oLogin = document.getElementById('login');
const oInvite = document.getElementById('invite');
const oCalling = document.getElementById('calling');
const oCalled = document.getElementById('called');
const oPhone = document.getElementById('phone');
const frmLogin = document.getElementById('frmLogin');
const txtUserName = document.getElementById('txtUserName');
const btnLogin = document.getElementById('btnLogin');
const oUserName1 = document.getElementById('username1');
const oUserName2 = document.getElementById('username2');
const oUserName3 = document.getElementById('username3');
const oUserName4 = document.getElementById('username4');
const oToUser1 = document.getElementById('touser1');
const oToUser2 = document.getElementById('touser2');
const oToUser3 = document.getElementById('touser3');
const lstUsers = document.getElementById('lstUsers');
const btnInvite = document.getElementById('btnInvite');
const btnLogout = document.getElementById('btnLogout');
const btnCancel1 = document.getElementById('btnCancel1');
```

```
const btnAgree = document.getElementById('btnAgree');
const btnDecline = document.getElementById('btnDecline');
const oVideoRemote = document.getElementById('video_remote');
const oVideoLocal = document.getElementById('video_local');
const btnCancel2 = document.getElementById('btnCancel2');
```

Интернет-адрес сервера будем формировать так же, как делали это при написании веб-чата (см. *упражнение 15.3*). Только в качестве обозначения протокола укажем **wss://** — защищенный WebSocket.

4. Добавим выражение, объявляющее константу для хранения интернет-адреса сервера чата:

```
const serverURL = 'wss://' + location.hostname + ':2345';
```

Нужно объявить ряд переменных: `oWS` (объект соединения WebSocket), `userName` (имя текущего абонента), `toUser` (имя его собеседника), `oRTC` (объект соединения WebRTC), `oLocalStream` (медиапоток с камеры) и `oRemoteStream` (медиапоток, получаемый от собеседника).

5. Добавим выражение, объявляющее необходимые переменные:

```
let oWS, userName, toUser, oRTC, oLocalStream, oRemoteStream;
```

Обработчик события `input` поля ввода `txtUserName` (имя абонента для входа в службу), который делает кнопку `btnLogin` доступной только в том случае, если в поле что-либо занесено, можно взять из кода веб-чата. Оттуда же можно взять обработчик события `submit` веб-формы `frmLogin`, создающий соединение WebSocket, и функцию `tryLogin()`, обработчик события `open` соединения, выполняющую вход в службу.

6. Добавим обработчики событий `input` кнопки `btnLogin`, `submit` веб-формы `frmLogin` и функцию `tryLogin()`, взяв их из кода веб-чата (см. *упражнение 15.3, шаги 6–8*).

Функция `tryLoginAnswer()`, обработчик события `message` соединения, будет выполнять те же действия, что и одноименная функция из кода чата. За следующими различиями: имя текущего абонента будет занесено в четыре элемента страницы (`username1`, `username2`, `username3` и `username4`), переход будет выполнен на экран вызова `invite` с веб-формой `frmInvite`, а список свободных абонентов `lstUsers` перед заполнением понадобится очистить полностью (поскольку в нем нет пункта **Всем**).

7. Добавим объявление функции `tryLoginAnswer()`:

```
function tryLoginAnswer(evt) {
    const oM = JSON.parse(evt.data);
    if (oM.type == 'userNameExists') {
        txtUserName.focus();
        window.alert('Заданное имя уже занято');
    } else {
        oLogin.style.display = 'none';
        oInvite.style.display = 'block';
```

```
        userName = txtUserName.value;
        oUserName1.textContent = userName;
        oUserName2.textContent = userName;
        oUserName3.textContent = userName;
        oUserName4.textContent = userName;
        frmInvite.reset();
        btnInvite.disabled = true;
        lstUsers.innerHTML = '';
        for (let userName of oM.users)
            addUser(userName);
        oWS.removeEventListener('open', tryLogin);
        oWS.removeEventListener('message', tryLoginAnswer);
        oWS.addEventListener('message', getMessage);
    }
}
```

Настала пора заняться функцией `getMessage()` — новым обработчиком события `message` соединения. Сразу же реализуем в ней обработку сигналов `userLoggedIn` (новый абонент вошел в службу), `free` (абонент освободился), `userLoggedOut` (абонент вышел из службы) и `busy` (абонент занят). При получении сигналов `userLoggedIn` и `free` присутствующее в составе сигнала имя абонента нужно добавить в список `lstUsers`, а при получении сигналов `userLoggedOut` и `busy` — удалить оттуда.

Функция `addUser(<имя>)` добавит заданное *имя* абонента в список `lstUsers` и, если список не пуст, сделает его доступным. Функция `removeUser(<имя>)` удалит заданное *имя* абонента из списка и, если список пуст, сделает недоступным и список, и кнопку вызова `btnInvite`.

8. Добавим объявления функций `getMessage()`, `addUser()` и `removeUser()`:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        case 'userLoggedIn':
        case 'free':
            addUser(oM.userName);
            break;
        case 'userLoggedOut':
        case 'busy':
            removeUser(oM.userName);
            break;
    }
}

function addUser(userName) {
    const oOpt = document.createElement('option');
    oOpt.value = userName;
```

```
        oOpt.textContent = userName;
        lstUsers.add(oOpt);
        if (lstUsers.length > 0)
            lstUsers.disabled = false;
}

function removeUser(userName) {
    const oOpt = lstUsers.querySelector('option[value=' +
                                        userName + ']');
    lstUsers.removeChild(oOpt);
    if (lstUsers.length == 0) {
        lstUsers.disabled = true;
        btnInvite.disabled = true;
    }
}
```

На экране вызова `invite`, в веб-форме `frmInvite`, в списке абонентов `lstUsers` изначально не выбран ни один пункт, и кнопка вызова `btnInvite` недоступна. Доступной она должна стать лишь после того, как в списке будет выбран вызываемый абонент. Реализуем это, обрабатывая событие `change` списка абонентов.

9. Добавим обработчик события `change` списка `lstUsers`:

```
lstUsers.addEventListener('change', () => {
    if (lstUsers.selectedIndex > -1)
        btnInvite.disabled = false;
});
```

После нажатия кнопки `btnInvite` следует отправить серверу сигнал занятости `busy`, сохранить имя вызываемого абонента, выбранное в списке `lstUsers`, в переменной `toUser`, отправить сигнал вызова `invite`, переключиться на экран ожидания вызова `calling` и вывести имя вызываемого абонента в элементе `touser1`, что присутствует на этом экране.

10. Добавим обработчик события `submit` веб-формы `frmInvite`, выполняющий эти действия:

```
frmInvite.addEventListener('submit', (evt) => {
    evt.preventDefault();
    let oM = { type: 'busy', userName: userName };
    oWS.send(JSON.stringify(oM));
    toUser = lstUsers.options[lstUsers.selectedIndex].value;
    oM = { type: 'invite', userName: userName, toUser: toUser };
    oWS.send(JSON.stringify(oM));
    oInvite.style.display = 'none';
    oCalling.style.display = 'block';
    oToUser1.textContent = toUser;
});
```

После нажатия кнопки выхода `btnLogout` нужно закрыть соединение WebSocket, переключиться на экран входа `login` и сбросить форму входа `frmLogin`.

11. Добавим обработчик события click кнопки btnLogout, который сделает это:

```
btnLogout.addEventListener('click', () => {
    oWS.close(1000);
    oWS = undefined;
    oLogin.style.display = 'block';
    oInvite.style.display = 'none';
    frmLogin.reset();
    btnLogin.disabled = true;
});
```

Вызываемый клиент, получив сигнал вызова invite, отправит сигнал занятости, переключится на экран получения вызова called и выведет имя вызывающего абонента в элементе touser2, который находится на этом экране.

12. Добавим в функцию getMessage() обработку сигнала invite, а также объявление функции inviteUser(*<имя вызывающего абонента>*), которая выполнит указанные ранее действия:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'invite':
            inviteUser(oM.userName);
            break;
    }
}

function inviteUser(inviter) {
    const oA = { type: 'busy', userName: userName };
    oWS.send(JSON.stringify(oA));
    oInvite.style.display = 'none';
    oCalled.style.display = 'block';
    toUser = inviter;
    oToUser2.textContent = toUser;
}
```

Теперь, если вызывающий абонент решит отменить вызов и нажмет кнопку отмены btnCancel1, находящуюся на экране ожидания вызова calling, клиент должен послать серверу сначала сигнал отмены вызова cancel, а потом — сигнал доступности free, переключиться на экран вызова invite и сбросить веб-форму frmInvite.

13. Добавим обработчик события click кнопки btnCancel1, выполняющий эти действия:

```
btnCancel1.addEventListener('click', () => {
    let oM = { type: 'cancel', userName: userName, toUser: toUser };
    oWS.send(JSON.stringify(oM));
    oM = { type: 'free', userName: userName };
```

```
        oWS.send(JSON.stringify(oM));
        oCalling.style.display = 'none';
        oInvite.style.display = 'block';
        frmInvite.reset();
        btnInvite.disabled = true;
    });
```

Вызываемый клиент, получив сигнал отмены вызова `cancel`, отправит сигнал доступности, переключится на экран вызова `invite` и сбросит веб-форму `frmInvite`.

14. Добавим в функцию `getMessage()` обработку сигнала `cancel`, а также объявление функции `callIsCancelled()`, которая выполнит эти действия:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'cancel':
            callIsCancelled();
            break;
    }
}

function callIsCancelled() {
    const oM = { type: 'free', userName: userName };
    oWS.send(JSON.stringify(oM));
    oCalled.style.display = 'none';
    oInvite.style.display = 'block';
    frmInvite.reset();
    btnInvite.disabled = true;
}
```

Вернемся к вызываемому абоненту. Если он решит отклонить вызов, то нажмет кнопку `btnDecline` на экране получения вызова `called`. Тогда вызываемый клиент должен отправить сигнал отклонения вызова `decline`, сигнал доступности и переключиться на экран вызова `invite`.

15. Добавим обработчик события `click` кнопки `btnDecline`, делающий это:

```
btnDecline.addEventListener('click', () => {
    let oM = { type: 'decline', userName: userName, toUser: toUser };
    oWS.send(JSON.stringify(oM));
    oM = { type: 'free', userName: userName };
    oWS.send(JSON.stringify(oM));
    oCalled.style.display = 'none';
    oInvite.style.display = 'block';
    frmInvite.reset();
    btnInvite.disabled = true;
});
```

Вызывающий клиент, получив сигнал отклонения вызова `decline`, должен поступить так же, как вызываемый клиент — при получении сигнала `cancel` (см. ранее). На *шаге 14* мы объявили функцию `callIsCancelled()`, производящую необходимые действия, — так используем ее и здесь.

16. Добавим в функцию `getMessage()` обработку сигнала `decline`:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
        . . .
        case 'cancel':
        case 'decline':
            callIsCancelled();
            break;
    }
}
```

Если же вызываемый абонент решит принять вызов, он нажмет кнопку `btnAgree` на экране получения вызова `called`. В ответ вызываемый клиент отправит сигнал приема вызова `agree`, переключится на экран сеанса `phone`, создаст соединение WebRTC, присоединит к нему медиапоток с камеры, создаст приглашение и отправит его в составе сигнала `offer`.

Создавать соединение WebRTC будет функция `createRTC()`, а присоединять к нему медиапоток — функция `getMediaTracks()`. Мы объявим их чуть позже.

17. Добавим обработчик события `click` кнопки `btnAgree`, выполняющий эти действия:

```
btnAgree.addEventListener('click', async function () {
    let oM = { type: 'agree', userName: userName, toUser: toUser };
    oWS.send(JSON.stringify(oM));
    oCalled.style.display = 'none';
    oPhone.style.display = 'block';
    oToUser3.textContent = toUser;
    createRTC();
    await getMediaTracks();
    const oOffer = await oRTC.createOffer();
    await oRTC.setLocalDescription(oOffer);
    oM = { type: 'offer', userName: userName, toUser: toUser,
                         data: oRTC.localDescription };
    oWS.send(JSON.stringify(oM));
});
```

Поскольку в теле функции-обработчика будут выполняться асинхронные операции, мы сделали ее асинхронной.

Вызывающий клиент, получив сигнал приема вызова `agree`, должен переключиться на экран сеанса `phone` и вывести в находящемся на нем элементе `touser3` имя вызываемого абонента.

18. Добавим в функцию `getMessage()` обработку сигнала `agree`, а также объявление функции `callIsAgreed()`, которая выполнит эти действия:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'agree':
            callIsAgreed();
            break;
    }
}

function callIsAgreed() {
    oCalling.style.display = 'none';
    oPhone.style.display = 'block';
    oToUser3.textContent = toUser;
}
```

Функция `createRTC()`, создающая соединение WebRTC, также должна привязать обработчики к событиям `icecandidate` и `track` этого соединения. Используем в качестве обработчиков функции `sendCandidate()` и `joinToBroadcasting()`, которые объявим позже.

19. Добавим объявление функции `createRTC()`:

```
function createRTC() {
    oRTC = new RTCPeerConnection({ iceServers: aICEServers });
    oRTC.addEventListener('icecandidate', sendCandidate);
    oRTC.addEventListener('track', joinToBroadcasting);
}
```

Функция `getMediaTracks()`, присоединяющая к созданному ранее соединению WebRTC медиапоток от камеры, также должна выводить его в контрольном видеопроигрывателе `video_local`. Поскольку в теле функции выполняются асинхронные операции, саму функцию тоже сделаем асинхронной.

20. Добавим объявление функции `getMediaTracks()`:

```
async function getMediaTracks() {
    oLocalStream = await navigator
                         .mediaDevices
                         .getUserMedia({ video: true, audio: true });
    for (let oTr of oLocalStream.getTracks())
        oRTC.addTrack(oTr, oLocalStream);
    oVideoLocal.srcObject = oLocalStream;
}
```

Вызывающий клиент, получив от вызываемого сигнал приглашения `offer`, должен создать соединение WebRTC, зарегистрировать в нем полученное при-

глашение, присоединить к соединению медиапоток от камеры, создать согласие и отправить его вызываемому клиенту в составе сигнала `answer`.

21. Добавим в функцию `getMessage()` обработку сигнала `offer` и объявим асинхронную функцию `getOffer()`, выполняющую нужные действия:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'offer':
            getOffer(oM.data);
            break;
    }
}

async function getOffer(data) {
    createRTC();
    await oRTC.setRemoteDescription(data);
    await getMediaTracks();
    const oAnswer = await oRTC.createAnswer();
    await oRTC.setLocalDescription(oAnswer);
    const oS = { type: 'answer', userName: userName, toUser: toUser,
                                  data: oRTC.localDescription };
    oWS.send(JSON.stringify(oS));
}
```

Вызываемый клиент, получив от вызывающего сигнал согласия `answer`, должен зарегистрировать в своем соединении WebRTC полученное с сигналом согласие.

22. Добавим в функцию `getMessage()` обработку сигнала `answer` и объявим асинхронную функцию `getAnswer()`, выполняющую нужные действия:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'answer':
            getAnswer(oM.data);
            break;
    }
}

async function getAnswer(data) {
    await oRTC.setRemoteDescription(data);
}
```

Обменявшись приглашением и согласием, клиенты начнут генерировать и пересылать друг другу претенденты. В функции `sendCandidate()`, ранее привязанной к событию `icecandidate` соединения в качестве обработчика, следует от-

править другому клиенту сгенерированный претендент в составе сигнала `candidate`. Получив этот сигнал, другой клиент зарегистрирует претендент в своем соединении.

23. Объявим функцию `sendCandidate()`, добавим в функцию `getMessage()` обработку сигнала `candidate` и объявим асинхронную функцию `getCandidate()`, регистрирующую полученный претендент:

```
function sendCandidate(evt) {
    if (evt.candidate) {
        const oM = { type: 'candidate', userName: userName,
                     toUser: toUser,  data: evt.candidate };
        oWS.send(JSON.stringify(oM));
    }
}

function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'candidate':
            getCandidate(oM.data);
            break;
    }
}

async function getCandidate(data) {
    await oRTC.addIceCandidate(data);
}
```

«Договорившись» по поводу параметров соединения, клиенты начнут пересылать друг другу медиапотоки, захваченные со своих камер. Чтобы каждый из абонентов смог увидеть своего собеседника, в функции `joinToBroadcasting()`, ранее привязанной к событию `track` соединения в качестве обработчика, нужно вывести получаемый от другого клиента медиапоток в видеопроигрывателе `video_remote`.

24. Объявим функцию `joinToBroadcasting()`:

```
function joinToBroadcasting(evt) {
    oRemoteStream = evt.streams[0];
    oVideoRemote.srcObject = oRemoteStream;
}
```

Чтобы завершить сеанс связи, абонент нажмет кнопку `btnCancel2` на экране `phone`. В ответ клиент отправит собеседнику сигнал разрыва связи `close` и завершит вещание.

Код, завершающий вещание, будет вызываться еще в одном месте. Поэтому вынесем его в отдельную функцию `stopCall()`, которую объявим позже.

25. Добавим обработчик события `click` кнопки `btnCancel2`, выполняющий эти действия:

```
btnCancel2.addEventListener('click', () => {
    const oM = { type: 'close', userName: userName, toUser: toUser };
    oWS.send(JSON.stringify(oM));
    stopCall();
});
```

Функция `stopCall()` при завершении вещания, помимо действий, описанных в *разд. 16.1.6*, также должна переключаться на экран вызова `invite` и сбрасывать веб-форму `frmInvite`.

26. Добавим объявление функции `stopCall()`:

```
function stopCall() {
    for (let oTr of oRemoteStream.getTracks())
        oTr.stop();
    for (let oTr of oLocalStream.getTracks())
        oTr.stop();
    oVideoRemote.srcObject = undefined;
    oVideoLocal.srcObject = undefined;
    oRemoteStream = undefined;
    oLocalStream = undefined;
    oRTC.removeEventListener('icecandidate', sendCandidate);
    oRTC.removeEventListener('track', joinToBroadcasting);
    oRTC.close();
    oRTC = undefined;
    const oM = { type: 'free', userName: userName };
    oWS.send(JSON.stringify(oM));
    oPhone.style.display = 'none';
    oInvite.style.display = 'block';
    frmInvite.reset();
    btnInvite.disabled = true;
}
```

Получив сигнал завершения сеанса `close`, другой клиент также должен закончить вещание вызовом только что объявленной функции `stopCall()`.

27. Добавим в функцию `getMessage()` обработку сигнала `close`:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        . . .
        case 'close':
            stopCall();
    }
}
```

28. Откроем командную строку, перейдем в папку, в которой находится сервер службы, и запустим его с помощью команды, описанной в *разд. 15.2.4*.

29. Скопируем файлы index.html, styles.css, iceservers.js и script.js в корневую папку веб-сервера и запустим веб-сервер.
30. Откроем веб-обозреватель и выполним переход по интернет-адресу **https://localhost/**. Мы увидим экран входа (рис. 16.1).

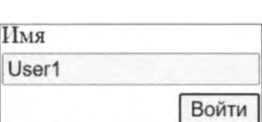


Рис. 16.1. Видеотелефон: экран входа

Рис. 16.2. Видеотелефон экран выбора абонента

Введем какое-либо имя абонента в поле **Имя** и нажмем кнопку **Войти**. Приложение переключится на экран выбора абонента (рис. 16.2).

Войдем в службу с какого-либо мобильного устройства, открыв веб-обозреватель и перейдя по интернет-адресу формата **https://<*IP-адрес компьютера, на котором запущен сервер*>/** (как узнать этот IP-адрес, было показано в *упражнении 11.4*). Наберем в поле **Имя** любое другое имя и нажмем кнопку **Войти**.

На основном компьютере выберем появившегося абонента в списке **Абонент** и нажмем кнопку **Связаться**. Клиент, запущенный на основном компьютере, переключится на экран ожидания вызова (рис. 16.3), а клиент на мобильном устройстве — на экран получения вызова (рис. 16.4).

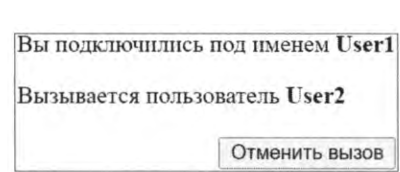


Рис. 16.3. Видеотелефон: экран ожидания вызова

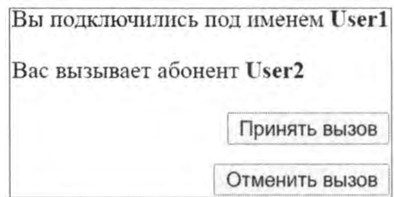


Рис. 16.4. Видеотелефон экран получения вызова

На мобильном устройстве примем вызов, нажав кнопку **Принять вызов** экрана получения вызова. Оба клиента переключатся на экран сеанса (рис. 16.5).

Завершим сеанс связи, нажав кнопку **Завершить звонок** экрана сеанса одного из клиентов. Оба клиента должны переключиться на экран выбора абонента.

Рис. 16.5. Видеотелефон: экран сеанса

Попробуем связаться еще раз, но на этот раз отменим вызов, нажав соответствующую кнопку. Напоследок выйдем из службы, нажав кнопку **Выход**.

16.3. Обмен произвольными данными посредством WebRTC

Технология WebRTC также позволяет клиентам пересылать друг другу произвольные данные. Предоставляемые для этого программные инструменты схожи с таковыми, обеспечиваемыми протоколом WebSocket (см. *урок 15*).

Далее описаны действия, необходимые для пересылки произвольных данных между клиентами средствами WebRTC.

1. Создание соединения WebRTC (см. *разд. 16.1.1*).
2. Создание канала данных.

   > *Канал данных* — виртуальный канал, создаваемый в рамках существующего соединения WebRTC и предназначенный для обмена произвольными данными. Должен иметь произвольное имя и уникальный целочисленный идентификатор.

   Имя канала используется исключительно в описательных целях (например, чтобы вывести его на экран). Идентификатор выбирается произвольно в диапазоне от 0 до 65 535 — таким образом, в рамках одного соединения можно создать 65 536 независимых каналов данных.

   Внимание!

   Каналы данных функционируют независимо от вещания. Поэтому обмениваться через них данными можно, не прерывая пересылку медиапотоков.

3. Обмен приглашением, согласием и претендентами (см. *разд. 16.1.3* и *16.1.4*).
4. Как только клиенты «договорятся» о приемлемых параметрах соединения — собственно обмен данными.
5. Закрытие канала данных.
6. Закрытие соединения WebRTC.

16.3.1. Создание канала данных. Равноправный режим

Создание канала данных выполняется вызовом метода `createDataChannel()` объекта соединения WebRTC:

```
createDataChannel(<имя канала>[, <параметры>=undefined])
```

Имя канала указывается в виде строки, его длина не должна превышать 65 536 символов. *Параметры* указываются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Поддерживаются следующие параметры:

- `negotiated` — следует указать `true`, чтобы задействовать равноправный режим.

  > *Равноправный режим* канала данных — требует создания канала данных обоими клиентами.

  Применение равноправного режима позволяет несколько упростить программирование. Однако в этом случае каждый из клиентов должен создать канал *с одним и тем же идентификатором*, что в ряде случаях может быть проблематично.

  Значение параметра по умолчанию — `false` (задействуется режим «главный — подчиненный», описываемый далее);

- `id` — идентификатор создаваемого канала в виде целого числа от 0 до 65 535;
- `ordered` — если `true`, сообщения, отправляемые одним клиентом, достигнут другого клиента в том порядке, в котором они были отправлены. Если `false`, порядок получения сообщений может не соблюдаться. Значение по умолчанию — `true`.

Метод возвращает созданный канал данных, представленный объектом класса `RTCDataChannel`.

Пример:

```
const oRTC = new RTCPeerConnection(oPs);
const oDC = oRTC.createDataChannel('chat',
                                   { negotiated: true, id: 1000 });
```

После создания канала данных в нем возникает событие `open`. Его обработчику в качестве параметра передается объект класса `RTCDataChannelEvent`, содержащий сведения о событии. Доступное только для чтения свойство `channel` этого класса хранит объект текущего канала.

16.3.2. Отправка данных

Для отправки клиенту-собеседнику указанных *данных* применяется метод `send(<данные>)` класса `RTCDataChannel`. *Данные* должны представлять собой строку, объект класса `Blob` или `ArrayBuffer`. Можно отправлять данные в формате JSON, представленные в виде строки.

Примеры:

```
oDC.send('Привет!');

oMessage = { type: 'message', content: 'Как дела?' };
oDC.send(JSON.stringify(oMessage));
```

16.3.3. Получение данных

При получении данных от клиента-собеседника в объекте канала данных возникает событие `message`. Обработчику этого события передается объект класса `MessageEvent`, хранящий сведения о событии. В их числе присутствуют и сами полученные данные — их можно извлечь из свойства `data`.

Пример:

```
// Получение данных в формате JSON
oDC.addEventListener('message', (evt) => {
    const oMessage = JSON.parse(evt.data);
    const type = oMessage.type;
    const content = oMessage.content;
});
```

16.3.4. Закрытие канала данных

Закрыть канал данных можно, вызвав у представляющего его объекта метод `close()`. Пример:

```
oDC.close();
```

После закрытия канала в его объекте возникает событие `close`.

16.3.5. Режим «главный — подчиненный»

> *Режим «главный — подчиненный»* канала данных — требует создания канала лишь одним из клиентов («главным»). Второй клиент («подчиненный») присоединяется к уже созданному каналу.

Этот режим не требует от обоих клиентов согласовывать параметры создаваемого канала, что может пригодиться в каких-либо случаях. Однако его использование несколько усложняет программирование.

Обмен данными в режиме «главный — подчиненный» производится так же, как и в равноправном режиме, за исключениями, приведенными далее.

- Канал данных создается только одним из клиентов — «главным», как было сказано ранее.
- При создании канала параметру `negotiated` следует дать значение `false` или вообще не указывать этот параметр (поскольку `false` — его значение по умолчанию).

  Параметр `id` также указывать необязательно — в этом случае веб-обозреватель сам присвоит идентификатор создаваемому каналу данных, выбрав его из еще не «занятых» идентификаторов.
- Другой, «подчиненный», клиент должен присоединиться к созданному каналу — обрабатывая событие `datachannel` соединения WebRTC, которое возникает при появлении нового канала данных в составе текущего соединения. Обработчику этого события передается объект класса `RTCDataChannelEvent`, в свойстве `channel` которого хранится объект нового канала данных.

  Класс `RTCDataChannel`, представляющий канал данных, поддерживает следующие полезные свойства, доступные только для чтения:

  - `label` — имя канала данных, указанное при его создании;
  - `id` — идентификатор канала данных;
  - `ordered` — значение параметра `ordered`.

  Далее второй, «подчиненный», клиент может посылать и получать данные, воспользовавшись инструментами, описанными в *разд. 16.3.2* и *16.3.3*.

Пример:

```
// «Главный» клиент
const oRTC2 = new RTCPeerConnection(oPs);
```

```
const oDC2 = oRTC.createDataChannel('chat');
oDC2.addEventListener('message', (evt) => {
    const oMessage = JSON.parse(evt.data);
    . . .
});
oDC2.send('Привет!');

// «Подчиненный» клиент
const oRTC3 = new RTCPeerConnection(oPs);
oDC3.addEventListener('datachannel', (evt) => {
    // Получаем канал данных и присоединяемся к нему
    const oChannel = evt.channel;
    oDC3.addEventListener('message', (evt) => {
        const oMessage = JSON.parse(evt.data);
        . . .
    });
    oDC3.send('Взаимно!');
});
```

16.4. Самостоятельное упражнение

Реализуйте в службе видеотелефона, написанной при выполнении *упражнения 16.2*, средства для пересылки сообщений, аналогичные предоставляемым чатом (см. *упражнение 15.3*).

Все необходимые элементы управления уже присутствуют на странице index.html, а все необходимые стили — в таблице стилей styles.css. Чтобы упростить программирование, используйте равноправный режим. Создаваемому каналу данных можете дать имя, например, `chat` и идентификатор 1000.

ЧАСТЬ VI
Прогрессивные веб-приложения (PWA)

- PWA
- Посредники
- Программируемый кэш
- Всплывающие оповещения
- Манифест PWA

Урок 17
Введение в прогрессивные веб-приложения

Три поколения веб-сайтов

Прогрессивные веб-приложения

Все существующие в настоящее время веб-сайты можно разделить на следующие три поколения:

- первое — *статические* сайты, — чьи страницы хранятся в обычных HTML-файлах.

  Достоинство: простота разработки. Недостаток: сложность пополнения и правки опубликованных на страницах материалов;

- второе — *динамические* — представляют собой набор серверных веб-приложений, генерирующих полноценные страницы.

  Достоинства: простота пополнения и правки опубликованных материалов; возможность реализовать расширенную функциональность (поисковые, почтовые службы, интернет-магазины и т. п.). Недостатки: усложнение разработки; необходимость установки дополнительных программ и ведения баз данных;

- третье — *сверхдинамические* — представляют собой комбинацию обычных страниц (фронтенда) и серверных веб-приложений (бэкенда). Бэкенд генерирует результаты в каком-либо компактном формате (обычно JSON), а веб-сценарии, находящиеся в составе фронтенда, получают эти данные, декодируют и выводят на экран.

  Достоинство (перед динамическими сайтами второго поколения): значительное уменьшение объема информации, пересылаемой по сети. Недостаток: существенное усложнение программирования.

  Тем не менее, несмотря на такой недостаток, сверхдинамические сайты в настоящее время наиболее востребованы.

Очень часто сверхдинамический сайт в своем составе имеет лишь одну веб-страницу (*одностраничный сайт*). В процессе навигации по такому сайту на единственной странице выводится тот или иной экран с теми или иными элементами. Нечто подобное мы реализовали в клиентах веб-чата (см. *упражнение 15.3*) и интернет-видеотелефона (см. *упражнение 16.2*).

Вполне логичным шагом выглядит более глубокая интеграция таких сайтов в операционную систему.

17.1. Что такое прогрессивное веб-приложение (PWA)?

> *Прогрессивное веб-приложение* (*PWA*, Progressive Web Application) — одностраничный веб-сайт, устанавливаемый в операционной системе, подобно обычному приложению.

Такое приложение представляет собой дальнейший этап развития обычных сайтов — шаг в направлении к обычным приложениям.

Прогрессивное веб-приложение:

- устанавливается в операционной системе непосредственно из веб-обозревателя;
- запускается посредством нажатия на ярлык, находящийся на рабочем столе;
- открывается в отдельном окне веб-обозревателя, из которого была выполнена его установка.

  Это окно не имеет обычных для веб-обозревателя панели инструментов и главного меню и содержит лишь ограниченный набор элементов управления. В частности, на настольных платформах это окно имеет в заголовке лишь кнопку открытия системного меню, посредством которого можно вырезать, скопировать выделенный текст в буфер обмена, вставить текст из буфера обмена в какое-либо поле ввода, выполнить некоторые другие действия, а также удалить приложение.

  На настольных платформах веб-обозреватель автоматически сохраняет местоположение и размеры окна с приложением перед его закрытием и восстанавливает при следующем запуске приложения;
- выводит всплывающие оповещения, опять же, средствами операционной системы;
- при этом фактически являясь веб-страницей, открываемой в веб-обозревателе.

Преимущества прогрессивных веб-приложений:

- перед обычными одностраничными сайтами:
  - существенное упрощение запуска (в самом деле, что может быть проще щелчка на ярлыке...);
  - при соблюдении принципов, описываемых далее, — существенное сокращение объема загружаемой по сети информации;
- перед традиционными приложениями:
  - могут программироваться на языках HTML, CSS и JavaScript, как и обычные веб-страницы;
  - могут с легкостью быть созданы на основе уже существующего одностраничного сайта;
  - собственно, представляют собой обычную веб-страницу (вследствие чего могут индексироваться поисковыми службами и распространяться путем передачи их интернет-адресов);

- работают на любой программной платформе, в которой присутствует более или менее современный веб-обозреватель.

Пожалуй, единственный недостаток PWA перед традиционными приложениями: возможность выполнения весьма ограниченного набора функций. PWA не имеют доступа к программному интерфейсу операционной системы, системным данным (в частности, к системному реестру) и локальной файловой системе (кроме файлов, непосредственно выбранных пользователем, — как и обычные страницы).

17.2. Основные принципы разработки PWA

При разработке прогрессивного веб-приложения рекомендуется руководствоваться следующими принципами.

- PWA, по возможности, должно иметь *минимальные отличия от традиционного приложения*:
  - должно устанавливаться в операционной системе, как традиционное приложение. При этом на рабочем столе должен появиться запускающий приложение ярлык;
  - при возникновении какого-либо события (например, приходе нового сообщения в чате) должно выводить всплывающее оповещение. Нажав на такое оповещение, пользователь переключится на приложение, которое его вывело.

  Достигается все это правильным заполнением манифеста PWA и использованием инструментов для вывода оповещений, встроенных в веб-обозреватель (будут описаны в следующих главах).

- PWA должно обеспечивать *уменьшение объема информации*, пересылаемой по сети (сетевого трафика).

  Бóльшая часть современных тарифов сотовой связи предполагает взимание платы в зависимости от объема загружаемой из сети информации. Поэтому, сократив этот объем, мы сэкономим деньги пользователей.

  Сократить сетевой трафик можно, принудительно сохраняя в кэше веб-обозревателя ключевые файлы, составляющие веб-приложение: саму страницу приложения, таблицу стилей, файлы веб-сценариев, изображения, являющиеся частью оформления, и т. п. Для этого могут быть использованы посредники и программируемый кэш, описываемые в следующих главах.

- PWA, по возможности, должно обеспечивать *адаптивность*.

  Желательно, чтобы PWA могли работать как на традиционных компьютерах, так и на мобильных устройствах с экранами разных размеров. Более того, некоторые источники рекомендуют писать прогрессивные приложения изначально под мобильные устройства, а уже потом при необходимости адаптировать их для работы на традиционных компьютерах.

- PWA должно обеспечивать *безопасность* — использовать в работе только безопасные протоколы: HTTPS и защищенную редакцию WebSocket.

При разработке PWA часто применяются такие технологии, как посредники и всплывающие оповещения. Они для успешной работы требуют, чтобы приложение загружалось по протоколу HTTPS. Применение для этой цели HTTP вызовет программную ошибку.

Внимание!

Веб-сервер, посредством которого распространяется прогрессивное веб-приложение, должен использовать надежный сертификат, выданный каким-либо сертификационным центром. Самоподписанные сертификаты использовать не допускается (подробнее о сертификатах и защищенных протоколах рассказывалось в *разд. 15.2.2.1*).

17.3. Отладка PWA

Чтобы разработчикам PWA для отладки приложений не пришлось приобретать доверенный сертификат (надо сказать, очень дорогой), в технологии оставлена «лазейка».

Прогрессивное приложение будет нормально работать, будучи загруженным по протоколу HTTP с локального хоста — интернет-адреса **http://localhost/**. При этом также будут нормально функционировать реализованные в приложении посредники и всплывающие оповещения.

Урок 18
Посредники и программируемый кэш

Посредники

Программируемый кэш

Области кэша

Задачи посредника

Чтобы уменьшить объем сетевого трафика при загрузке PWA, ключевые файлы, составляющие приложение, принудительно сохраняют в кэше веб-обозревателя. Код, сохраняющий файлы в кэше и впоследствии выдающий их в ответ на запросы фронтенда, помещают в посредниках.

18.1. Посредники

> *Посредник* (service worker, SW)[1] — программа, которая выступает как прослойка между фронтендом и бэкендом, перехватывает клиентские запросы, отправляемые фронтендом бэкенду, и обрабатывает их самостоятельно.

Программный код посредника должен храниться в отдельном файле.

Первоначально посредник регистрируется в веб-обозревателе. Регистрация выполняется в самом начале загрузки страницы приложения. Сразу после регистрации посредник активизируется и начинает работать.

При регистрации посредник может выполнить какие-либо действия. Обычно он загружает ключевые файлы приложения и записывает их в кэш.

Перехватив очередной клиентский запрос на загрузку файла, отправленный фронтендом бэкенду, посредник ищет этот файл в кэше. Если такой файл существует, посредник автоматически формирует на его основе полноценный серверный ответ (не отличающийся от такового, полученного от бэкенда) и выдает его фронтенду. Если же запрашиваемого файла в кэше нет, посредник пересылает запрос бэкенду.

---

[1] В англоязычной литературе используется не очень корректный термин «service worker» (служебный обработчик). В русскоязычной литературе нет устоявшегося термина. Автор предпочитает использовать термин «посредник» как наиболее точно отражающий особенности подобного рода «промежуточных» программ.

Посредник функционирует все время, пока пользователь работает с приложением, которое его зарегистрировало. При переходе на другой интернет-адрес веб-обозреватель останавливает его работу, а по возвращении на зарегистрировавшее посредник приложение — запускает вновь. Благодаря этому посредник не мешает работе других приложений и сайтов и не отнимает зря системные ресурсы.

Зарегистрированный в веб-обозревателе посредник связывается с интернет-адресом страницы, которая его зарегистрировала. Поэтому веб-обозреватель «знает», когда его следует запустить.

Посредник работает независимо от фронтенда. При обновлении страницы приложения (например, нажатием клавиши <F5>) код посредника не загружается повторно с веб-сервера, и работа посредника не прерывается.

> **ВНИМАНИЕ!**
>
> Разные версии одного и того же прогрессивного веб-приложения, если они содержат разные посредники, следует хранить по разным интернет-адресам (например, по разным путям в пределах одного хоста — например: **https://somesite.ru/pwa1/** и **https://somesite.ru/pwa2/**). Благодаря этому каждая версия приложения будет использовать строго «свой» посредник.

18.1.1. Регистрация посредника

Объект класса `Navigator`, хранящийся в свойстве `navigator` текущего окна и представляющий саму программу веб-обозревателя, содержит свойство `serviceWorker`. Оно хранит объект класса `ServiceWorkerContainer`, который представляет подсистему веб-обозревателя, служащую для управления посредниками.

Загрузку файла с кодом посредника и его регистрацию запускает метод `register()` класса `ServiceWorkerContainer`:

```
register(<интернет-адрес файла с посредником>[, <параметры>=undefined])
```

Интернет-адрес, как правило, относительный, указывается в виде строки.

Параметры задаются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. В настоящее время поддерживается лишь параметр `scope`. Он задает путь, запросы по которому будут перехватываться создаваемым посредником, также станут перехватываться запросы по вложенным путям. Если этот параметр не указан, будут перехватываться лишь запросы по пути, по которому хранится сам файл посредника, и вложенным в него путям.

Метод `register()` возвращает промис, который подтверждается после успешной регистрации посредника и получает в качестве нагрузки объект класса `ServiceWorkerRegistration`, хранящий сведения о зарегистрированном посреднике и называемый *регистратором*.

Код, выполняющий регистрацию посредника, лучше помещать в самом начале HTML-кода страницы, даже перед прологом (тегом `<!doctype>`). В этом случае посредник будет зарегистрирован и начнет работу до того, как веб-обозреватель начнет загружать файлы с таблицами стилей, внешними веб-сценариями, изображе-

ниями, сможет перехватить запросы на их загрузку и выдать копии этих файлов, уже сохраненные им в кэше.

Примеры:

```
// Код страницы приложения /chat1/index.html
(async function () {
    // Этот посредник будет перехватывать лишь запросы по пути /chat1/
    // и вложенным в него путям
    await navigator.serviceWorker.register('sw.js');

    // Этот посредник будет перехватывать лишь запросы по пути
    // /chat1/assets/ и вложенным в него путям
    await navigator.serviceWorker.register('sw.js',
                                          { scope: '/chat1/assets/' });

    // Этот посредник будет перехватывать запросы по любому пути
    await navigator.serviceWorker.register('sw.js', { scope: './' });
})();
<!doctype html>
<html>
    . . .
</html>
```

Сам посредник представляется объектом класса `ServiceWorkerGlobalScope`, который доступен в коде посредника (а он, как говорилось ранее, сохраняется в отдельном файле), в автоматически создаваемой переменной `self`.

Код посредника выполняется в контексте («внутри») представляющего его объекта. Поэтому при обращении к свойствам и методам текущего посредника можно не указывать переменную `self`.

Сразу после запуска регистрации посредника в представляющем его объекте класса `ServiceWorkerGlobalScope` возникает событие `install`. Его обработчику в качестве параметра передается объект класса `ExtendableEvent`, представляющий сведения о событии.

Класс `ExtendableEvent` поддерживает метод `waitUntil(<промис>)`. Он приостанавливает процесс регистрации посредника (а также обработку других событий, которые будут возникать в нем) до тех пор, пока заданный *промис* не будет подтвержден.

Обычно этому методу в качестве параметра передают вызов какой-либо функции, которая и выполняет необходимые действия (например, записывает файлы в кэш). Процесс регистрации посредника будет приостановлен до тех пор, пока эта функция не завершит свою работу. По завершении работы функция возвращает подтвержденный промис, и метод `waitUntil()` возобновляет работу посредника.

По завершении регистрации посредника и перед его запуском в объекте посредника возникает событие `activate`. Его обработчику передается объект со сведениями о возникшем событии, который также принадлежит классу `ExtendableEvent`. Обыч-

но в обработчике события `activate` выполняется удаление устаревших файлов из кэша.

Примеры обработки событий `install`, `activate` и применения метода `waitUntil()` будут рассмотрены далее.

18.1.2. Перехват запроса и формирование ответа

Одно из основных назначений посредника — перехват клиентских запросов, отправляемых фронтендом, и, при необходимости, самостоятельное формирование в ответ на них готовых серверных ответов, обычно содержащих копии ключевых файлов, извлеченных из кэша.

Как только посредник перехватывает клиентский запрос, в его объекте возникает событие `fetch`. Его обработчику передается объект класса `FetchEvent`, хранящий сведения о возникшем событии.

Класс `FetchEvent` поддерживает полезные свойство и методы:

- `request` — свойство, доступно только для чтения, — объект класса `Request`, представляющий перехваченный клиентский запрос (будет в деталях описан позже);
- `waitUntil()` — был описан в *разд. 18.1.1*;
- `respondWith(<ответ>|<файл>|<промис>)` — формирует серверный ответ. В вызове этого метода можно указать:
  - непосредственно *ответ* в виде объекта класса `Response` (будет описан позже);
  - *файл*, извлеченный из кэша, — в этом случае веб-обозреватель сформирует на его основе серверный ответ автоматически;
  - *промис*, который после подтверждения должен получить в качестве нагрузки объект класса `Response` или файл из кэша.

  Обычно в вызов метода `respondWith()` подставляют функцию, возвращающую нужное значение.

18.1.2.1. Формирование произвольных ответов

Можно сформировать произвольный ответ, создав представляющий его объект класса `Response`. Конструктор этого класса вызывается в следующем формате:

```
Response(<содержание>[, <параметры>=undefined])
```

Содержание ответа можно указать в виде строки или объекта класса `Blob`. *Параметры* задаются в виде служебного объекта со свойствами, одноименными соответствующим им параметрам. Поддерживаются параметры:

- `status` — код статуса ответа в виде целого числа (по умолчанию — 200, т. е. запрос обработан успешно);
- `statusText` — строковое описание статуса ответа (по умолчанию — `'OK'`);

- `headers` — набор заголовков, добавляемых в ответ. Может быть задан в форматах, поддерживаемых методом `fetch()` (см. *разд. 6.1.1*).

Пример:

```
// Код посредника
self.addEventListener('fetch', (evt) => {
    evt.respondWith(
        new Response('Это прогрессивное веб-приложение!',
                     { status: 200, statusText: 'Все хорошо',
                       headers: { 'Content-Type': 'text/plain' }}));
});
```

18.1.2.2. Получение параметров запроса

Клиентский запрос представляется классом `Request` и извлекается из свойства `request` класса `FetchEvent`.

Класс `Request` поддерживает несколько полезных свойств, доступных только для чтения:

- `url` — интернет-адрес, по которому отправляется запрос, в виде строки;
- `method` — обозначение метода, которым был выполнен запрос, в виде строки, набранной в верхнем регистре (например: `'GET'`, `'POST'`);
- `headers` — заголовки запроса в виде объекта класса `Headers` (см. *разд. 6.1.1*)
- `cache` — обозначение режима кэширования ожидаемого ответа (см. *разд. 6.1.2*).

Метод `formData()` класса `Request` возвращает промис, который после подтверждения получает в качестве нагрузки объект класса `FormData`, представляющий данные, которые были отправлены из веб-формы.

18.1.3. Удаление посредника

Для удаления не нужного более посредника следует выполнить действия, перечисленные далее.

1. Получить объект с регистрационными сведениями об удаляемом посреднике — вызвав метод `getRegistration()` у объекта подсистемы посредников:

   `getRegistration(<интернет-адрес файла с посредником>)`

   Методу следует передать *`интернет-адрес`* удаляемого посредника в виде строки.

   Метод возвращает промис, после подтверждения получающий в качестве нагрузки объект класса `ServiceWorkerRegistration`, т. е. нужный нам объект-регистратор, или `undefined`, если посредника с указанным *`интернет-адресом`* зарегистрировано не было.

2. Если посредник с заданным *`интернет-адресом`* есть — удалить его, — вызвав метод `unregister()` у полученного ранее объекта регистратора.

Метод возвращает промис, после подтверждения получающий в качестве нагрузки значение `true`, если посредник был успешно удален, и `false` — в противном случае.

Обычно удаление посредника выполняется в коде новой версии прогрессивного приложения. Удаляется посредник, принадлежащий старой версии, а потом регистрируется посредник новой версии. Пример:

```
// Код страницы новой версии приложения /chat2/index.html
(async function () {
    const oSWC = navigator.serviceWorker;
    // Удаляем посредник старой версии приложения
    const oOldSWR = await oSWC.getRegistration('/chat1/sw.js');
    if (oOldSWR)
        await oOldSWR.unregister();
    // Регистрируем посредник новой версии
    await oSWC.register('sw2.js');
})();
```

18.2. Программируемый кэш

> *Программируемый кэш* — набор программных инструментов, предназначенных для работы с содержимым локального кэша (см. *разд. 6.1.2*): создания и удаления областей кэша, сохранения, выборки и удаления файлов.
>
> *Область кэша* — отдельное хранилище, программно создаваемое в локальном кэше и предназначенное для хранения файлов приложения. Должно иметь уникальное имя.

Разные области кэша могут хранить, например, файлы разных веб-приложений или разных версий одного и того же приложения. Количество создаваемых областей кэша не ограничено.

18.2.1. Получение доступа к области кэша

Класс `ServiceWorkerGlobalScope`, представляющий посредник, поддерживает свойство `caches`. Оно хранит объект класса `CacheStorage`, являющийся частью программируемого кэша и предоставляющий инструменты для работы с областями кэша.

Получить область кэша с заданным *именем* можно вызовом метода `open(<имя области кэша>)` объекта программируемого кэша. *Имя области кэша* указывается в виде строки. Если задано *имя* несуществующей области, эта область будет автоматически создана.

Метод возвращает промис, который в качестве нагрузки получит объект класса `Cache`, представляющий затребованную область кэша.

Пример:

```
self.addEventListener('install', (evt) => {
    evt.waitUntil(
```

```
        self.caches.open('chat-app-v1.0').then((oCO) => {
            // Выполняем какие-либо действия с полученной областью кэша
            // (например, сохраняем в ней файлы приложения) и не забываем
            // вернуть подтвержденный промис
            . . .
        })
    );
});
```

Класс `CacheStorage` поддерживает два полезных метода:

- `has(<имя области кэша>)` — возвращает промис, который получит в качестве нагрузки значение `true`, если область кэша с указанным именем существует, или `false` — в противном случае:

```
// Поскольку код посредника выполняется «внутри» представляющего его
// объекта, переменную self можно не указывать
caches.has('chat-app-v1.1').then((flag) => {
    if (flag) {
        // Область кэша chat-app-v1.1 существует
    } else {
        // Область кэша chat-app-v1.1 не существует
    }
});
```

- `keys()` — возвращает промис, получающий в качестве нагрузки массив с именами всех созданных к текущему моменту областей кэша:

```
caches.keys().then((aNames) => {
    . . .
});
```

18.2.2. Сохранение файлов в области кэша

Сохранять ключевые файлы приложения следует в обработчике события `install` посредника. Благодаря этому посредник в дальнейшем, перехватив первый же запрос, сможет извлечь запрашиваемый файл из кэша.

При сохранении файла в области кэша вместе с ним сохраняется его интернет-адрес и обозначение HTTP-метода, которым был выполнен запрос на загрузку этого файла.

Перед сохранением файлов следует получить область кэша, в которой они будут храниться (см. *разд. 18.2.1*).

Для добавления файлов в область кэша используются следующие методы класса `Cache`:

- `add(<интернет-адрес>)` — добавляет в текущую область кэша файл с указанным *интернет-адресом*, в качестве которого можно указать:

- собственно интернет-адрес в виде строки — в этом случае веб-обозреватель отправит бэкенду запрос на получение заданного файла, получит и сохранит файл в кэше;
- объект класса `Request`, представляющий запрос на получение сохраняемого файла, — тогда веб-обозреватель переправит запрос бэкенду, также получит файл и запишет его в кэш.

Если файл с заданным *интернет-адресом* уже хранится в текущей области кэша, он будет перезаписан.

Метод возвращает промис, который подтверждается после успешного сохранения файла и получает в качестве нагрузки значение `undefined`.

Пример:

```
self.addEventListener('install', (evt) => {
    evt.waitUntil(
        caches.open('chat-app-v1.0').then((oCO) => {
            return Promise.all([
                // Сохраняем в кэше страницу приложения index.html,
                // указав ее интернет-адрес — /chat1/
                oCO.add('/chat1/'),
                // Сохраняем в кэше остальные файлы приложения
                oCO.add('styles.css'),
                oCO.add('script.js'),
                oCO.add('images/background.jpg'),
                oCO.add('images/logo.png')
            ]);
        })
    );
});
```

- ♦ `addAll(<массив интернет-адресов>)` — добавляет в текущую область кэша все файлы с интернет-адресами из заданного *массива*. В остальном аналогичен методу `add()`. Пример:

```
evt.waitUntil(
    caches.open('chat-app-v1.0').then((oCO) => {
        return oCO.addAll(['/chat1/', 'styles.css', 'script.js',
                           'images/background.jpg',
                           'images/logo.png']);
    })
);
```

- ♦ `put()` — записывает в текущую область кэша файл, полученный в составе указанного *серверного ответа*, под заданным *интернет-адресом*:

```
put(<интернет-адрес>, <серверный ответ>)
```

Интернет-адрес представляется в тех же форматах, что и у метода `add()`, а *серверный ответ* должен быть представлен объектом класса `Response`.

Метод возвращает промис, который подтверждается после успешного сохранения файла и получает в качестве нагрузки значение `undefined`.

Код, загружающий и сохраняющий файл в кэше, может быть очень сложным, поэтому его удобнее оформить в виде асинхронного замыкания. Это замыкание ставится в качестве параметра в вызов метода `waitUntil()` объекта события и должно возвращать подтвержденный промис с произвольным значением (например, `undefined`).

Пример:

```
evt.waitUntil((async function () {
    const url = 'images/background.jpg';
    const oCO = await caches.open('chat-app-v1.0');
    oCO.put(url, await fetch(url));
    return Promise.resolve(undefined);
})());
```

Для простого сохранения файлов в кэше удобнее использовать методы `add()` и `addAll()`, описанные ранее. Метод `put()` следует применять лишь в специфических случаях (например, если по какой-то причине нужно загрузить файл с одного интернет-адреса, а в кэше сохранить под другим).

18.2.3. Извлечение файлов

Извлечение файлов из кэша выполняется в обработчике события `fetch` посредника, поскольку именно в нем обрабатываются клиентские запросы на загрузку файлов.

18.2.3.1. Извлечение файлов из области кэша

Для извлечения файлов из области кэша следует использовать один из следующих двух методов класса `Cache`:

- `match()` — ищет в текущей области кэша файл с заданным *интернет-адресом* с учетом указанных *параметров*:

  `match(<интернет-адрес>[, <параметры>=undefined])`

  Интернет-адрес может быть указан в виде строки или объекта класса `Request`, представляющего запрос на получение файла.

  Параметры указываются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. Поддерживаются параметры:

  - `ignoreSearch` — если `false`, при поиске файла в области кэша будет учитываться набор и значения присутствующих в интернет-адресе GET-параметров, если `true`, GET-параметры не будут учитываться (по умолчанию — `false`).

Например, если в кэше присутствуют три файла с интернет-адресами:

№ 1 — **http://www.somesite.ru/?page=1**

№ 2 — **http://www.somesite.ru/?search=javascript**

№ 3 — **http://www.somesite.ru/?page=2&search=php**

и выполняется извлечение файла с интернет-адресом:

http://www.somesite.ru/?search=javascript

по умолчанию будет извлечен файл № 2 (поскольку его интернет-адрес вместе с набором GET-параметров совпадает с искомым).

Если же указать веб-обозревателю не принимать во внимание GET-параметры, дав параметру `ignoreSearch` значение `true`, будет извлечен файл № 1 (поскольку его интернет-адрес без GET-параметров совпадает с искомым, также без GET-параметров);

- `ignoreMethod` — если `false`, при поиске файла в области кэша будет учитываться HTTP-метод, посредством которого был загружен файл, если `true`, HTTP-метод учитываться не будет (по умолчанию — `false`).

Метод возвращает промис, получающий в виде нагрузки объект класса `Response`, представляющий готовый, автоматически сформированный серверный ответ с найденным файлом, или `undefined`, если поиски файла в кэше не увенчались успехом.

Если поиски файла в кэше не увенчались успехом, этот файл следует загрузить с сервера и выдать клиенту, попутно сохранив в кэше. Однако серверный ответ можно либо выдать клиенту, либо сохранить в кэше — но не то и другое одновременно. Поэтому в кэше сохраняется копия серверного ответа, которую можно получить, вызвав у объекта ответа метод `clone()`.

Код, выдающий клиенту файл из кэша или с сервера, также может оказаться весьма сложным, и его имеет смысл оформить в виде асинхронного замыкания. Оно ставится в качестве параметра в вызов метода `respondWith()` объекта события и должно возвращать серверный ответ с запрашиваемым файлом.

Пример:

```
self.addEventListener('fetch', (evt) => {
    // Если перехваченный запрос был выполнен HTTP-методом GET
    // (т. е. выполняется загрузка файла), ищем запрашиваемый файл
    // в кэше
    if (evt.request.method == 'GET') {
        evt.respondWith((async function () {
            const oCO = await self.caches.open('chat-app-v1.0');
            const oR = await oCO.match(evt.request);
            if (oR)
                //  Если файл в кэше найден, выдаем его клиенту
                return oR;
```

```
            else {
                // В противном случае загружаем файл, сохраняем
                // в кэше копию содержащего его серверного ответа
                // и также выдаем клиенту
                const oR2 = await fetch(evt.request);
                await oCO.put(evt.request, oR2.clone());
                return oR2;
            }
        })());
    }
});
```

Если в кэше присутствует несколько копий искомого файла, записанных под разными интернет-адресами (пример возникновения такой ситуации был представлен ранее, при описании параметра `ignoreSearch`), метод `match()` выдаст самую первую найденную копию файла, а остальные проигнорирует;

- `matchAll()` — аналогичен `match()`, только выдает массив всех найденных копий искомого файла, сохраненных в кэше. Если подходящие файлы не найдены, выдает «пустой» массив.

В ряде случаев может пригодиться метод `keys()` класса `Cache`:

```
keys([<интернет-адрес>=undefined[, <параметры>=undefined]])
```

Он возвращает промис, получающий в качестве нагрузки массив клиентских запросов на загрузку всех файлов, которые хранятся в кэше под заданным *интернет-адресом*. *Интернет-адрес* и *параметры* указываются в тех же форматах, что и у метода `match()`. Клиентские запросы, содержащиеся в выданном методом массиве, представляются объектами класса `Request`.

18.2.3.2. Извлечение файлов из кэша

Класс `CacheStorage`, представляющий сам кэш, также поддерживает методы:

- `match()` — ищет файл с заданным интернет-адресом во всем кэше. Вызывается так же, как и одноименный метод класса `Cache` (см. *разд. 18.2.3.1*). Пример:

```
self.addEventListener('fetch', (evt) => {
    if (evt.request.method == 'GET') {
        evt.respondWith((async function () {
            const oR = await caches.match(evt.request);
            return oR || fetch(evt.request);
        })());
    }
});
```

Поддерживается дополнительный параметр `cacheName`, указывающий имя области кэша, в которой будет выполняться поиск файла, в виде строки. Если этот параметр не задан, поиск файла будет выполняться во всем кэше. Пример:

```
const oR2 = await caches.match(evt.request, { cacheName: ' chat-app-v1.0' });
```

- `keys()` — аналогичен одноименному методу класса `Cache`, но не позволяет указывать интернет-адрес и параметры.

18.2.4. Удаление файлов из области кэша

Для удаления из текущей области кэша файла с заданным *интернет-адресом* служит метод `delete()` класса `Cache`:

```
delete(<интернет-адрес>[, <параметры>=undefined])
```

Интернет-адрес и *параметры* задаются в том же формате, что и при вызове метода `match()` (см. *разд. 18.2.3.1*).

Метод возвращает промис, получающий в качестве нагрузки значение `true`, если файл был успешно удален, и `false` — в противном случае (например, если выполняется попытка удалить файл, отсутствующий в кэше).

Как правило, удаление файлов из кэша выполняется в обработчике события `activate` посредника, которое возникает перед запуском посредника. Пример:

```
self.addEventListener('activate', (evt) => {
    evt.waitUntil((async function () {
        const oCO = await caches.open('chat-app-v1.0');
        return oCO.delete('data.json');
    })());
});
```

После удаления файла из области кэша в объекте посредника возникает событие `contentdelete`.

18.2.5. Удаление области кэша

Удалить область кэша с заданным *именем* вместе со всеми хранящимися в ней файлами можно вызовом метода `delete(<имя области кэша>)` класса `CacheStorage`. Метод возвращает промис, получающий в качестве нагрузки значение `true`, если область кэша была успешно удалена, и `false` — в противном случае (например, если выполняется попытка удалить несуществующую область кэша). Пример:

```
self.addEventListener('activate', (evt) => {
    evt.waitUntil((async function () {
        // Удаляем область кэша, хранящую файлы устаревшей версии
        // приложения
        return caches.delete('chat-app-v0.9');
    })());
});
```

18.3. Упражнение. Добавляем посредник в клиент веб-чата

Добавим в клиент службы веб-чата (см. *упражнение 15.3*) посредник, который сохранит в кэше ключевые файлы клиента и будет выдавать их при перехвате запросов на их загрузку.

Код посредника сохраним в файле sw.js. Он сохранит в области кэша `chat-app-v1.0` файлы index.html, styles.css и script.js.

1. Найдем в папке 15\ex15.3 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css и script.js. Скопируем их куда-либо на локальный диск.
2. Найдем в той же папке 15\ex15.3 папку Workerman и файл server.php, содержащие код сервера службы. Также скопируем их куда-либо на локальный диск.
3. Откроем в текстовом редакторе копию файла index.html и вставим в самое начало HTML-кода страницы веб-сценарий, регистрирующий наш посредник:

```
<script type="text/javascript">
    (async function () {
        await navigator.serviceWorker.register('sw.js');
    })();
</script>
<!doctype html>
<html>
    . . .
</html>
```

Настала пора создать файл sw.js, в котором будет храниться код посредника. Поскольку мы неоднократно будем обращаться к области кэша `chat-app-v1.0`, сохраним ее имя в константе `cacheName`.

4. Создадим в той же папке, где хранятся файлы index.html, styles.css и script.js, файл sw.js, откроем его в текстовом редакторе и запишем в него выражение, объявляющее константу `cacheName` с именем области кэша:

```
const cacheName = 'chat-app-v1.0';
```

При регистрации посредника следует сохранить в кэше файлы, упомянутые ранее.

5. Добавим обработчик события `install` посредника, сохраняющий файлы приложения в кэше:

```
self.addEventListener('install', (evt) => {
    evt.waitUntil(
        caches.open(cacheName).then((oCO) => {
            return oCO.addAll(['/chat1/', 'styles.css', 'script.js']);
        })
    );
});
```

6. Добавим обработчик события `fetch` посредника, выдающий файл при перехвате очередного клиентского запроса:

```
self.addEventListener('fetch', (evt) => {
    if (evt.request.method == 'GET') {
        evt.respondWith((async function () {
            const oCO = await caches.open(cacheName);
            const oR = await oCO.match(evt.request);
            if (oR)
                return oR;
            else {
                const oR2 = await fetch(evt.request);
                await oCO.put(evt.request, oR2.clone());
                return oR2;
            }
        })());
    }
});
```

7. Откроем командную строку, перейдем в папку, в которой находится сервер чата, и запустим его с помощью команды, описанной в *разд. 15.2.4*.
8. Создадим в корневой папке веб-сервера папку chat1, скопируем в нее файлы index.html, styles.css, script.js, sw.js и запустим веб-сервер. Откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/chat1/**. Войдем в службу под каким-либо именем.
9. Откроем другую вкладку того же веб-обозревателя и выполним вход в чат от имени другого пользователя. Проверим, как работает пересылка сообщений.

Напоследок выйдем из чата и завершим работу его сервера.

18.4. Задачи посредника

> *Задача посредника* — задача, периодически выполняемая посредником в состоянии простоя.

Задачи посредника аналогичны фоновым задачам (см. *разд. 8.1*), только они выполняются посредником, а не фронтендом. Они будут работать только в том случае, если приложение содержит зарегистрированный посредник (поскольку код, реализующий эти задачи, записывается в составе кода посредника).

Задачи посредника можно использовать для периодической загрузки и сохранения в кэше каких-либо часто изменяющихся файлов.

ВНИМАНИЕ!

Задачи посредника успешно выполняются только в *уже установленных* PWA (об установке прогрессивных приложений будет рассказано на *уроке 20*). Если приложение еще не установлено, ни одну задачу даже не удастся зарегистрировать.

В настоящее время задачи посредника не поддерживаются Mozilla Firefox.

18.4.1. Регистрация задач посредника

Чтобы зарегистрировать новую задачу посредника, следует выполнить действия, перечисленные далее.

1. Получить регистратор — обратившись к свойству `ready` объекта подсистемы PWA (как говорилось ранее, этот объект класса `ServiceWorkerContainer` хранится в свойстве `serviceWorker` объекта `navigator`).

   Свойство `ready` выдаст промис, который после подтверждения получит в качестве нагрузки объект класса `ServiceWorkerRegistration`, представляющий регистратор.

2. Получить объект подсистемы задач посредника — обращением к свойству `periodicSync` объекта регистратора.

   Подсистема задач посредника представляется объектом класса `PeriodicSyncManager`.

3. Зарегистрировать задачу посредника — вызовом метода `register()` объекта подсистемы задач посредника:

   ```
   register(<имя задачи посредника>[, <параметры>=undefined])
   ```

   Имя должно быть уникальным в пределах текущего приложения и задается в виде строки. *Параметры* указываются служебным объектом со свойствами, одноименными с соответствующими им параметрами. В настоящее время поддерживается лишь параметр `minInterval`, задающий минимальный промежуток времени между запусками регистрируемой задачи на выполнение в виде целого числа в миллисекундах.

   > **Внимание!**
   > Задаваемый при регистрации минимальный промежуток времени между запусками задачи является лишь рекомендацией веб-обозревателю, которой он может и не последовать. Так, если веб-обозреватель посчитает, что задан слишком маленький промежуток времени, а само приложение находится в активном режиме, он может не запустить задачу в очередной момент.

   Метод `register()` возвращает промис, который подтверждается после регистрации задачи посредника и в качестве нагрузки получает значение `undefined`.

Пример регистрации задачи посредника `get-data`, выполняемой каждые 12 часов:

```
(async function () {
    try {
        const oSWR = await navigator.serviceWorker.ready;
        await oSWR.periodicSync.register('get-data',
                                  { minInterval: 12 * 60 * 60 * 1000 });
    }
    catch(exc) {
        // Задачу зарегистрировать не удалось, поскольку либо приложение
        // еще не установлено, либо веб-обозреватель не поддерживает
        // задачи посредника
    }
})();
```

Класс `PeriodicSyncManager` поддерживает метод `getTags()`, возвращающий промис, который после подтверждения получает в качестве нагрузки массив из имен всех задач посредника, которые были зарегистрированы к настоящему времени. Пример:

```
const aSWTNames = await oSWR.getTags();
if (aSWTNames.includes('get-data')) {
    // Задача посредника get-data уже зарегистрирована
}
```

18.4.2. Выполнение задач посредника

Как только наступает время выполнить очередную задачу посредника, в объекте посредника возникает событие `periodicsync`. Его обработчику в качестве параметра передается объект класса `PeriodicSyncEvent`, хранящий сведения о событии.

Класс `PeriodicSyncEvent` поддерживает доступное только для чтения свойство `tag`, содержащее имя задачи, которая должна быть выполнена.

Еще этот класс поддерживает метод `waitUntil()`, описанный в *разд. 18.1.1* (собственно, класс `PeriodicSyncEvent` является производным от класса `ExtendableEvent`, от него он и получает метод `waitUntil()`). В вызове этого метода ставится функция, которая и реализует выполняемую задачу.

Пример выполнения задачи `get-data`:

```
self.addEventListener('periodicsync', (evt) => {
    if (evt.tag == 'get-data')
        evt.waitUntil((async function () {
            const oCO = await caches.open(cacheName);
            return oCO.add('data.json');
        })());
});
```

18.4.3. Удаление задач посредника

Если требуется удалить не нужную более задачу посредника с заданным *именем*, поможет метод `unregister(<имя задачи посредника>)` класса `PeriodicSyncManager`.

Пример удаления задачи `get-data`:

```
await oSWR.unregister('get-data');
```

18.5. Отладочные инструменты для работы с посредниками и кэшем

В составе отладочных инструментов, встроенных в веб-обозреватель, присутствуют средства для работы с посредниками, областями кэша и задачами посредника.

Чтобы добраться до этих средств, следует вывести панель с отладочными инструментами, нажав клавишу <F12>, и переключиться на вкладку **Application** этой

панели. Сами средства представлены в иерархическом списке, расположенном в левой части панели, в виде отдельных пунктов.

18.5.1. Инструменты для работы с посредниками

Для переключения на инструменты, предназначенные для работы с посредниками, следует выбрать в области **Application** иерархического списка пункт **Service Workers**. В правой части панели появится перечень всех посредников, зарегистрированных открытым в веб-обозревателе приложением, и некоторые элементы управления, часть которых будет нам полезна (рис. 18.1).

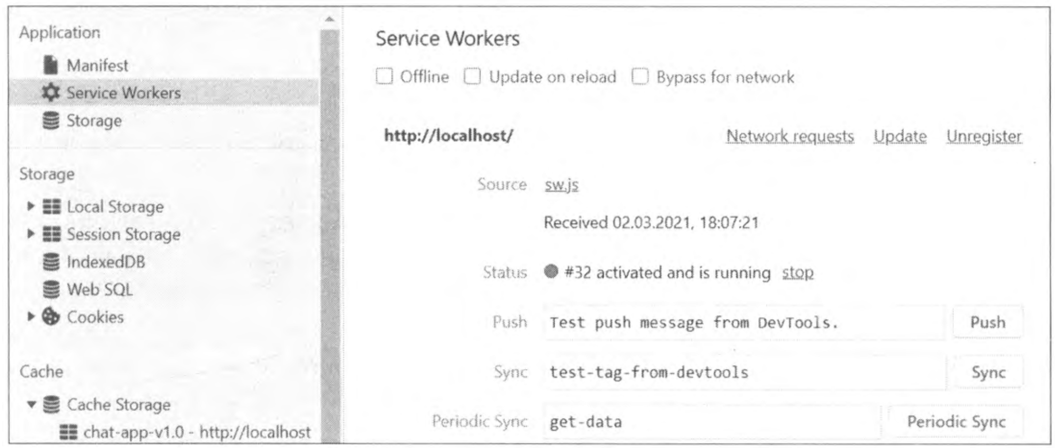


Рис. 18.1. Перечень зарегистрированных посредников

Сведения о каждом посреднике выводятся в отдельном блоке, в порядке сверху вниз:

- полужирным шрифтом — интернет-адрес, с которого был загружен посредник.

  Правее интернет-адреса находятся три гиперссылки для управления посредником, две из которых нам пригодятся;
- **Source** — имя файла, хранящего код посредника;
- **Received** — дата и время регистрации;
- **Status** — состояние, в том числе: порядковый номер, который увеличивается при каждой регистрации посредника, и описание состояния (если посредник зарегистрирован, активирован и работает — **activated and is running**).

  Ниже расположены три поля ввода с кнопками, одно из которых пригодится при отладке задач посредника (см. далее).

Над посредником можно выполнить следующие манипуляции:

- принудительно перезагрузить с сервера (что может понадобиться после правки программного кода посредника — в этом случае порядковый номер, поставленный в графе **Status**, будет увеличен) — щелкнув на гиперссылке **Update** правее интернет-адреса;

- удалить из веб-обозревателя (если в процессе отладки потребуется выполнить регистрацию посредника повторно) — щелкнув на гиперссылке **Unregister** правее интернет-адреса;
- приостановить его работу — щелкнув на гиперссылке **stop** в строке **Status**, правее текстового описания состояния. После этого описание состояния сменится на **activated and is stopped** (активирован и остановлен), а гиперссылка **stop** — на **start**;
- возобновить работу ранее приостановленного посредника — щелкнув на гиперссылке **start** в строке **Status**, правее текстового описания состояния. После этого описание состояния сменится на **activated and is running** (активирован и работает), а гиперссылка **start** — на **stop**.

В самом верху панели со сведениями о посредниках находятся три флажка, которые могут пригодиться в специфических случаях:

- **Offline** — отключает посредники от сети, предписывая им загружать все запрашиваемые фронтендом файлы только из кэша;
- **Update on reload** — указывает веб-обозревателю при каждом обновлении страницы приложения повторно загружать с сервера код посредника. Пожалуй, при отладке посредников всегда следует держать этот флажок установленным;
- **Bypass for network** — указывает веб-обозревателю отсылать все клиентские запросы напрямую веб-серверу, минуя посредник. Пригодится, если нужно проверить, как приложение работает без посредника.

18.5.2. Инструменты для работы с программируемым кэшем

Эти инструменты представлены ветвью **Cache Storage**, находящейся в области **Cache** левого иерархического списка. В эту ветвь вложены пункты, представляющие отдельные области кэша и имеющие названия формата:

<имя области кэша, заданное при его создании>* - *<интернет-адрес приложения>

При выборе любой области кэша в правой части панели будет выведено ее содержимое (рис. 18.2).

Правая часть панели содержит два раздела. В верхнем разделе показывается перечень сохраненных в области кэша файлов, имеющий вид таблицы со столбцами:

- **#** — порядковый номер, под которым файл был записан в кэш;
- **Name** — путь к файлу относительно корневой папки сайта.

  Если навести курсор мыши на какую-либо строку таблицы, появится всплывающая подсказка с полным интернет-адресом представленного в ней файла;
- **Response-Type** — всегда **basic** (файл был успешно загружен). Другие значения выводятся в крайне специфических случаях;
- **Content-Type** — MIME-тип;

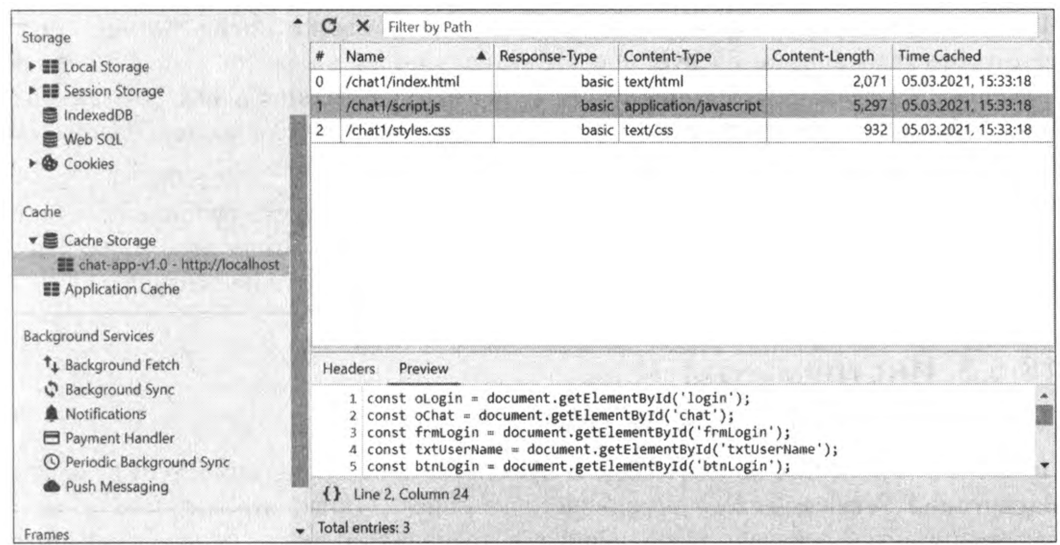


Рис. 18.2. Содержимое выбранной области кэша

- **Content-Length** — размер файла в байтах;
- **Time Cached** — дата и время занесения в кэш.

Можно менять размеры отдельных столбцов таблицы, перетаскивая мышью разделяющие их линии. Чтобы вернуть размеры столбцов по умолчанию, следует щелкнуть правой кнопкой мыши на таблице и выбрать в появившемся контекстном меню пункт **Header Options | Reset Columns**.

Можно отсортировать файлы в таблице по значению какого-либо из столбцов, щелкнув на заголовке этого столбца. Повторный щелчок на заголовке того же столбца меняет порядок сортировки на противоположный. Также можно выбрать столбец для сортировки в подменю **Sort By** контекстного меню.

Чтобы обновить содержимое таблицы файлов, следует щелкнуть на кнопке **Refresh** (**C**), расположенной в панели инструментов над таблицей, или выбрать в контекстном меню пункт **Refresh**.

В той же панели инструментов находится поле ввода. Если ввести в него какое-либо значение, в таблице будут показаны только файлы, пути к которым начинаются с введенного значения.

С выбранным в таблице файлом можно выполнить следующие действия:

- просмотреть содержимое — в нижнем разделе правой части панели, на вкладке **Preview**. Поддерживается вывод веб-страниц (к сожалению, текст в кодировке UTF-8 отображается некорректно), содержимого таблиц стилей и сценариев (так, на рис. 18.2 отображается содержимое файла script.js) и изображений.

  Вкладка **Headers** нижнего раздела выводит заголовки серверного ответа, в котором был получен выбранный файл;
- удалить — щелкнув на кнопке **Delete Selected** (**✕**) в панели инструментов или выбрав одноименный пункт контекстного меню.

Чтобы обновить список областей кэша, выводимый в ветви **Cache Storage** левого иерархического списка, следует щелкнуть на этой ветви правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Refresh Caches**. Хотя, скорее всего, делать это придется нечасто — обычно веб-обозреватель после создания или удаления области кэша обновляет список областей автоматически.

Удалить область кэша целиком можно, щелкнув на представляющем ее пункте в ветви **Cache Storage** правой кнопкой мыши и выбрав в контекстном меню пункт **Delete**. Следует помнить, что область кэша удаляется без предупреждения.

18.5.3. Инструменты для работы с задачами посредника

Переключиться на них можно выбором пункта **Periodic Background Sync** в области **Background Services** левого иерархического списка. После этого в правой части панели появится перечень выполненных к текущему моменту задач посредника, изначально пустой (рис. 18.3).

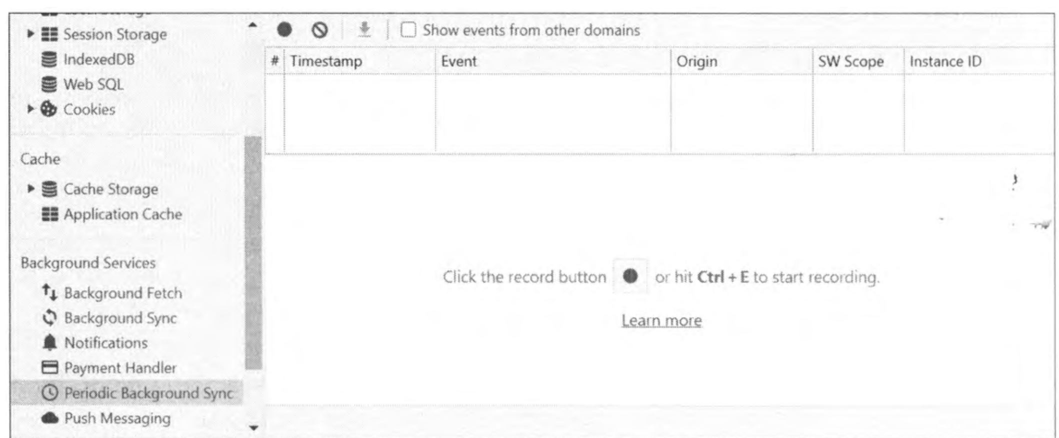


Рис. 18.3. Пустой перечень выполненных задач посредника

Нужно уяснить, что сведения о выполненных задачах не заносятся в этот список автоматически. Чтобы задачи регистрировались в списке, нужно включить режим записи, для чего достаточно выполнить одно из следующих действий:

- щелкнуть на черной кнопке **Start recording events** (●), которая присутствует как над перечнем файлов, в небольшой панели инструментов, так и под ним, в составе надписи **Click the record button**;
- нажать комбинацию клавиш <Ctrl>+<E>.

После этого под списком появится текст, сообщающий о том, что режим записи включен, а в панели инструментов кнопка **Start recording events** сменится красной кнопкой **Stop recording events** (●).

Внимание!

Режим записи, будучи включенным, продолжит работу даже после закрытия отладочных инструментов. Отключить его можно только явно.

Чтобы не дожидаться, когда веб-обозреватель запустит очередную задачу посредника, можно вызвать ее запуск принудительно. Для этого достаточно переключиться на инструменты для работы с посредниками (см. *разд. 18.5.1*), найти блок, где выводятся сведения о посреднике, в котором реализована нужная задача, ввести ее имя в поле ввода **Periodic Sync** и нажать расположенную правее одноименную кнопку (см. рис. 18.1).

#	Timestamp	Event	Origin	SW Scope	Instance ID
1.	2021-03-03 11:26:01.451	Dispatched periodicsync event	http://localhost/	/	get-data

Рис. 18.4. Сведения о выполненной задаче посредника, выведенные в перечне

Как только задача посредника выполнится, строка со сведениями о ней появится в перечне (рис. 18.4), который имеет вид таблицы со столбцами:

- # — порядковый номер выполненной задачи;
- **Timestamp** — дата и время выполнения;
- **Event** — описание события, в обработчике которого была выполнена задача. Всегда имеет вид: **Dispatched periodicsync event** (Обработанное событие `periodicsync`);
- **Origin** — интернет-адрес хоста, с которого был загружен посредник, реализующий выполненную задачу;
- **SW Scope** — интернет-адрес, запросы по которому перехватывает посредник и который был указан в параметре `scope` при регистрации этого посредника (см. *разд. 18.1.1*);
- **Instance ID** — имя выполненной задачи посредника.

Можно менять размеры отдельных столбцов таблицы, перетаскивая мышью разделяющие их линии. Чтобы вернуть размеры столбцов по умолчанию, следует выбрать пункт **Header Options | Reset Columns** контекстного меню.

Чтобы выключить режим записи, следует выполнить одно из следующих действий:

- щелкнуть на красной кнопке **Stop recording events** (●) в панели инструментов;
- еще раз нажать комбинацию клавиш <Ctrl>+<E>.

Внимание!

В перечне хранятся только задачи посредника, выполненные в течение последних трех суток. Сведения о более «старых» задачах удаляются.

Очистить перечень выполненных задач посредника можно, щелкнув на кнопке **Clear** (⊘) в панели инструментов.

18.5.4. Удаление данных PWA

С помощью отладочных инструментов можно удалить все данные, которые сохранило приложение. Это может понадобиться при повторной регистрации посредников в процессе отладки.

Для удаления данных нужно выбрать пункт **Storage** в области **Application** левого иерархического списка и прокрутить содержимое правой части панели вниз. Там присутствуют флажки для выбора удаляемых данных, разбитые на группы (рис. 18.5):

- **Application**:
  - **Unregister service workers** — все посредники, зарегистрированные приложением;
- **Storage**:
  - **Local and session storage** — локальное и сессионное хранилища;
  - **Indexed DB** — базы данных;
- **Cache**:
  - **Cache storage** — области кэша.

Все эти флажки изначально установлены, так что по умолчанию будет удалено все. Если что-то требуется оставить, достаточно сбросить соответствующий флажок.

Собственно удаление данных выполняется нажатием кнопки **Clear site data**.

Вверху правой части панели указано, какой объем имеют данные, сохраненные приложением, и присутствует диаграмма, показывающая, какую часть этих данных занимают посредники, области кэша и др.

Рис. 18.5. Элементы управления для выбора удаляемых данных

18.6. Самостоятельное упражнение

Добавьте посредник в клиент усовершенствованной службы чата (см. *упражнение 15.4* в папке 15\s15.4 электронного архива), превратив его во вторую версию прогрессивного веб-приложения. Для хранения этого приложения создайте папку chat2 в корневой папке веб-сервера.

Посредник, входящий в состав новой версии приложения, должен сохранять ключевые файлы в области кэша `chat-app-v2.0`.

Перед регистрацией нового посредника удалите посредник старой версии. За основу можете взять код, приведенный в *разд. 18.1.3*.

Также новый посредник должен удалять область кэша `chat-app-v1.0`, в которой хранились файлы старой версии приложения. Удаление старой области кэша выполните в обработчике события `activate` нового посредника (подробности — в *разд. 18.2.5*).

Урок 19
Всплывающие оповещения

Всплывающее оповещение

Активация клиента

Чтобы уведомить пользователя о каком-либо событии, прогрессивное веб-приложение может вывести всплывающее оповещение.

> *Всплывающее оповещение* — небольшое окно, содержащее текстовое уведомление о каком-либо событии, которое произошло в приложении (например, о получении нового сообщения чата).

Всплывающее оповещение выводится средствами операционной системы (не веб-обозревателя) и демонстрируется в течение нескольких секунд, после чего автоматически скрывается. Обычно оно содержит заголовок, сам текст уведомления и небольшое графическое изображение (как правило, логотип приложения).

19.1. Работа со всплывающими оповещениями

19.1.1. Запрос разрешения на вывод всплывающих оповещений

Перед выводом первого всплывающего оповещения следует проверить, дал ли ранее пользователь разрешение на вывод оповещений, и, если это не так, запросить такое разрешение.

Проверить, дал ли пользователь разрешение, можно, обратившись к статическому, доступному только для чтения свойству `permission` класса `Notification`, который представляет оповещение. Свойство хранит одно из следующих строковых значений:

- `'granted'` — если пользователь дал разрешение;
- `'denied'` — если пользователь запретил вывод оповещений;
- `'default'` — если разрешение на вывод оповещений еще не выводилось.

Если разрешение еще не выводилось, его нужно вывести. Сделать это можно вызовом статического метода `requestPermission()` класса `Notification`.

При вызове этого метода на экране появляется окно-предупреждение с запросом на вывод оповещений (рис. 19.1). Нажатие кнопки **Разрешить** даст разрешение на вывод оповещений, нажатие кнопки **Блокировать** запретит их вывод.

Рис. 19.1. Окно-предупреждение, запрашивающее разрешение на вывод всплывающих оповещений

Метод `requestPermission()` возвращает промис, который после подтверждения в качестве нагрузки получает строку с обозначением выбора пользователя (см. описание свойства `permission`). Если этот результат не нужен, можно проигнорировать возвращаемый промис.

Разрешение следует запрашивать после какого-либо действия, выполненного пользователем (например, нажатия им определенной кнопки). Запрашивать его непосредственно после загрузки страницы не рекомендуется, поскольку это может показаться пользователю слишком назойливым, вдобавок, будущие версии веб-обозревателей, возможно, будут блокировать такие запросы.

Пример:

```
<input type="button" id="btnPermission" value="Запросить разрешение">
. . .
btnPermission.addEventListener('click', async function () {
    if (Notification.permission == 'default')
        await Notification.requestPermission();
});
```

19.1.2. Вывод всплывающих оповещений

Всплывающее оповещение может быть выведено как фронтендом, так и посредником (о которых рассказывалось в *разд. 18.1*).

19.1.2.1. Вывод всплывающих оповещений фронтендом

Чтобы вывести всплывающее оповещение в коде фронтенда, достаточно создать объект класса `Notification`, представляющий выводимое оповещение. Конструктор этого класса вызывается в следующем формате:

```
Notification(<заголовок оповещения>[, <параметры>=undefined])
```

Заголовок оповещения указывается в виде строки. Необязательные *параметры* должны быть заданы в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. В настоящее время поддерживаются следующие параметры:

- `body` — текст оповещения, выводимый под заголовком;
- `icon` — интернет-адрес изображения размерами 128×128 пикселов, которое будет выводиться в качестве значка оповещения;

- `image` — интернет-адрес изображения, которое будет выведено в составе оповещения над заголовком. Такое изображение может иметь произвольные размеры, поскольку перед выводом масштабируется, чтобы поместиться на оповещении;
- `tag` — тег оповещения, который указывается в виде строки и выбирается произвольно.

  Если последовательно вывести два всплывающих оповещения с одинаковыми тегами, будет выведено лишь второе оповещение, а первое скроется. Если же задать у выводимых оповещений разные теги, будут выведены оба;
- `badge` — интернет-адрес изображения, применяемого для представления оповещения, если на экране недостаточно места для вывода полного оповещения;
- `data` — произвольные данные, которые могут быть любого типа;
- `requireInteraction` — если `false`, оповещение будет автоматически скрыто спустя определенное время. Если `true`, оповещение будет постоянно присутствовать на экране, и пользователю, чтобы убрать его, понадобится нажать кнопку **Закрыть** на оповещении. Значение по умолчанию — `false`;
- `renotify` — используется только при последовательном выводе нескольких оповещений с одинаковыми тегами. Если `false`, более позднее оповещение заменит собой более ранее, не уведомляя пользователя, если `true`, пользователь будет уведомлен об этом. Значение по умолчанию — `false`;
- `silent` — если `false`, при выводе оповещения будет выдаваться сигнал, установленный в настройках операционной системы (обычно звук и, на мобильных устройствах, вибрация). Если `true`, сигнал выдаваться не будет. Значение по умолчанию — `false`.

> ***Внимание!***
>
> В настоящее время параметры `image`, `badge`, `requireInteraction`, `renotify` и `silent` игнорируются Mozilla Firefox.

Примеры:

```
if (Notification.permission == 'granted')
    const oNotif = new Notification('Веб-чат',
                                    { body: 'Вам пришло новое сообщение',
                                      icon: '/notification.png',
                                      tag: 'chat' });

const oNotif2 = new Notification('Веб-чат',
                                 { body: 'Пришел новый пользователь',
                                   icon: '/notification.png',
                                   tag: 'chat-new-user',
                                   data: newUserName });
```

19.1.2.2. Вывод всплывающих оповещений посредником

В коде посредника класс `Notification` недоступен. Для вывода всплывающих оповещений в посреднике следует выполнить шаги, перечисленные далее.

1. Получить объект регистратора — обратившись к свойству `registration` объекта посредника.
2. Создать оповещение — вызовом метода `showNotification()` объекта регистратора. Этот метод имеет тот же формат вызова, что и конструктор класса `Notification`, и поддерживает те же параметры (см. *разд. 19.1.2.1*).

   Метод возвращает промис, который подтверждается после вывода оповещения и получает в качестве нагрузки значение `undefined`.

Пример:

```
self.addEventListener('fetch', (evt) => {
    evt.respondWith((async function () {
        const oR = await fetch(evt.request);
        if (Notification.permission == 'granted')
            self.registration
                .showNotification('Загрузчик больших файлов',
                                  { body: 'Большой файл загружен',
                                    icon: '/loader.png',
                                    data: evt.request.url,
                                    tag: 'file-load' });
        return oR;
    })());
});
```

19.1.3. Управление всплывающими оповещениями

19.1.3.1. Управление всплывающими сообщениями во фронтендах

Объект класса `Notification`, представляющий всплывающее оповещение, поддерживает ряд полезных событий:

- `show` — возникает после вывода всплывающего оповещения;
- `click` — возникает, когда пользователь щелкает мышью на всплывающем оповещении;
- `close` — возникает после закрытия оповещения пользователем;
- `error` — возникает, если оповещение не может быть выведено в случае какой-либо программной ошибки.

> ***Внимание!***
>
> В настоящее время событие `error` не поддерживается Mozilla Firefox.

Еще класс `Notification` поддерживает ряд полезных свойств, доступных только для чтения:

- `title` — заголовок всплывающего оповещения;
- `body`, `icon`, `image`, `tag`, `badge`, `data`, `requireInteraction`, `renotify` и `silent` — соответствуют одноименным параметрам, указываемым в вызове конструктора класса (см. *разд. 19.1.2.1*).

Пример:

```
oNotif2.addEventListener('show', (evt) => {
    const oN = evt.target;
    // Если это оповещение сообщает о появлении в службе чата
    // нового пользователя...
    if (oN.tag == 'chat-new-user') {
        // ...получаем имя нового пользователя...
        const newUserName = oN.data;
        // ...и используем его в работе (например, выводим на странице)
        . . .
    }
});
```

Получить сам объект оповещения в коде фронтенда, чтобы, например, привязать к нему обработчик события, не составляет труда — этот объект возвращается конструктором класса `Notification` при вызове.

19.1.3.2. Управление всплывающими сообщениями в посредниках

Получить объект оповещения в коде посредника несколько сложнее. Для этого следует вызвать метод `getNotifications([<`*параметры*`>=undefined])` объекта регистратора. Он вернет промис, который после подтверждения получит в качестве нагрузки массив всех объектов оповещений, созданных к настоящему моменту в текущем приложении.

Необязательные *параметры* указываются в виде служебного объекта со свойствами, одноименными с соответствующими им параметрами. В настоящее время поддерживается лишь параметр `tag`, задающий тег, который должны содержать выбираемые оповещения.

Пример:

```
// Код посредника
(async function () {
    const aNs = await self.registration
                          .getNotifications({ tag: 'file-load' });
    for (let oN of aNs) {
        const url = oN.data;
        . . .
    }
})();
```

Объект посредника поддерживает событие `notificationclose`, возникающее после закрытия всплывающего оповещения пользователем. Его обработчику в качестве параметра передается объект класса `NotificationEvent`, хранящий сведения о событии. Этот класс поддерживает свойство `notification`, хранящее объект закрытого оповещения, и описанный в *разд. 18.1.1* метод `waitUntil()`.

Пример:

```
self.addEventListener('notificationclose', (evt) => {
    const oN = evt.notification;
    if (oN.tag == 'file-load') {
        . . .
    }
});
```

19.1.4. Активация клиента

> *Активация клиента* — вывод на передний план окна приложения, создавшего всплывающее оповещение, при щелчке на этом оповещении.

Благодаря реализованной в приложении активации клиента, пользователь сможет быстро переключиться на это приложение, щелкнув на оповещении.

19.1.4.1. Активация клиента в коде фронтенда

Реализовать активацию клиента в коде фронтенда очень просто — достаточно соответствующим образом обработать событие `click` оповещения. Пример:

```
oNotif.addEventListener('click', () => {
    // Делаем вкладку веб-обозревателя, в которой открыто
    // приложение, активной
    window.focus();
});
```

19.1.4.2. Активация клиента в коде посредника

Активация клиента в коде посредника реализуется несколько сложнее.

Она выполняется в коде обработчика события `notificationclick` посредника, которое возникает после щелчка на оповещении. Обработчику этого события передается объект класса `NotificationEvent`, описанного в *разд. 19.1.3.2*.

В коде обработчика упомянутого события следует получить список всех клиентов, обслуживаемых текущим посредником. К клиентам относятся фронтенды, открытые во вкладках веб-обозревателя, фоновые потоки и др. Но нас сейчас интересуют лишь фронтенды.

Свойство `clients` посредника хранит объект класса `Clients`, представляющий список всех имеющихся у посредника клиентов. Чтобы извлечь из этого объекта пригодный для обработки массив фронтендов, следует вызвать у него метод `matchAll()`. Метод вернет промис, который после подтверждения получит в качестве нагрузки массив фронтендов.

К сожалению, полученный массив не является последовательностью и не может быть обработан в цикле перебора последовательности (см. *разд. 3.1*). Так что придется применять цикл со счетчиком.

Отдельный фронтенд, присутствующий в полученном массиве, представляется объектом класса `WindowClient`. Этот класс поддерживает следующие полезные свойства, доступных только для чтения:

- `url` — интернет-адрес, с которого был загружен клиент, в виде строки;
- `focused` — `true`, если текущий клиент активен, `false` — в противном случае.

Еще класс `WindowClient` поддерживает следующие методы:

- `focus()` — активизирует окно, в котором открыт текущий клиент. Возвращает промис, подтверждаемый после активизации окна и получающий в качестве нагрузки текущий объект клиента;
- `navigate(<интернет-адрес>)` — выполняет в окне, в котором открыт текущий клиент, навигацию по заданному в виде строки *интернет-адресу*. Возвращает тот же результат, что и метод `focus()`.

Наконец, класс `Clients` поддерживает метод `openWindow(<интернет-адрес>)`, который открывает новую вкладку веб-обозревателя и выполняет в нем навигацию по заданному *интернет-адресу*. Он возвращает тот же результат, что и метод `focus()` класса `Client`.

Пример:

```
// Интернет-адрес приложения
const appURL = 'https://somesite.ru/chat/';

self.addEventListener('notificationclick', (evt) => {
    evt.waitUntil((async function () {
        const aClients = await clients.matchAll();
        for (let i = 0; i < aClients.length; i++) {
            let oClient = aClients[i];
            if ((oClient.url == appURL) && (!oClient.focused))
                return oClient.focus();
        }
        return clients.openWindow(appURL);
    })());
});
```

19.1.5. Удаление всплывающих оповещений

Удалить всплывающее оповещение можно вызовом метода `close()` у объекта нужного оповещения. При этом оповещение будет удалено и с экрана, и из истории созданных оповещений, которая ведется операционной системой. Пример:

```
oNotif2.close();
```

19.2. Упражнение. Реализуем вывод всплывающих оповещений у клиента веб-чата

Реализуем в клиенте службы веб-чата (см. *упражнение 18.3*) вывод всплывающего оповещения при получении очередного нового сообщения.

Разрешение на вывод оповещений будем запрашивать при нажатии кнопки **Войти** на экране входа.

Каждое оповещение будет содержать заголовок **Веб-чат**, текст **Пришло сообщение от пользователя <*имя*>** и значок. При щелчке на оповещении будет активизироваться вкладка с клиентом чата.

1. Найдем в папке 18\ex18.3\chat1 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css, script.js и sw.js. Скопируем их куда-либо на локальный диск.
2. Найдем в папке 18\ex18.3\chat1 папку Workerman и файл server.php, содержащие код сервера службы. Также скопируем их куда-либо на локальный диск.
3. Найдем в папке 19\!sources электронного архива файл notification.png (значок для оповещений) и скопируем его в ту же папку, в которую ранее скопировали файлы index.html, styles.css, script.js и sw.js.

   Когда пользователь входит в службу чата, следует спросить у него, хочет ли он получать всплывающие оповещения. Соответствующий код следует добавить в код, выполняющий вход.
4. Откроем в текстовом редакторе файл script.js и добавим в обработчик события `submit` веб-формы `frmLogin` код, запрашивающий разрешение на вывод оповещений:

```
frmLogin.addEventListener('submit', (evt) => {
    . . .
    oWS.addEventListener('message', tryLoginAnswer);
    if (Notification.permission == 'default')
        Notification.requestPermission();
});
```

   Поскольку нам не нужен результат, выдаваемый методом `requestPermission()`, мы проигнорируем возвращаемый им промис и не будем ждать его подтверждения.

   Для вывода оповещения с заданным *текстом* объявим функцию `showNotification(<текст>)`. Заголовок и значок у всех оповещений будут одинаковыми.

   К создаваемому оповещению следует привязать обработчики событий `click` и `close`. Обработчиком события `click` станет функция `activateClient()`, выполняющая активацию клиента. Обрабатывать событие `close` будет функция `cleanupNotif()` — она уберет у закрытого оповещения привязанные ранее обра-

ботчики событий, чтобы предотвратить их случайное выполнение после закрытия оповещения.

5. Добавим объявления функций `showNotification()`, `activateClient()` и `cleanupNotif()`:

```
function showNotification(body) {
    if (Notification.permission == 'granted') {
        const oNotif = new Notification('Веб-чат',
                                        { body: body,
                                          icon: '/notification.png' });
        oNotif.addEventListener('click', activateClient);
        oNotif.addEventListener('close', cleanupNotif);
    }
}

function activateClient() {
    window.focus();
}

function cleanupNotif(evt) {
    evt.target.removeEventListener('click', activateClient);
    evt.target.removeEventListener('close', cleanupNotif);
}
```

Выводить оповещения о новом сообщении следует при получении от сервера службы сигнала `message`.

6. Добавим в объявление функции `getMessage()` код, выводящий оповещение о получении нового сообщения:

```
function getMessage(evt) {
    const oM = JSON.parse(evt.data);
    switch (oM.type) {
        case 'message':
            showMessage(oM);
            const s = 'Пришло сообщение от пользователя ' +
                      oM.userName;
            showNotification(s);
            break;
        . . .
    }
}
```

7. Откроем командную строку, перейдем в папку, в которой находится сервер чата, и запустим его с помощью команды, описанной в *разд. 15.2.4*.

8. Создадим в корневой папке веб-сервера папку chat1, скопируем в нее файлы index.html, styles.css, script.js, sw.js, notification.png и запустим веб-сервер. Откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/chat1/**. Выпол-

ним вход в службу под каким-либо именем и, как только появится окно с запросом разрешения на вывод всплывающих уведомлений, разрешим их.

9. Откроем другую вкладку того же веб-обозревателя и выполним вход в чат от имени другого пользователя. Перешлем сообщение первому пользователю и проверим, выводит ли клиент сообщение о его получении. Щелкнем на сообщении, чтобы активизировать клиент.

После чего выйдем из чата и завершим работу его сервера.

19.3. Самостоятельное упражнение

Реализуйте в клиенте службы веб-чата (см. *упражнение 19.2*) вывод всплывающих оповещений с уведомлениями о входе нового пользователя в чат и выходе из него.

Урок 20
Манифест PWA. Установка веб-приложений

Манифест PWA
Установка PWA

> *Манифест PWA* — файл, хранящий сведения о прогрессивном веб-приложении и являющийся признаком того, что текущая веб-страница, собственно, и является приложением.

Загрузив манифест, веб-обозреватель «поймет», что соответствующая ему страница суть приложение, и предложит пользователю установить его и создать на рабочем столе ярлык, указывающий на это приложение.

20.1. Манифест PWA

Манифест PWA записывается в формате JSON. Файл с манифестом может иметь произвольное имя (например, manifest.json).

20.1.1. Формат манифеста PWA

Манифест PWA должен иметь следующие свойства:

- `name` — имя приложения в виде строки;
- `icons` — массив значков приложения, используемых для создания ярлыка, указывающего на приложение. Должен включать значки с одинаковым изображением, но с разными размерами — для вывода на экранах с разной разрешающей способностью.

  Каждый элемент массива должен быть объектом со следующими свойствами:

  - `src` — путь к файлу со значком. Можно указать путь:
    - абсолютный (отсчитываемый от корневой папки сайта);
    - относительный — который будет отсчитываться от папки, в которой хранится файл манифеста;
  - `sizes` — размеры значка в виде строки. Можно указать:
    - один размер в формате `<ширина>x<высота>`, где *ширина* и *высота* представлены целыми числами и измеряются в пикселах, — если файл хранит один значок определенного размера:

      ```
      "sizes": "144x144"
      ```

  - несколько размеров в описанном ранее формате, разделенные пробелами, — если файл хранит несколько значков разных размеров:

    ```
    "sizes": "64x64 128x128 144x144 256x256"
    ```

- `type` — MIME-тип файла со значком в виде строки.

  Полный набор значков для настольных компьютеров и мобильной операционной системы Google Android должен включать значки с размерами 72×72, 96×96, 128×128, 144×144, 152×152, 192×192, 384×384 и 512×512 пикселов, полный набор значков для системы Apple iOS — 120×120 и 180×180.

  На практике часто указывают лишь значки с размерами 120×120 и 144×144 пиксела. Значки остальных размеров веб-обозреватель при необходимости создаст сам, путем масштабирования имеющихся значков (однако, возможно, с потерей качества);

- `start_url` — интернет-адрес страницы приложения, который будет записан в ярлык. Может быть указан в виде полного интернет-адреса, абсолютного или относительного пути, в последнем случае путь отсчитывается от папки, в которой хранится файл манифеста. Примеры:

  ```
  "start_url": "/pages/start_page.html"

  // Так указывается текущий интернет-адрес
  "start_url": "./"
  ```

  Остальные свойства не обязательны для указания:

- `description` — краткое описание приложения в виде строки;
- `short_name` — краткое имя приложения, выводящееся на экран, если на нем не хватает места для вывода полного имени (задаваемого в свойстве `name`);
- `scope` — путь к папке, из которой приложение сможет загружать файлы, также оно будет иметь доступ к файлам, хранящимся во вложенных папках. Если приложение попытается открыть файл, расположенный в какой-либо иной папке, веб-обозреватель заблокирует загрузку.

  Можно указать абсолютный или относительный путь, в последнем случае путь отсчитывается от папки, в которой хранится файл манифеста. Пример:

  ```
  // Приложение может загружать только файлы, расположенные в папке
  // appfiles, которая вложена в папку с манифестом, и во вложенных
  // в нее папках
  "scope": "appfiles/"
  ```

  Если свойство не указано, приложение может загружать файлы только из папки, в которой хранится манифест приложения, и вложенных в нее папок;

- `display` — обозначение режима отображения в виде строки. Поддерживаются следующие режимы:
  - `'browser'` — приложение выводится в отдельной вкладке веб-обозревателя, как обычная страница;

- `'minimal-ui'` — приложение выводится веб-обозревателем в отдельном окне. Это окно будет содержать ограниченный набор стандартных элементов управления веб-обозревателя: кнопки управления навигацией, строку состояния и др. Конкретный набор выводимых элементов управления зависит от веб-обозревателя;
- `'standalone'` — то же самое, что и `'minimal-ui'`, только набор выводимых элементов управления еще более ограничен. В частности, не будут выведены кнопки управления навигацией;
- `'fullscreen'` — приложение займет весь экран. Никаких дополнительных элементов управления выводиться не будет.

Если свойство не указано, используется режим `'browser'`;

- ♦ `orientation` — ориентация мобильного устройства, требуемая для работы приложения (на традиционных компьютерах игнорируется), в виде одного из следующих строковых обозначений:
  - `'any'` — любая ориентация;
  - `'portrait'` — портретная (вертикальная);
  - `'landscape'` — ландшафтная (горизонтальная).

  Если свойство не указано, используется ориентация `'any'`;
- ♦ `background_color` — цвет, которым веб-обозреватель закрашивает фон страницы приложения перед тем, как будут загружены привязанные к странице таблицы стилей. Может быть указан в любом формате, поддерживаемом атрибутом стиля `background-color`. Примеры:

```
"background-color": "lightgrey"
"background-color": "#FFDD78"
"background-color": "rgba(127, 63, 203, 0.5)"
```

  Если свойство не указано, используется цвет фона по умолчанию, зависимый от конкретного веб-обозревателя;
- ♦ `theme_color` — цвет темы приложения. Этим цветом в диспетчере задач Android закрашивается позиция, представляющая приложение. Задается в тех же форматах, что и цвет фона (свойство `background_color`).

Пример манифеста PWA будет рассмотрен далее, при выполнении практического упражнения.

20.1.2. Привязка манифеста к веб-странице PWA

Для привязки манифеста к странице приложения используется одинарный тег `<link>`, располагаемый в секции заголовка страницы (в теге `<head>`). В атрибуте `href` тега `<link>` указывается интернет-адрес файла с манифестом, а в атрибуте `rel` — значение `manifest`. Пример:

```
<link rel="manifest" href="manifest.json">
```

20.2. Установка PWA

Процесс установки прогрессивных веб-приложений на мобильных и настольных операционных системах несколько различается.

20.2.1. Установка PWA на мобильных платформах

На мобильных платформах процесс установки PWA выполняется в три шага:

1. Веб-обозреватель, загрузив и прочитав файл с манифестом, «понимает», что текущая страница — на самом деле страница прогрессивного приложения.
2. Веб-обозреватель выводит предложение установить это приложение — при условии, что пользователь открывал текущую страницу, как минимум, дважды и провел на ней не менее 5 минут.

   Предложение обычно имеет вид полоски с поясняющим текстом, именем приложения (которое берется из свойства `name` манифеста) и гиперссылкой, нажатие на которую запускает установку. Выводится оно обычно внизу экрана. Точный вид предложения зависит от операционной системы.
3. Если пользователь нажмет на гиперссылку, находящуюся на предложении, — веб-обозреватель устанавливает приложение и помещает на рабочий стол указывающий на него ярлык.

   Также установить приложение можно из меню веб-обозревателя. Например, в Chrome для Android для этого следует выбрать пункт **Добавить на главный экран**.

Для запуска установленного приложения следует нажать созданный в процессе установки ярлык.

Удалить установленное прогрессивное приложение можно так же, как и обычное.

20.2.2. Установка PWA на настольных платформах

Считается, что прогрессивные приложения разрабатываются, в первую очередь, для мобильных платформ. Поэтому веб-обозреватель, работающий на мобильной платформе, должен *самостоятельно* предлагать пользователю установить приложение.

Напротив, на настольных платформах PWA будут, опять же, согласно существующим представлениям, использоваться существенно реже. Следовательно, настольные веб-обозреватели должны предлагать пользователю установить приложение только по его, пользователя, желанию.

Для реализации установки прогрессивного приложения на настольной платформе необходимо выполнить следующие действия.

1. Подавить вывод предложения установки сразу после открытия его страницы.

   Непосредственно перед выводом предложения, если приложение не было установлено ранее, в объекте текущего окна веб-обозревателя возникает собы-

тие `beforeinstallprompt`. Его обработчику передается объект класса `BeforeInstallPromptEvent`, хранящий сведения о событии.

В обработчике этого события необходимо:

- собственно подавить вывод предложения — вызвав у объекта события метод `preventDefault()`;
- сохранить объект события в какой-либо глобальной переменной, поскольку он понадобится в дальнейшем для вывода предложения.

2. В ответ на какое-либо действие пользователя (например, щелчок на специально созданной на странице гиперссылке или кнопке) — вывести предложение установки.

   Предложение выводится вызовом метода `prompt()` класса `BeforeInstallPromptEvent` и имеет вид, представленный на рис. 20.1. Приложение будет установлено после нажатия кнопки **Установить**. Кнопка **Отмена** отменяет установку.

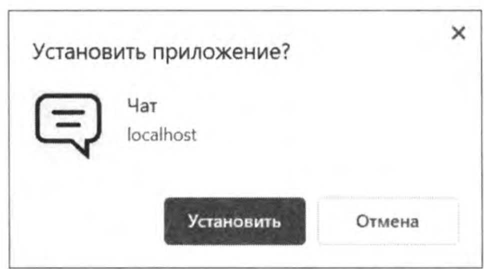


Рис. 20.1. Предложение установки PWA, выводимое на настольной платформе

> ***Внимание!***
>
> Событие `beforeinstallprompt` возникает также и на мобильных платформах. Поэтому всегда следует проверять, на какой платформе открыта страница приложения, и выполнять описанные ранее действия только на настольных платформах.

Пример реализации установки приложения будет рассмотрен далее, при выполнении упражнения.

Выяснить, разрешил ли пользователь установку приложения, можно, обратившись к свойству `userChoice` класса `BeforeInstallPromptEvent`. Оно выдаст промис, который после подтверждения получит в качестве нагрузки служебный объект со свойством `outcome`, которое будет хранить строку `'accepted'`, если пользователь выбрал установку приложения, и `'dismissed'`, если он отказался от этого.

После установки приложения на рабочем столе также появится ярлык, который при щелчке запускает приложение.

Удалить установленное приложение можно, запустив его, найдя в заголовке его окна кнопку с тремя точками, расположенными по вертикали (рис. 20.2), щелкнув на этой кнопке и выбрав в появившемся системном меню пункт **Удалить приложение "<*имя приложения*>"**.

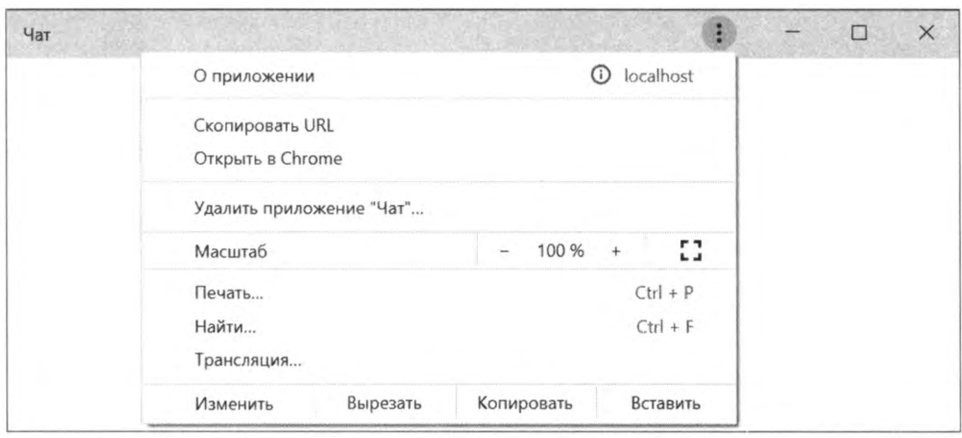


Рис. 20.2. Системное меню прогрессивного веб-приложения

> ***Внимание!***
>
> В настоящее время установку прогрессивных веб-приложений на настольных платформах поддерживает только Google Chrome для Windows.

20.3. Упражнение. Веб-чат: пишем манифест и реализуем установку

Добавим в клиент веб-чата (см. *упражнение 19.3*) манифест PWA и возможность установки на рабочий стол в настольных платформах.

Установка приложения будет выполняться после щелчка на гиперссылке **Установить**, которая будет выводиться на экране входа под веб-формой.

1. Найдем в папке 19\ex19.2\chat1 сопровождающего книгу электронного архива (см. *приложение*) файлы index.html, styles.css, script.js, sw.js и notification.png. Скопируем их куда-либо на локальный диск.

   Сервер чата нам не понадобится, т. к. мы не будем проверять работу службы в полном объеме. Нам понадобится лишь проверить, нормально ли устанавливается приложение.

2. Найдем в папке 20\!sources электронного архива файлы chat120x120.png и chat144x144.png (значки приложения размеров 120×120 и 144×144 пикселов соответственно) и скопируем их в ту же папку, в которую ранее скопировали файлы index.html, styles.css, script.js, sw.js и notification.png.

   Начнем с создания манифеста, который сохраним в файле manifest.json. Дадим приложению имя **Веб-чат**, укажем ему работать в режиме `standalone`, в портретной ориентации, в качестве цветов фона и темы укажем белый. Не забудем занести в него оба ранее скопированных значка.

3. Создадим в той же папке, где хранятся ранее скопированные файлы, файл manifest.json, откроем его в текстовом редакторе и сохраним в нем код манифеста:

```
{
    "name": "Веб-чат",
    "icons": [
        {"src": "chat120x120.png", "sizes": "120x120",
         "type": "image/png"},
        {"src": "chat144x144.png", "sizes": "144x144",
         "type": "image/png"}],
    "start_url": "./",
    "description": "Веб-чат, работающий через WebSocket",
    "display": "standalone",
    "orientation": "portrait",
    "background_color": "#fff",
    "theme_color": "#fff"
}
```

4. Откроем в текстовом редакторе файл index.html и вставим тег, привязывающий только что созданный манифест к странице приложения:

```
<!doctype html>
<html>
    <head>
        . . .
        <link rel="manifest" href="manifest.json">
    </head>
    . . .
</html>
```

Займемся реализацией установки. Под веб-формой входа `frmLogin` поместим абзац с гиперссылкой **Установить**. Абзацу дадим якорь `install_text` и сделаем его изначально скрытым, вставив в его тег соответствующий встроенный стиль. Гиперссылке дадим якорь `install`. Абзац `install_text` отделим от веб-формы `frmLogin` «пустым» абзацем.

5. Вставим под веб-формой входа HTML-код, создающий абзац с гиперссылкой, которая установит приложение:

```
<form id="frmLogin">
    . . .
</form>
<p> </p>
<p id="install_text" style="display: none;">
    <a id="install" href="#">Установить</a>
</p>
```

6. Откроем в текстовом редакторе файл script.js и добавим выражения, получающие доступ к добавленным ранее абзацу и гиперссылке:

```
const oInstallText = document.getElementById('install_text');
const oInstall = document.getElementById('install');
```

7. Добавим выражение, объявляющее переменную `oEvent` для хранения объекта события `beforeinstallprompt`:

   ```
   let oEvent;
   ```

   Предпринимать действия, необходимые для реализации установки на настольной платформе и описанные в *разд. 20.2.2*, следует, если страница приложения открывается на настольной платформе. Поэтому предварительно нужно выяснить, на какой платформе открывается страница. Это можно сделать, получив из свойства `userAgent` объекта `navigator` строку с описанием текущего веб-обозревателя и проверив, присутствуют ли в ней строки `'android'`, `'iphone'`, `'ipad'` или `'ipod'`. Если какая-либо из указанных строк там присутствует, значит, страница открыта на мобильном устройстве.

   Проверку платформы, на которой открывается страница, реализуем в функции `isMobileDevice()`. Если страница открыта на мобильной платформе, функция вернет `true`, в противном случае — `false`. Для упрощения программирования в теле функции применим регулярное выражение.

8. Добавим объявление функции `isMobileDevice()`:

   ```
   function isMobileDevice() {
       const rex = /android|iphone|ipad|ipod/i;
       return navigator.userAgent.match(rex);
   }
   ```

   Как только веб-обозреватель «решит» вывести предложение установить текущее веб-приложение, если текущая платформа является настольной, нужно подавить вывод предложения, сохранить объект события в объявленной ранее переменной `oEvent` и вывести абзац `install_text` с гиперссылкой **Установить**. Для проверки платформы используем только что объявленную функцию `isMobileDevice()`.

9. Добавим обработчик события `beforeinstallprompt`, выполняющий эти действия:

   ```
   window.addEventListener('beforeinstallprompt', (evt) => {
       if (!isMobileDevice()) {
           evt.preventDefault();
           oEvent = evt;
           oInstallText.style.display = 'block';
       }
   });
   ```

   Если пользователь решит установить приложение, он щелкнет на гиперссылке `install` (**Установить**). В таком случае следует проверить, действительно ли в переменной `oEvent` сохранен объект события (во избежание потенциальных ошибок), и, если это так, вывести предложение установки, удалить объект события и скрыть абзац `install_text`.

10. Добавим обработчик события `click` гиперссылки `install`, который выполнит все это:

```
oInstall.addEventListener('click', (evt) => {
    evt.preventDefault();
    if (oEvent) {
        oEvent.prompt();
        oEvent = undefined;
        oInstallText.style.display = 'none';
    }
});
```

11. Создадим в корневой папке веб-сервера папку chat1, скопируем в нее файлы index.html, styles.css, script.js, sw.js, notification.png, manifest.json, chat120x120.png, chat144x144.png и запустим веб-сервер. Откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost/chat1/**. Как только под веб-формой входа появится гиперссылка **Установить**, щелкнем на ней и согласимся на установку приложения.

    После установки приложения на рабочем столе появится его ярлык, а веб-обозреватель запустит установленное приложение в отдельном окне.

12. Удалим приложение (как это сделать, было описано в *разд. 20.2.2*) и закроем его окно.

20.4. Отладочные инструменты для работы с манифестом PWA

Среди отладочных инструментов веб-обозревателя есть средства для просмотра манифеста PWA текущего приложения. Чтобы открыть их, нужно вывести панель с отладочными инструментами, нажав клавишу <F12>, переключиться на вкладку **Application** этой панели и выбрать пункт **Manifest** в области **Application** расположенного слева иерархического списка.

В правой части панели будет выведено содержимое манифеста. Сначала отобразятся строковые сведения, взятые из одноименных свойств манифеста: имя приложения, его интернет-адрес, режим работы, требуемая ориентация устройства и цвета приложения (рис. 20.3). Далее будут выведены значки: первым, с подзаголовком **Primary icon**, покажется значок, в текущий момент используемый веб-обозревателем для представления приложения, а далее будут показаны все значки, описанные в манифесте, с указанием их размеров и MIME-типов (рис. 20.4).

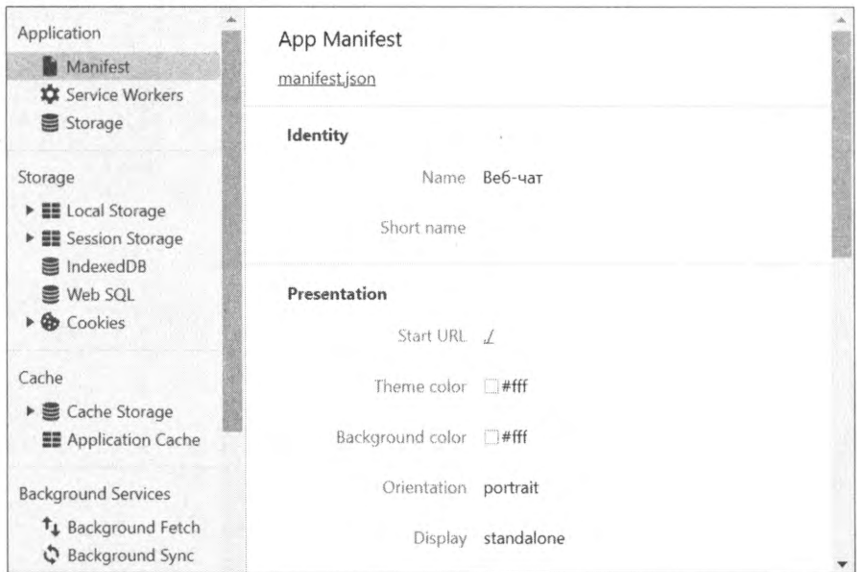


Рис. 20.3. Содержимое манифеста PWA: строковые сведения

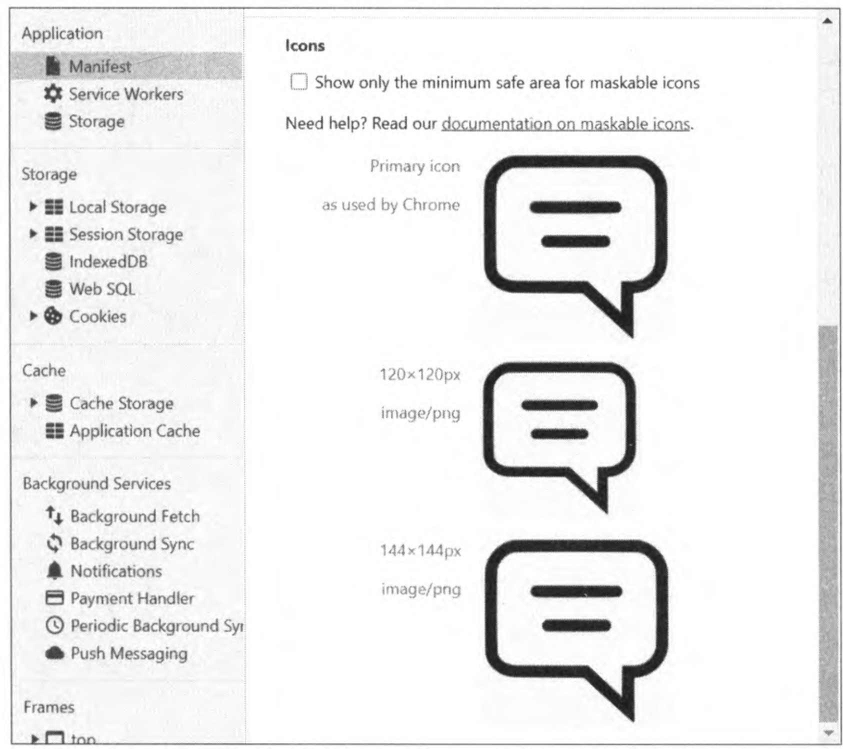


Рис. 20.4. Содержимое манифеста PWA: значки

Заключение

Вот и подошли к концу 20 дополнительных уроков по языку программирования JavaScript.

- Мы изучили новые впечатляющие программные инструменты, предлагаемые современными веб-обозревателями: встроенную СУБД, средства для захвата и кодирования видео, обработки и визуализации звука, поддержки WebSocket и WebRTC, программирования прогрессивных веб-приложений, а также HTML-компоненты и новые возможности JavaScript.
- Мы выполнили множество практических упражнений, попутно написав, в том числе, веб-приложение для съемки видео, телефонный справочник, службы веб-чата и интернет-видеотелефонии.
- Мы существенно повысили свой программистский уровень — с начинающего до среднего — и обогатили свой программистский опыт.
- И теперь с уверенностью можем претендовать на новое место работы — безусловно, с более высокой зарплатой и, возможно, на более высокой должности.

Бóльшая часть описанных в книге программных инструментов основаны на вполне устоявшихся интернет-стандартах, которые в дальнейшем вряд ли будут меняться. Так что соответствующие знания и навыки, полученные при прочтении этой книги, пригодятся нам и в будущем.

Однако некоторые программные интерфейсы из числа рассмотренных в книге основаны на черновых редакциях стандартов, которые наверняка будут дополнены и изменены позже, возможно даже в значительной степени.

Кроме того, часть веб-обозревателей (а именно Google Chrome и Microsoft Edge Chromium, которые в этом смысле, что называется, впереди планеты всей) уже получили поддержку ряда новых, перспективных, но еще в значительной степени «сырых» технологий. В их числе: средства для аутентификации на веб-сайтах, оплаты в интернет-магазинах, доступа к локальной файловой системе, взаимодействия между отдельными страницами, поддержки Bluetooth, виртуальной, расширенной реальности и пр. Это весьма впечатляющие и наверняка востребованные в будущем инструменты, но по причине их «сырости» они в этой книге не описывались.

Узнать все об этих инструментах и быть в курсе всех последних изменений в их реализации позволят следующие интернет-ресурсы:

- **https://developer.mozilla.org/en-US/docs/Web** — подборка самых актуальных руководств по различным веб-технологиям. Руководства сгруппированы по те-

мам: HTML-компоненты, мультимедиа, PWA и др. Также присутствуют гиперссылки на руководства и справочники по HTML, CSS и JavaScript;

- **https://developer.mozilla.org/en-US/docs/Web/API** — справочник по программным интерфейсам веб-обозревателей. Организован по алфавитному принципу;
- **https://developer.mozilla.org/en-US/docs/Web/JavaScript** — исчерпывающее руководство и справочник по JavaScript, включающий все последние нововведения, появившиеся в этом языке.

Портал, объединяющий все эти ресурсы, создан и поддерживается командой разработчиков веб-обозревателя Mozilla Firefox. Так что все представленные в нем материалы — что называется, из первых рук.

Внимание!

Этот портал включает русскоязычный раздел, однако он очень неполный. Поэтому автор рекомендует пользоваться англоязычным разделом.

На этом автор прощается с вами, уважаемые читатели! До свидания и успехов на ниве JavaScript-программирования!

Владимир Дронов

Приложение
Описание электронного архива

Электронный архив, сопровождающий книгу, выложен на FTP-сервер издательства «БХВ» по интернет-адресу: **ftp://ftp.bhv.ru/9785977567817.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

Список папок и файлов, имеющихся в архиве (вложенность папок и файлов показана отступами):

- <*номер урока*> — папка с материалами урока с указанным *номером*. Состав папки:
  - !sources — папка с исходными материалами, необходимыми для выполнения упражнений в текущем уроке;
  - ex<*номер раздела*> — результаты выполнения упражнения, приведенного в разделе с указанным *номером*;
  - s<*номер раздела*> — результаты выполнения самостоятельных упражнений, приведенных в разделе с указанным *номером*;
- readme.txt — текстовый файл с описанием электронного архива.

Предметный указатель

A

abort 111
abort() 110
AbortError 173
activate 303
add() 109, 307
addAll() 308
addIceCandudate() 265
addTrack() 261
advance() 117
AggregateError 47
all() 46
allSettled() 46
AnalyserNode 219
any() 47
append() 91
array_keys() 247
assignedSlot 166
async 53, 55
attachShadow() 147
attributeChangedCallback() 155
audioBitsPerSecond 177
AudioContext 207
AudioDestinationNode 209
AudioNode 209
AudioParam 211
autoIncrement 106
await 54

B

badge 327
beforeinstallprompt 338
BeforeInstallPromptEvent 338
binaryType 233
BiquadFilterNode 212
Blob 176
blob() 95
body 327
bound() 114
boundingClientRect 85
bufferedAmount 233

C

cache 305
Cache 306
caches 306
CacheStorage 306
cancelAnimationFrame() 138
cancelIdleCallback() 137
candidate 264
catch() 44
channel 291
class 18
click 327
clients 329
clone() 310
cloneNode() 160
close 236, 292, 327
close() 123, 218, 236, 242, 266, 292, 330
CloseEvent 236
code 236
commit() 110
complete 110
connect() 209
connectedCallback() 156
connections 239
ConstraintError 104, 110
content 160
contentdelete 312
continue() 117
continuePrimaryKey() 117
count 239
count() 120
createAnalyser() 219
createAnswer() 263
createBiquadFilter() 212
createDataChannel() 290
createElement() 151
createGain() 212
createIndex() 106
createMediaElementSource() 208
createMediaStreamDestination() 210
createMediaStreamSource() 208
createObjectStore() 104
createObjectURL() 176
createOffer() 262
createStereoPanner() 211
currentTime 218
CustomElementRegistry 150
customElements 150

D

data 175, 327
dataavailable 175
databases() 103
datachannel 292
DataError 110
db 109
define() 150
delete() 92, 122, 312
deleteDatabase() 103
deleteIndex() 107
deleteObjectStore() 105
destination 209
destroy() 242
detune 214
didTimeout 137
direction 117
disconnect() 84, 217
disconnectedCallback() 156
DocumentFragment 160
DOM 146
◊ скрытая 146

E

entries() 92
error 102, 103, 111, 177, 236, 327
errors 47
export 72
export default 72
ExtendableEvent 303
extends 23

F

fetch 304
fetch() 89, 94
FetchEvent 304
fftSize 221
fillLightMode 202
finally() 44
Float32Array 220
focus() 330
focused 330
formData() 305
frequency 213
frequencyBinCount 220
function* 62

G

gain 212, 214
GainNode 212
get() 91, 113, 114, 115, 158
getAll() 119
getAllKeys() 119
getAttribute() 88, 148
getAudioTracks() 203

getByteFrequencyData() 222
getByteTimeDomainData() 221
getDisplayMedia() 187, 192
getFloatFrequencyData() 222
getFloatTimeDomainData() 220
getKey() 114, 115
getNotifications() 328
getOutputTimestamp() 218
getPhotoSettings() 204
getRegistration() 305
getSettings() 203
getSupportedConstraints() 197
getTags() 316
getTracks() 197, 203
getUserMedia() 171, 192
getVideoTracks() 190, 203

H

has() 91, 307
headers 94, 305
Headers 91
HTML API 82
HTMLSlotElement 166
HTMLTemplateElement 160
HTML-компонент 145
◊ автономный 145

I

ICE 259
◊ сервер 260
icecandidate 264
icon 327
id 242, 292
IDBCursor 118
IDBCursorWithValue 117
IDBDatabase 102
IDBFactory 101
IDBIndex 106
IDBKeyRange 114, 115
IDBObjectStore 105
IDBOpenDBRequest 101
IDBTransaction 109
IDBVersionChangeEvent 102
IdleDeadline 137
image 327
ImageCapture 190
ImageCapture() 190
imageHeight 202
imageWidth 202
includes() 116
index() 113
indexedDB 101
indexNames 107
install 303
IntersectionObserver 82
IntersectionObserverEntry 85
intersectionRatio 85
intersectionRect 85
InvalidStateError 105
is 151
isConnected 156
isIntersecting 85
isTypeSupported() 177
iterator 57

J

json() 95
json_decode() 247

K

key 117
keyPath 106, 107
keys() 92, 307, 311, 312
kind 203

L

label 292
localDescription 262
logFile 239
lower 115
lowerBound() 115
lowerOpen 116

M

match() 309, 311
matchAll() 311, 329
max 202
maxDecibels 221
maxPackageSize 242
maxSendBufferSize 242
mediaDevices 171
MediaDevices 171
mediaElement 208
MediaElementAudioSourceNode 208
MediaRecorder 174
MediaSettingsRange 202
mediaStream 208
MediaStream 172
MediaStreamAudioDestinationNode 210
MediaStreamAudioSourceNode 208
MediaStreamTrack 190
message 235, 291
MessageEvent 235
method 305
mimeType 177
min 202
minDecibels 221
mode 109
multiEntry 107

N

name 104, 106, 107, 162, 239
navigate() 330
newVersion 102
NotAllowedError 173
NotFoundError 105, 109
notification 328
Notification 324, 325
notificationclick 329
notificationclose 328
NotificationEvent 328
NotReadableError 173
NotSupportedError 150

O

objectStore() 109
objectStoreNames 105, 109
observe() 84
observedAttibutes 155
ok 94
oldVersion 102
onClose 238
onConnection 238
onError 238
only() 115
onMessage 238
onWorkerStart 238
onWorkerStop 238
open 232, 291
open() 101, 306
openCursor() 116
openKeyCursor() 118
openWindow() 330
ordered 292
OverconstrainedError 173

P

pan 211
part 157
pause 177
pause() 177
periodicsync 316
periodicSync 315
PeriodicSyncEvent 316
PeriodicSyncManager 315
permission 324
PhotoCapabilities 202
primaryKey 117
promise 48
Promise 42
PromiseRejectionEvent 48
prompt() 338
put() 120, 308
PWA 298

Q

Q 213

R

race() 47
ReadOnlyError 110
ready 315

readyState 233
reason 48, 236
redEyeReduction 202
register() 302, 315
registration 327
reject() 48
rejectionhandled 48
renotify 327
request 304
Request 94, 304, 305
requestAnimationFrame() 138
requestIdleCallback() 136
requestPermission() 324
requireInteraction 327
resolve() 47
respondWith() 304
Response 94, 304
result 102
resume 177
resume() 218
revokeObjectURL() 176
root 84
rootBounds 85
rootMargin 84
RTCDataChannel 291
RTCDataChannelEvent 291
RTCIceCandidate 264
RTCPeerConnection 260
RTCPeerConnectionIceEvent 264
RTCSessionDescription 262
RTCTrackEvent 265
runAll() 239

S

SecurityError 173
self 303
send() 233, 241, 291
serviceWorker 302
ServiceWorkerContainer 302
ServiceWorkerGlobalScope 303
ServiceWorkerRegistration 302
set() 91
setAttribute() 151
setLocalDescription() 262
setRemoteDescription() 262
shadowRoot 147
ShadowRoot 147
show 327
showNotification() 327, 331
silent 327
slot 162, 163
slotchange 166
smoothingTimeConstant 221
srcObject 172
SSL 241
start 177
start() 175
state 177, 217
static 22
status 94
statusText 94
stdClass 247
step 202
StereoPannerNode 211
stop 177
stop() 175, 197
stopAll() 239
stream 177, 210
streams 265
success 102
super 24
suspend() 218
Symbol 57, 58

T

tag 316, 327
takePhoto() 190, 201
takeRecords() 86
target 85
TcpConnection 241
template 159
text() 95
then() 43
threshold 84
time 85
timeRemaining() 137
title 327
track 265
transaction 107, 112
transaction() 108
transport 241
type 212
TypeError 173

U

Uint8Array 221
unhandledrejection 48
unique 107
unobserve() 84
unregister() 305, 316
update() 121
upgradeneeded 101
upper 115
upperBound() 115
upperOpen 116
url 94, 233, 305, 330
userChoice 338

value 117, 211
values() 92
version 104
videoBitsPerSecond 177

waitUntil() 303
wasClean 236
WebRTC 258
WebSocket 231, 232
whenDefined() 158
WindowClient 330
worker 242
Worker 238
Workerman 237

yield 62
yield* 63

А

Автоматическая регулировка усиления 196
Активация клиента 329
Анализатор звука 219
Атрибут тега
◊ id 28

Б

База данных 99
Битрейт 175

В

Веб-компонент 25
Веб-приложение
◊ прогрессивное 298
Веб-сайт
◊ динамический 297
◊ одностраничный 297
◊ сверхдинамический 297
◊ статический 297
Вещание 259
Всплывающее оповещение 324

Г

Генератор 62
◊ асинхронный 67
Геттер 17

Д

Деструктурирование 36
◊ массива 36
◊ объекта 36, 37
Детектор видимости 82
Документ 99

З

Задача
◊ посредника 314
◊ синхронного вывода 138
◊ фоновая 136
Запрос 101
Застревание в кэше 93
Звуковой контекст 207

И

Импорт 71
◊ именованный 75
◊ по умолчанию 74
Индекс 100
◊ уникальный 106
Исключение отклоняющее 42
Источник звука 206
Итератор 57
◊ асинхронный 65

К

Кадр 220
◊ размер 220
Канал данных 290
◊ равноправный режим 290
◊ режим 292
Класс
◊ анонимный 24
◊ базовый 23
◊ именованный 25
◊ производный 23
Ключ 100, 239
◊ закрытый 239
◊ открытый 239
◊ сеанса 240
Контроллер 124
Курсор 116
Кэш
◊ клиентский 92
◊ локальный 92
◊ программируемый 306
Кэширование 92

М

Манифест PWA 334
Медиапоток 172
Метод
◊ асинхронный 55
◊ волшебный 155
◊ генератор 63
◊ генератор асинхронный 67
◊ закрытый 21, 23
◊ общедоступный 21
◊ статический 22
Модель 124
Модуль 71
◊ внешний 71
◊ встроенный 71

Н

Нагрузка 41
Наследование 23

О

Область
◊ видимости 81
◊ кэша 306
Обработчик звука 206
Оператор
◊ возврата с приостановкой 62
◊ вызова генератора 63
◊ ожидания 54
◊ упаковки-распаковки 33, 34
Операция
◊ асинхронная 40
◊ синхронная 40
Осциллограмма 220
Отложенная загрузка 82

П

Перекрытие 23
Переопределение 24
Подкласс 23
Поиск 100
Полоса пропускания 213
Получатель звука 206
Последовательность 56
Посредник 301
Поток 135
◊ основной 135
◊ фоновый 135
Предел 83
Пресет 193
Претендент 259
Приглашение 259
Присваивание деструктурирующее 36
Прогон 76
Промис 41

Р

Разрядность дискретизации 195

С

Свойство
◊ динамическое 17
◊ закрытое 21, 23
◊ индексированное 100
◊ общедоступное 21
◊ статическое 22
Сертификат открытого ключа 240
◊ самоподписанный 240
Сертификационный центр 240
Сеттер 17
Сеть
◊ двухранговая 258
◊ одноранговая 258
Сигнал 232
◊ доступности 269
◊ занятости 268
Символ 58
Синхронный вывод 138
Слот 161
◊ именованный 162
◊ неименованный 162
Слушатель 237
Согласие 259
Соединение
◊ временное 231
◊ постоянное 232
Состояние промиса
◊ ожидания 41
◊ отклоненное 42
◊ подтвержденное 41
Состояние простоя 135
Спектр 221
Спойлер 25
СУБД 99
Суперкласс 23

Т

Тег
◊ компонента 145
◊ расширенный 146
Транзакция 100
◊ откат 100
◊ отклонение 100
◊ подтверждение 100

У

Узел 207

Ф

Фильтр
◊ верхних частот 213
◊ нижних частот 213
◊ полосно-задерживающий 213
◊ полосно-пропускающий 213
◊ фазовый 214
Фильтрация 100
Функция
◊ catch 43
◊ finally 44
◊ then 43

Функция *(прод.)*
- ◊ асинхронная 53
- ◊ отклоняющая 42
- ◊ подтверждающая 42

Х

Хранилище 99

Ц

Цикл
- ◊ перебора последовательности 56
- ◊ перебора с ожиданием 66

Ч

Частота
- ◊ дискретизации 195
- ◊ среза 213
- ◊ центральная 214

Числа Фибоначчи 59

Ш

Шаблон 159
Шифрование
- ◊ асимметричное 239
- ◊ симметричное 240

Э

Экспорт 71
- ◊ именованный 72
- ◊ по умолчанию 72

Элемент веб-страницы
- ◊ базовый 25

Я

Якорь 28

JavaScript

Дополнительные уроки для начинающих

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 30 популярных компьютерных книг, в том числе «Laravel 8. Быстрая разработка веб-сайтов на PHP», «Django 3.0. Практика создания веб-сайтов на Python», «JavaScript. 20 уроков для начинающих», «HTML и CSS. 25 уроков для начинающих», «PHP и MySQL. 25 уроков для начинающих», «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

Книга является продолжением популярной книги «JavaScript. 20 уроков для начинающих». Простым языком, кратко и наглядно рассказано о новых программных инструментах, появившихся в последние годы в языке JavaScript. В книге 20 иллюстрированных уроков, более 20 практических упражнений, 18 заданий для самостоятельной работы и электронное приложение.

Вы узнаете, как

- использовать протокол WebSocket для программирования веб-чата;
- посредством технологии WebRTC создать службу интернет-видеотелефонии;
- захватить видео или фото со встроенной камеры и сохранить его в файле;
- вывести осциллограмму и спектр воспроизводящегося звука;
- наложить эффекты на воспроизводящийся звук;
- сохранить любые документы на стороне клиента во встроенной СУБД и впоследствии извлечь нужный документ по заданным критериям;
- создавать прогрессивные веб-приложения (PWA);
- упростить программирование и сделать свой код простым и наглядным, применив новые инструменты JavaScript.

Осваиваем
новые инструменты
JavaScript

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6781-7

Все необходимые для работы файлы и результаты выполнения упражнений можно скачать по ссылке ftp://ftp.bhv.ru/ 9785977567817.zip, а также со страницы книги на сайте www.bhv.ru.