Руководство jQuery-разработчика

Веб-приложения на JavaScript

Алекс Маккоу

O'REILLY®

Alex MacCaw

JavaScript Web Applications

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Алекс Маккоу

Веб-приложения на JavaScript

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

ББК 32.988.02-018
УДК 004.738.5
М15

Маккоу А.

М15 **Веб-приложения на JavaScript. — СПб.: Питер, 2012. — 288 с.: ил.**

ISBN 978-5-459-01504-1

Поддержка HTML5 и CSS3 с каждым днем становится все качественнее и полнее, но вам необходимо решить, основываясь на вашей клиентской аудитории, где именно можно использовать данные технологии. Ведь создание на языке JavaScript многофункциональных rich-приложений, которые выполняются на стороне клиента, — непростая задача. Это книга поможет вам изучить все приемы, используемые для создания самых современных JavaScript-приложений, в том числе структуры, использование MVC, фреймы, связь с сервером и кросс-доменные запросы, создание приложений реального времени и многое другое.

Чтобы помочь вам понять концепции разработки JavaScript-приложений, рассмотрена работа реальных приложений.

Для опытных разработчиков.

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1449303518 англ. © Authorized Russian translation of the English edition of titled JavaScript Web Applications (ISBN 978-1449303518) © 2011 Alex MacCaw
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-459-01504-1 © Перевод на русский язык ООО Издательство «Питер», 2012
© Издание на русском языке, оформление ООО Издательство «Питер», 2012

Краткое оглавление

Введение ... 13

Глава 1. MVC и классы ... 21

Глава 2. События и наблюдение ... 40

Глава 3. Модели и данные ... 53

Глава 4. Контроллеры и состояния ... 73

Глава 5. Представления и использование шаблонов ... 91

Глава 6. Управление зависимостями ... 100

Глава 7. Работа с файлами ... 109

Глава 8. Веб-технологии реального времени ... 126

Глава 9. Тестирование и отладка ... 137

Глава 10. Развертывание ... 164

Глава 11. Библиотека Spine ... 172

Глава 12. Библиотека Backbone ... 198

Глава 13. Библиотека JavaScriptMVC ... 220

Приложение А. Основы jQuery ... 245

Приложение Б. CSS-расширения ... 257

Приложение В. Справочник по CSS3 ... 262

Оглавление

Введение ... 13

Для кого предназначена эта книга? ... 13
Структура книги ... 14
Соглашения, используемые в данной книге ... 17
Сопроводительные файлы ... 18
Соглашения, касающиеся программного кода ... 18
Примеры jQuery ... 19
Holla ... 19
Примечание автора ... 19
От издательства ... 20

Глава 1. MVC и классы ... 21

В начале пути ... 21
Добавление структуры ... 22
Что такое MVC? ... 23
Модель ... 23
Представление ... 24
Контроллер ... 25
Стремясь к модульности, создаем классы ... 26
Добавление функций к классам ... 28
Добавление методов к нашей библиотеке классов ... 28
Наследование в классе с использованием свойства Prototype ... 31
Добавление наследования к нашей библиотеке класса ... 32
Вызов функции ... 33
Управление областью действия в нашей библиотеке классов ... 35
Добавление закрытых функций ... 37
Библиотеки классов ... 38

Глава 2. События и наблюдение ... 40

Прослушивание событий ... 40
Порядок обработки событий ... 41
Отмена событий ... 42

Объект события 42
Библиотеки событий 44
Изменение контекста 45
Делегирование событий 45
Пользовательские события 46
Пользовательские события и дополнительные модули jQuery 47
Элементы, не имеющие отношения к модели DOM 49

Глава 3. Модели и данные 53

MVC и организация пространства имен 53
Создание ORM 55
Прототипное наследование 55
Добавление свойств ORM 56
Удерживание записей 58
Добавление поддержки ID 59
Адресация ссылок 60
Загрузка в данные 61
Включение данных в код страницы 62
Загрузка данных с помощью Ajax 63
JSONP 66
Безопасность при использовании междоменных запросов 67
Заполнение нашей модели ORM 67
Локальное хранение данных 68
Добавление локального хранилища к нашей модели ORM 70
Отправка новых записей на сервер 71

Глава 4. Контроллеры и состояния 73

Модульная схема 74
Глобальный импорт 74
Глобальный экспорт 74
Добавление контекста 75
Абстрагирование в библиотеку 76
Загрузка контроллеров после документа 78
Доступ к представлениям 79
Делегирование событий 81
Конечные автоматы 83
Маршрутизация 85
Использование хэшей URL-адресов 85
Обнаружение изменений хэша 86
Ajax Crawling 87
Использование History API HTML5 88

Глава 5. Представления и использование шаблонов 91

Динамически интерпретируемые представления 91
Шаблоны 93
Шаблонные помощники 95
Хранение шаблонов 95
Связывание 97
Привязка моделей 98

Глава 6. Управление зависимостями 100

CommonJS 101
Объявление модуля 102
Модули и браузер 102
Загрузчики модулей 103
Yabble 103
RequireJS 104
Изолирование модулей 106
Альтернативы модулям 107
LABjs 107
FUBC 108

Глава 7. Работа с файлами 109

Поддержка браузерами 109
Получение информации о файлах 110
Ввод файлов 110
Перетаскивание 111
Захват и перетаскивание 112
Освобождение после перетаскивания 114
Отмена действия по умолчанию при перетаскивании 115
Копирование и вставка 115
Копирование 116
Вставка после копирования 116
Чтение файлов 117
Блобы и части 119
Собственные кнопки просмотра 120
Передача файлов 120
Индикатор выполнения на основе Ajax 122
Сценарий передачи файлов с использованием перетаскивания и jQuery 124
Создание области освобождения перетаскиваемого элемента 124
Передача файла 125

Глава 8. Веб-технологии реального времени ... 126

История работы в режиме реального времени ... 126
WebSockets ... 127
Node.js и Socket.IO ... 131
Архитектура реального времени ... 133
Ощущаемая скорость ... 135

Глава 9. Тестирование и отладка ... 137

Блочное тестирование ... 139
Утверждения ... 139
QUnit ... 140
Jasmine ... 144
Драйверы ... 146
Автономное тестирование ... 149
Zombie ... 149
Ichabod ... 151
Распределенное тестирование ... 152
Предоставление поддержки ... 153
Инспекторы ... 153
Веб-инспектор ... 154
Firebug ... 155
Консоль ... 156
Вспомогательные функции консоли ... 157
Использование отладчика ... 158
Анализ сетевых запросов ... 160
Профилирование и хронометраж ... 161

Глава 10. Развертывание ... 164

Производительность ... 164
Кэширование ... 165
Минификация ... 167
Сжатие с помощью Gzip ... 169
Использование CDN ... 169
Аудиторы ... 170
Ресурсы ... 171

Глава 11. Библиотека Spine ... 172

Установка ... 172
Классы ... 173
Создание экземпляра ... 173

Расширение классов 174
Контекст 175
События 176
Модели 177
Извлечение записей 178
События моделей 179
Проверка 180
Сохранение состояния 180
Контроллеры 183
Использование представительства 184
Элементы 184
Делегирование событий 185
События контроллера 185
Глобальные события 186
Схема визуализации 186
Схема элемента 187
Создание программы управления контактами 189
Модель Contact 190
Контроллер Sidebar 191
Контроллер Contacts 193
Контроллер App 196

Глава 12. Библиотека Backbone 198

Модели 199
Модели и атрибуты 200
Коллекции 201
Управление порядком следования экземпляров моделей в коллекции 202
Представления 203
Визуализация представлений 204
Делегирование событий 204
Привязка и контекст 205
Контроллеры 206
Синхронизация с сервером 208
Заполнение коллекций 210
На серверной стороне 210
Настройка поведения 211
Создание списка To-Do (текущих дел) 213

Глава 13. Библиотека JavaScriptMVC 220

Установка 221
Классы 221
Создание экземпляра 222
Вызов основных методов 222

Представительства 222
Статическое наследование 223
Самодиагностика 223
Пример модели 223
Модель 224
Атрибуты и наблюдаемые объекты 224
Расширенные модели 227
Методы-установщики 227
Умолчания 228
Методы-помощники 228
Инкапсуляция служб 229
Создание задачи 230
Получение задачи 231
Получение задач 231
Обновление задачи 231
Удаление задачи 232
Преобразование типов 232
CRUD-события 233
Использование в представлении шаблонов на стороне клиента 233
Основное применение 234
Модификаторы jQuery 234
Загрузка из script-тега 235
$.View и подшаблоны 235
Deferred-объекты 235
Упаковка, предварительная загрузка и производительность 236
$.Controller: фабрика по производству дополнительных модулей для jQuery 237
Общее представление 239
Создание экземпляра контроллера 240
Привязка событий 241
Шаблонные действия 241
Объединение компонентов: обобщенный CRUD-список 243

Приложение А. Основы jQuery 245

Обход элементов DOM-модели 247
Работа с DOM 248
События 250
Ajax 251
Исполнение роли законопослушной гражданки 252
Расширения 253
Создание дополнительного модуля jQuery Growl 254

Приложение Б. CSS-расширения 257

Переменные 257
Миксины 258
Вложенные правила 258
Включение других таблиц стилей 259
Цвета 259
Как можно воспользоваться Less? 259
  С помощью окна командной строки 260
  С помощью модуля Rack 260
  С помощью JavaScript 260
  Less.app 260

Приложение В. Справочник по CSS3 262

Префиксы 263
Цвета 263
Скругленные углы 264
Отбрасываемые тени 265
Тени для текста 266
Градиенты 267
Составной фон 268
Селекторы 268
  N-й дочерний элемент 269
  Прямой потомок 269
  Обращение селекторов 270
Переходы 270
Создание границ с помощью изображения 271
Изменения алгоритма расчета ширины и высоты элемента 272
Преобразования 272
Модель гибких прямоугольных блоков 273
Шрифты 274
Постепенная деградация 275
  Modernizr 276
  Расширение Google Chrome Frame 277
Создание макета 278

Введение

С 1995 года JavaScript прошел длинный путь от скромного компонента браузера Netscape до современных высокопроизводительных JIT-интерпретаторов. Казалось бы, всего лет пять назад разработчики были ошеломлены появлением Ajax и технологии постепенно исчезающей желтой подсветки, а уже сейчас сложные JavaScript-приложения достигли объемов в сотни и тысячи строк кода.

В прошлом году появилось новое поколение JavaScript-приложений, ничем не отличающихся от приложений рабочего стола, — невероятный прогресс веб-технологий. Ушли в прошлое медленно выполняемые запросы страниц при каждом взаимодействии пользователя с приложением. Движки JavaScript стали настолько мощными, что появилась возможность сохранять состояние на стороне клиента, что существенно ускорило реакцию приложения и улучшило качество его работы.

Но усовершенствовались не только движки JavaScript: спецификации CSS3 и HTML5, еще только выйдя из стадии разработки, уже получили широкую поддержку в таких современных браузерах, как Safari, Chrome, Firefox и, в известной степени, в IE9. Программирование привлекательных интерфейсов теперь занимает всего лишь небольшую часть того времени, которое требовалось для этого в прежние времена, и уже не требует всех этих пресловутых вырезок и сборок изображений. Поддержка HTML5 и CSS3 с каждым днем становится все качественнее и полнее, но вам нужно решить, основываясь на вашей клиентской аудитории, где именно можно использовать данные технологии.

Перемещение состояния приложения на сторону клиента — непростая задача. Она требует совершенно иных подходов к разработке приложений на серверной стороне. Нужна хорошо продуманная структура, организация работы с шаблонами, четкое взаимодействие с сервером, с работающими на нем программными средами и многое другое. Для освещения этих вопросов и предназначена эта книга. Я проведу вас по всем этапам, необходимым для создания самых современных JavaScript-приложений.

Для кого предназначена эта книга?

Она не для новичков JavaScript, поэтому если вы незнакомы с основами языка, я советую почитать «JavaScript: Сильные стороны» Дугласа Крокфорда («Питер», 2012). А наша книга предназначена для разработчиков с определенным опытом

работы на JavaScript, с использованием таких библиотек, как jQuery — для тех, кто хочет приступить к созданию более совершенных JavaScript-приложений. Кроме этого, многие разделы данной книги, и особенно приложения, также могут послужить полезным справочником для опытных JavaScript-разработчиков.

Структура книги

Глава 1

Глава начинается с рассмотрения истории JavaScript и затрагивает основы языка, оказывающие влияние на текущую реализацию и на круг его пользователей. Затем дается введение в архитектуру MVC, исследуются имеющиеся в JavaScript функции-конструкторы, прототипное наследование и способы создания собственной библиотеки классов.

Глава 2

Здесь речь пойдет об обработке событий в браузерах, включая историю, API и поведение. Рассматриваются вопросы привязки событий с помощью jQuery, использования делегирования и создания собственных событий. Также исследуется использование событий, не имеющих отношения к DOM-модели, с применением схемы PubSub.

Глава 3

В этой главе приведен порядок использования MVC-моделей в приложении, а также порядок загрузки удаленных данных и работы с ними. Объясняется важность MVC и организации пространства имен, а затем создается собственная ORM-библиотека для работы с данными модели. Затем рассматриваются способы загрузки удаленных данных и использованием JSONP и кроссдоменной технологии Ajax. И наконец, даются сведения о сохранении данных модели с помощью локального хранилища HTML5 и их отправке на сервер, отвечающий требованиям RESTful.

Глава 4

В этой главе показываются способы использования схемы контроллера для сохранения состояния на стороне клиента. Рассматриваются вопросы использования модулей для инкапсуляции логики и предотвращения засорения глобального пространства имен, затем рассматривается вопрос аккуратного связывания контроллеров с представлениями, прослушивания событий и работы с DOM. И наконец, рассматриваются вопросы маршрутизации, сначала с использованием фрагмента URL, начинающегося с символа решетки, а затем с использованием нового API истории HTML5. Мы стремимся объяснить все «за» и «против» обоих подходов.

Глава 5

Здесь рассматриваются представления и работа с шаблонами с помощью JavaScript. Рассматриваются разные способы динамического вывода представлений, а также различные библиотеки работы с шаблонами и места хранения шаблонов (в составе страниц, в `script`-тегах или удаленное хранение с загруз-

кой). Затем рассматривается привязка данных — подключение контроллеров моделей и представления к динамически синхронизируемым данным модели и данным представления.

Глава 6

В этой главе рассматриваются подробности управления JavaScript-зависимостями с использованием модулей CommonJS. Изучается история и рассуждения, положенные в основу механизма CommonJS, способы создания CommonJS-модулей в браузере и различные вспомогательные библиотеки загрузки модулей, такие как Yabble и RequireJS. Затем рассматриваются способы автоматической изоляции модулей на стороне сервера, повышения производительности и экономии времени. И наконец, рассматриваются разные альтернативы CommonJS, такие как Sprockets и LABjs.

Глава 7

Здесь изучается преимущество, предоставляемое HTML5: API для работы с файлами. Рассматривается браузерная поддержка, множественная отправка файлов на сервер, получение файлов, перетаскиваемых в окно браузера, и файлов от событий клавиатуры. Затем объясняется чтение файлов с использованием блобов и слайсов (частей) и вывод результатов в браузер. Рассматриваются вопросы отправки файлов на сервер в фоновом режиме с использованием новой спецификации XMLHttpRequest Level и, наконец, показывается способ предоставления пользователям индикатора выполнения операции отправки файла и способ объединения отправки файлов с имеющимся в jQuery Ajax API.

Глава 8

Здесь рассматриваются некоторые весьма увлекательные вопросы разработки приложений реального времени и технология WebSockets. Сначала в главе рассматривается довольно бурная история развития технологии реального времени и ее текущая поддержка в браузерах. Затем изучаются подробности WebSockets и высокоуровневой реализации этой технологии, браузерной поддержки и API JavaScript. После этого демонстрируется простой RPC-сервер, использующий WebSockets для соединения серверов и клиентов. Затем уделяется внимание Socket.IO и изучается вопрос вписывания технологий реального времени в архитектуру приложения и в пользовательское восприятие.

Глава 9

В этой главе рассматриваются тестирование и отладка, являющиеся очень важной частью разработки веб-приложений на JavaScript. Изучаются вопросы, касающиеся кроссбраузерного тестирования, на каких браузерах следует проводить тестирование, вопросы блочного тестирования и рассматриваются библиотеки для тестирования, такие как QUnit и Jasmine. Затем обращается внимание на автоматизированное тестирование и на постоянно работающие объединительные сервера, такие как Selenium. Затем осуществляется переход к вопросам отладки, исследуются веб-инспекторы Firefox и WebKit, консоль и вопросы использования отладчика JavaScript.

Глава 10

В этой главе рассматривается еще одна важная, но часто недооцениваемая часть работы с веб-приложением JavaScript: его развертывание. Упор делается главным образом на производительность и на способы использования кэширования, минификации, gzip-сжатия и других технологий, сокращающих начальное время загрузки вашего приложения. В конце главы кратко рассматриваются вопросы использования сети доставки контента (CDN) для обслуживания статического содержимого в ваших интересах и вопросы использования встроенного в браузер механизма аудита, который может быть чрезвычайно полезен для повышения производительности вашего сайта.

Глава 11

Следующие три главы являются введением в некоторые популярные JavaScript-библиотеки, предназначенные для разработки приложений. Spine является весьма небольшой по объему MVC-совместимой библиотекой, использующей множество концепций, рассмотренных в данной книге. В главе рассматриваются основные части этой библиотеки: классы, события, модели и контроллеры. В конце главы создается пример приложения, являющегося диспетчером контактов, в котором демонстрируется все, изученное в данной главе.

Глава 12

В главе дается полноценное введение в имеющую огромную популярность библиотеку Backbone, предназначенную для создания JavaScript-приложений. Рассматриваются основные понятия и классы Backbone, такие как модели, коллекции, контроллеры и представления. Затем исследуются вопросы синхронизации данных модели с данными на сервере с помощью RESTful JSON-запросов и вопросы составления подходящих для Backbone ответов со стороны сервера. В конце главы создается приложение для ведения списка текущих дел, в котором демонстрируется работа основной части библиотеки.

Глава 13

В этой главе исследуется библиотека JavaScriptMVC, являющаяся популярной средой разработки, основанной на использовании библиотеки jQuery и применяемой для создания веб-приложений на JavaScript. Изучаются все основные компоненты JavaScriptMVC, такие как классы, модели и контроллеры, а также использование шаблонов на стороне клиента с целью визуализации представлений. В конце главы дается практический пример списка, реагирующего на CRUD-операции, который демонстрирует простоту создания абстрактных, многократно используемых, не засоряющих память виджетов с помощью JavaScriptMVC.

Приложение А

Это приложение предоставляет краткое введение в jQuery, которое пригодится, если вы почувствуете необходимость освежить свое представление об этой библиотеке. Библиотека jQuery используется в большинстве примеров этой книги, поэтому знакомство с ней играет весьма важную роль. Здесь рассматривается основная часть API, например обход элементов DOM, работа с DOM, а также привязка событий, их инициирование и делегирование. Затем более пристально

рассматривается имеющийся в jQuery Ajax API, отправка GET и POST JSON-запросов. После этого рассматриваются расширения jQuery и порядок использования инкапсуляции для обеспечения этой библиотекой роли законопослушной веб-гражданки. В конце приложения рассматривается практический пример: создание дополнительного модуля jQuery под названием Growl.

Приложение Б

В Приложении Б рассматривается Less, надстройка над CSS, расширяющая синтаксис каскадных таблиц стилей с помощью переменных, миксинов, операций и вложенных правил. Less способна существенно сократить объем набираемого кода CSS, особенно когда дело касается специфических для производителей браузеров CSS3-правил. В этом приложении рассматриваются основные улучшения синтаксиса, присущие Less, а также вопросы использования инструментов командной строки и библиотеки JavaScript для компиляции Less-файлов в обычный код CSS.

Приложение В

Последнее приложение является справочником по CSS3. В нем предоставляются основы CSS3, объясняются префиксы производителей браузеров, а затем просматриваются основные добавления к спецификации. Среди других свойств CSS3 в этом приложении рассматриваются скругленные углы, `rgba`-цвета, отбрасываемые тени, градиенты, переходы и преобразования. В конце приложения рассматривается так называемая постепенная деградация с использованием библиотеки Modernizr и практический пример использования новой спецификации `box-sizing`.

Соглашения, используемые в данной книге

В данной книге используются следующие соглашения, связанные с типографским оформлением:

Курсив

Служит признаком новых понятий.

Шрифт без засечек

Служит признаком URL, адресов электронной почты, имен файлов, расширений этих имен.

`Моноширинный шрифт`

Служит признаком компьютерного кода в широком смысле, включая команды, массивы, элементы, инструкции, варианты, переключатели, переменные, атрибуты, ключи, функции, типы, классы, пространства имен, методы, модули, свойства, аргументы, значения, объекты, обработчики событий, XML-теги, HTML-теги, макросы, содержимое файлов и вывод из команд.

`Моноширинный наклонный шрифт`

Показывает текст, который должен быть заменен значениями, предоставляемыми пользователями, или значениями, определяемыми контекстом.

Сопроводительные файлы

Дополнительные файлы к книге выложены на хостинге GitHub (https://github.com/maccman/book-assets). Вы можете просматривать их в интерактивном режиме или же загрузить zip-архив (https://github.com/maccman/book-assets/zipball/master) и работать с ними локально. Все ресурсы разбиты по главам и содержат все требуемые библиотеки. Большинство примеров, приводимых в данной книге, доступны также в виде отдельных файлов.

Так что любая использованная в этой книге ссылка на конкретный ресурс будет иметь вид assets/*номер_главы*/*имя*.

Соглашения, касающиеся программного кода

Во всей книге для демонстрации значений переменных или результатов вызова функций будут использоваться функции `assert()` и `assertEqual()`. Функция `assert()` (утверждение) является простым способом демонстрации того, что значение конкретной переменной сводится к `true`; это широко распространенная схема, получившая особое распространение в автоматическом тестировании. Функции `assert()` передаются два аргумента: значение и необязательное сообщение. Если значение не равно `true`, функция выдаст ошибку:

```
var assert = function(value, msg) {
    if ( !value )
    throw(msg || (value + " не равно true"));
};
```

Функция `assertEqual()` является простым способом демонстрации того, что значение одной переменной равно значению другой переменной. Она работает практически так же, как и функция `assert()`, но ей передаются уже два значения. Если эти два значения не равны друг другу, утверждение становится несостоятельным:

```
var assertEqual = function(val1, val2, msg) {
    if (val1 !== val2)
    throw(msg || (val1 + " не равно " + val2));
};
```

Пользоваться этими двумя функциями, как видно из приводимых ниже примеров, довольно просто. Если утверждение не состоялось, вы увидите в консоли браузера сообщение об ошибке:

```
assert( true );

// Эквивалент функции assertEqual()
assert( false === false );

assertEqual( 1, 1 );
```

Я немного приукрасил работу функции `assertEqual()`, поскольку, как оказалось, сравнение объектов не будет успешным, пока они не будут использовать одну и ту

. , -
assets/chOO/deep_equality.html.

jQuery

jQuery, -
JavaScript- , ,
DOM- ,
Ajax. -
, , , jQuery
JavaScript,
.

jQuery, -
. API, -
DOM- . jQuery
.

Holla

, Holla ,
JavaScript. Holla , -
,
. , Holla , :
CSS3 HTML5 ;
;
Sprockets Less;
WebSockets ;
JavaScript- , .

Holla GitHub **(http://github.com/maccman/holla)** .
Holla (. . 0.1).

, . -
,
— -
. , -
,
, .

.
(Stuart Eccles), (Tim Malbon),

Рис. 0.1. Holla, пример чат-приложения

(Ben Griffins) и Шону О'Халпину (Sean O'Halpin), предоставившим мне шанс и возможность испытать столько впечатлений; и Джеймсу Адаму (James Adam), Полу Баттли (Paul Battley) и Джона Фоксу (Jonah Fox) за наставления и терпеливое отношение к моим глупостям.

Спасибо также техническим рецензентам, внесшим реальную помощь в придание книге нужной формы: Хенрику Джоретегу (Henrik Joreteg), Джастину Мейеру (Justin Meyer), Леа Веро (Lea Verou), Адди Османи (Addy Osmani), Алексу Барбара (Alex Barbara), Максу Уильямсу (Max Williams) и Джулио Цезару Оду (Julio Cesar Ody).

Но, что самое важное, спасибо моим родителям за их поддержку.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты `comp@piter.com` (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти на хостинге GitHub (https://github.com/maccman/book-assets).

На веб-сайте издательства `http://www.piter.com` вы найдете подробную информацию о наших книгах.

1 MVC и классы

В начале пути

По сравнению с первоначальным замыслом разработка на JavaScript заметно изменилась. Трудно представить, какой длинный путь прошел этот язык от своей исходной реализации в браузере Netscape до современных мощных машин, таких как созданный Google механизм V8. Это был тернистый путь переименований, объединений и конечной стандартизации в виде ECMAScript. Сегодняшние возможности превзошли самые смелые мечты первых разработчиков.

Но несмотря на успех и популярность JavaScript, его по-прежнему зачастую недооценивают. Лишь немногие знают, что это очень эффективный и динамичный объектно-ориентированный язык, но и эти немногие очень удивляются, узнав о таких его более развитых свойствах, как прототипное наследование, использование модулей и пространств имен. Так в чем же причина такой недооценки JavaScript?

Отчасти она возникла из-за предыдущих, не самых удачных реализаций JavaScript, а отчасти она заложена в самом имени этого языка, где префикс *Java* предполагает некоторое отношение к языку Java, хотя в действительности это совершенно разные языки. И все-таки я считаю, что настоящая причина кроется в том, как этот язык преподнесен большинству разработчиков. Если взять другие языки, например Python или Ruby, разработчики зачастую прилагают общие усилия для изучения языка по книгам, видеороликам и обучающим программам. Но до недавнего времени JavaScript такой поддержкой не пользовался. От разработчиков могли требовать добавления JavaScript к уже существующему коду, зачастую в весьма сжатые сроки, чтобы осуществить проверку приемлемости данных в формах, а также, возможно, для каких-нибудь выделений текста или для создания фотогалерей. Они могли воспользоваться сценариями, найденными в Интернете, и посчитать работу завершенной, так толком и не разобравшись в языке, положенном в основу этих сценариев. После такого поверхностного соприкосновения с языком некоторые разработчики могли даже добавить знание JavaScript в свои резюме.

В последнее время JavaScript-машины и браузеры приобрели такую мощность, что создание полноценных приложений на JavaScript стало не только реально

выполнимой, но и весьма популярной задачей. Такие приложения, как Gmail и Google Maps, создали предпосылки для совершенно иного взгляда на веб-приложения, и пользователи стали еще требовательнее. Компании стали нанимать разработчиков программ на JavaScript на постоянную работу. JavaScript перестал быть вспомогательным инструментом, предназначенным только для написания простых сценариев и подсистемы проверки приемлемости данных формы, — теперь он стал вполне самостоятельным языком, имеющим право на реализацию своего полного потенциала.

Этот рост популярности означает, что было создано множество новых JavaScript-приложений. К сожалению, вероятно, из-за сложной истории этого языка многие из этих приложений весьма убоги. По каким-то причинам, как только дело доходит до JavaScript, все общепризнанные шаблоны и богатый практический опыт куда-то улетучиваются. Разработчики игнорируют такие архитектурные модели, как схему модель-представление-контроллер (Model View Controller, MVC), создавая вместо этого в своих приложениях некую небрежную смесь HTML и JavaScript.

Эта книга вряд ли сможет стать для вас обычным учебником по JavaScript как по языку программирования, для этого больше подходят другие книги — например, книга Дугласа Крокфорда (Douglas Crockford) «JavaScript: The Good Parts» (издательство O'Reilly). Но наша книга покажет вам, как структурировать и создавать сложные JavaScript-приложения, позволяя приобрести высочайшую квалификацию в сфере веб-разработки.

Добавление структуры

Секрет создания больших JavaScript-приложений кроется *не* в создании огромных приложений как таковых, а в том, что нужно разбить ваше приложение на ряд практически независимых компонентов. Разработчики часто допускают ошибку, создавая приложения, содержащие множество взаимозависимостей, с огромными и прямолинейными JavaScript-файлами, генерирующими массу HTML-тегов. Приложения такого сорта трудно обслуживать и расширять, поэтому их нужно избегать любой ценой.

Существенно повлиять на конечный результат может внимание к структуре приложения, проявленное в самом начале разработки. Выбросьте из головы все предубеждения относительно JavaScript и считайте его полноценным объектно-ориентированным языком — так и есть на самом деле. Используйте классы, наследования, объекты и схемы точно так же, как это делается при создании приложения на другом языке, например на Python или Ruby. Архитектура играет большую роль для серверных приложений, так почему бы ее не перенести и на приложения, работающие на стороне клиента?

В этой книге пропагандируется подход, связанный со схемой MVC, являющийся опробованным и проверенным способом проектирования приложений, гарантирующий возможность их эффективного обслуживания и расширения. Кроме того, эта схема особенно хорошо вписывается в JavaScript-приложения.

Что такое MVC?

MVC — это шаблон проектирования, разбивающий приложение на три части: данные (Model, модель), уровень их отображения (View, представление) и уровень взаимодействия с пользователем (Controller, контроллер). Иными словами, события протекают следующим образом.

1. Пользователь взаимодействует с приложением.
2. Вызываются обработчики событий контроллера.
3. Контроллер запрашивает данные из модели, предоставляя их представлению.
4. Представление предоставляет данные пользователю.

Или, чтобы дать вам реальный пример, на рис. 1.1 показано, как отправка нового чат-сообщения будет работать с Holla.

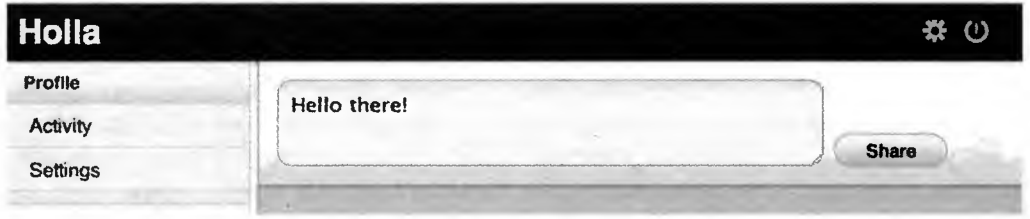

Рис. 1.1. Отправка нового чат-сообщения из Holla

1. Пользователь отправляет новое чат-сообщение.
2. Вызываются обработчики событий контроллера.
3. Контроллер создает новую запись в модели Chat Model.
4. Затем контроллер обновляет представление.
5. Пользователь видит свое новое чат-сообщение в чат-записи.

Архитектурный шаблон MVC может быть реализован даже без библиотек или инфраструктур. Главное — разделить зоны ответственности MVC-компонентов на четко определенные фрагменты кода, сохраняя их обособленность. Это позволит вести независимую разработку, тестирование и обслуживание каждого компонента.

Давайте изучим компоненты MVC более подробно.

Модель

Модель является тем местом, где хранятся объекты данных приложения. Например, у нас может быть модель `User`, содержащая перечень пользователей, их свойства и любую логику, связанную конкретно с этой моделью.

Модель ничего не знает о представлениях и контроллерах. Единственным содержимым модели являются данные и логика, непосредственно связанная с этими данными.

Любой код обработки событий, шаблоны представлений или логика, не имеющая отношения к модели, должны храниться за ее пределами. Как только в моделях будет замечен код представления, знайте: MVC-архитектура приложения нарушена. Модели должны быть полностью отделены от всего остального приложения.

Когда контроллеры извлекают данные с серверов или создают новые записи, они заключают их в экземпляры модели. Это означает, что ваши данные являются объектно-ориентированными, и любые функции или логика, определенная в этой модели, могут быть вызваны непосредственно в отношении этих данных.

Поэтому, вместо этого:

```
var user = users["foo"];
destroyUser(user);
```

можно сделать вот это:

```
var user = User.find("foo");
user.destroy();
```

Первый пример не использует пространство имен или объектную ориентацию. Если в нашем приложении определена еще одна функция `destroyUser()`, эти две функции будут конфликтовать. Количество глобальных переменных и функций должно быть всегда сведено к абсолютному минимуму. Во втором примере функция `destroy()` входит в пространство имен экземпляров класса `User`, как и все сохраненные записи. Это идеальный вариант, так как мы сводим количество глобальных переменных к минимуму, открывая меньше областей для потенциальных конфликтов. Код выглядит аккуратнее и использует возможности наследования, поэтому функции, подобные `destroy()`, не требуют отдельного определения в каждой модели.

Более подробно модели будут исследованы в главе 3, в которой рассматриваются такие темы, как загрузка данных с серверов и создание средств объектно-реляционных отображений (object-relational mapper, ORM).

Представление

Уровень представления — это то, что показывается пользователю, и с чем, собственно, он взаимодействует. В приложениях JavaScript представление будет создаваться преимущественно с помощью шаблонов HTML, CSS и JavaScript. Кроме простых инструкций, задающих условия в шаблонах, представления не должны содержать никакой логики.

Фактически, как и модели, представления должны быть отделены от всего остального приложения. Представления ничего не должны знать о контроллерах и моделях — они должны быть независимы. Смешивание представлений с логикой — верный путь к проблемам.

Но это не значит, что в MVC не разрешена презентационная логика, если таковая не определена внутри представлений. Презентационная логика находится в так назы-

ваемых помощниках: сценариях, предназначенных исключительно для небольших вспомогательных функций, связанных с представлением.

Нижеприведенный пример, включающий логику в представления, показывает, что не нужно делать:

```
// template.html
<div>
  <script>
    function formatDate(date) {
      /* ... */
    };
  </script>
  ${ formatDate(this.date) }
</div>
```

В этом коде функция `formatDate()` вставляется непосредственно в представление, что нарушает принцип MVC, и в результате чего получается какая-то беспорядочная смесь из тегов, не поддающаяся облуживанию. Выделяя презентационную логику в помощники, как в примере, показанном ниже, мы избегаем этой проблемы, сохраняя структуру нашего приложения MVC-совместимой.

```
// helper.js
var helper = {};
helper.formatDate = function(){ /* ... */ };
// template.html
<div>
  ${ helper.formatDate(this.date) }
</div>
```

Вдобавок к этому вся презентационная логика заключена в пространство имен внутри переменной `helper`, предупреждая конфликты и сохраняя чистоту и расширяемость кода.

Специфика представлений и шаблонов нас пока не должна особо волновать, поскольку она будет подробно рассмотрена в главе 5. Цель данного раздела — показать место представлений в архитектурной схеме MVC.

Контроллер

Контроллеры являются связующим звеном между моделями и представлениями. Контроллеры получают сведения о событиях, а также ввод от представлений, обрабатывают их (возможно, с привлечением моделей) и соответствующим образом обновляют представления. При загрузке страницы контроллер добавляет к представлениям приемники событий для определения моментов отправки форм или щелчков на кнопках. Затем, когда пользователь работает с приложением, события вызывают действия внутри контроллеров.

Для реализации контроллеров не понадобятся какие-либо библиотеки или объектные структуры. Вот пример, в котором используется старый добрый jQuery:

```
var Controller = {};
// использование безымянной функции для инкапсуляции области действия
(Controller.users = function($){
  var nameClick = function(){
    /* ... */
  };
  // Присоединение приемников событий при загрузке страницы
  $(function(){
    $("#view .name").click(nameClick);
  });
})(jQuery);
```

Мы создаем контроллер `users`, заключенный в пространство имен переменной `Controller`. Затем мы используем безымянную функцию для инкапсулирования области действия, предотвращая «загрязнение» переменными глобальной области действия. При загрузке страницы мы добавляем приемник события *щелчка* к элементу представления.

Как видите, контроллеры не требуют библиотеки или объектной структуры. Но для выполнения архитектурных требований MVC они должны быть отделены от моделей и представлений. Более подробно контроллеры и состояния рассмотрены в главе 4.

Стремясь к модульности, создаем классы

Прежде чем добраться до основных элементов MVC, мы собираемся рассмотреть ряд предварительных понятий, таких как классы и события JavaScript. Тем самым будет заложен надежный фундамент для перехода к более широким понятиям.

Литералы объектов JavaScript хороши для статических классов, но они зачастую полезны для создания классов в их классическом понимании, с наследованием и экземплярами. Важно подчеркнуть, что JavaScript является языком прототипов и, по сути, не включает естественную реализацию класса. Но поддержка может быть довольно легко сымитирована.

О классах в JavaScript часто отзываются плохо, критикуют их, говоря, что они не являются частью «пути JavaScript» — понятия, которое, по сути, ничего не значит. Библиотека jQuery, когда дело доходит до структурной методологии или схем наследования, выдерживает нейтралитет. В силу этого у разработчиков JavaScript-приложений может сложиться мнение, что они не должны брать в расчет структуру, т. е. что классы не доступны или не могут быть использованы. На самом деле классы — это всего лишь еще один инструмент, и, будучи прагматиком, я верю, что они полезны в JavaScript точно так же, как и в любом другом современном языке.

Вместо определений класса в JavaScript есть функции-конструкторы и оператор `new`. Функция-конструктор может определить начальные свойства и значения

объекта при создании его экземпляра. В качестве конструктора может быть использована любая функция JavaScript. Для создания нового экземпляра объекта нужно с функцией-конструктором использовать инструкцию new.

Оператор new изменяет контекст функции, а также поведение инструкции return. По сути, использование new и конструкторов очень похоже на использование языков с естественной реализацией классов:

```
var Person = function(name) {
  this.name = name;
};
// Создание экземпляра Person
var alice = new Person('alice');
// Проверка наличия экземпляра
assert( alice instanceof Person );
```

По соглашению, чтобы функции-конструкторы отличались от обычных функций, используют имена в смешанном регистре, которые начинаются с большой буквы. Это важно, потому что вам никогда не потребуется вызывать функцию-конструктор без префикса new.

```
// Так делать нельзя!
Person('bob'); //=> undefined (не определена)
```

Функция просто вернет undefined, и поскольку контекстом является объект окна (глобальный объект), вы невольно создадите глобальную переменную name. Функции-конструкторы всегда нужно вызывать, используя ключевое слово new.

Когда функция-конструктор вызывается с использованием ключевого слова new, контекст переключается с глобального (окно) на новый пустой контекст, характерный для данного экземпляра. Значит, ключевое слово this ссылается на текущий экземпляр. При всей кажущейся сложности, по сути, это можно считать неким подобием естественной реализации класса в других языках.

По умолчанию, если вы ничего не возвращаете из функции-конструктора, будет возвращен текущий контекст — this. Или же можно вернуть данные любого непримитивного типа, например, функцию, учреждающую новый класс, которая станет первым шагом на пути создания нашей собственной библиотеки имитации класса:

```
var Class = function(){
  var klass = function(){
    this.init.apply(this, arguments);
  };
  klass.prototype.init  = function(){};
  return klass;
};
var Person = new Class;
Person.prototype.init = function(){
  // Вызывается при создании экземпляра Person
};
// Порядок использования:
var person = new Person;
```

Вот причина путаницы: согласно так никогда и не реализованной спецификации JavaScript 2, «class» является зарезервированным ключевым словом. По общепринятому соглашению вместо имени переменной «class» используется имя `_class` или `klass`.

Добавление функций к классам

Добавление функций класса к функции конструктора аналогично добавлению свойства к любому объекту в JavaScript:

```
Person.find = function(id){ /*...*/ };
var person = Person.find(1);
```

Для добавления функций экземпляра к функции конструктора нужно воспользоваться конструкторским свойством `prototype`:

```
Person.prototype.breath = function(){ /*...*/ };
var person = new Person;
person.breath();
```

Сложилась практика назначения прототипу класса укороченного псевдонима `fn`:

```
Person.fn = Person.prototype;
Person.fn.run = function(){ /*...*/ };
```

Примеры можно наблюдать во всех дополнительных модулях jQuery, где просто добавляются функции к прототипу jQuery, имеющему псевдоним `jQuery.fn`.

Добавление методов к нашей библиотеке классов

На данный момент наша библиотека классов включает средства создания и инициализации экземпляров. Добавление свойств к классам — то же самое, что и добавление свойств к функциям-конструкторам.

Свойства, установленные непосредственно в отношении класса, будут эквивалентны статическим функциям-членам класса:

```
var Person = new Class;

// Статические функции, добавленные непосредственно в отношении класса
Person.find = function(id){ /* ... */ };

// И теперь эти функции можно вызывать напрямую
var person = Person.find(1);
```

А свойства, установленные в отношении прототипа класса, также доступны в отношении экземпляров:

```
var Person = new Class;

// Функции экземпляра, объявленные в отношении прототипа
Person.prototype.save = function(){ /* ... */ };

// И теперь их можно вызвать в отношении экземпляров
var person = new Person;
person.save();
```

Но я считаю этот синтаксис немного запутанным, непрактичным и имеющим много повторений. При его использовании трудно с первого взгляда определить перечень статических свойств класса, а также свойств экземпляра класса. Вместо этого давайте создадим другой способ добавления свойств к нашему классу с использованием двух функций: `extend()` и `include()`:

```
var Class = function(){
  var klass = function(){
    this.init.apply(this, arguments);
  };

  klass.prototype.init  = function(){};

  // Сокращенная форма записи для доступа к прототипу
  klass.fn = klass.prototype;

  // Другая форма записи для доступа к классу
  klass.fn.parent = klass;

  // Добавление свойств класса
  klass.extend = function(obj){
    var extended = obj.extended;
    for(var i in obj){
      klass[i] = obj[i];
    }
    if (extended) extended(klass)
  };

  // Добавление свойств экземпляра
  klass.include = function(obj){
    var included = obj.included;
    for(var i in obj){
      klass.fn[i] = obj[i];
    }
    if (included) included(klass)
  };
  return klass;
};
```

В показанной выше улучшенной библиотеке класса мы добавляем к сгенерированным классам функцию расширения — `extend()`, которая принимает объект. Осуществляется последовательный перебор свойств объекта и их копирование непосредственно в класс:

```
var Person = new Class;
Person.extend({
  find:   function(id) { /* ... */ },
  exists: functions(id) { /* ... */ }
});
var person = Person.find(1);
```

Функция включения — `include()` — работает точно так же, за исключением того, что свойства копируются в прототип класса, а не непосредственно в сам класс. Иными словами, свойства попадают в экземпляры класса, а не оседают статически в классе.

```
var Person = new Class;
Person.include({
  save:    function(id) { /* ... */ },
  destroy: functions(id) { /* ... */ }
});
var person = new Person;
person.save();
```

Мы также реализуем поддержку откликов на расширения (`extended`) и на включения (`included`). Если эти свойства присутствуют в переданном объекте, они будут вызваны:

```
Person.extend({
  extended: function(klass) {
    console.log(klass, " был расширен!");
  }
});
```

Если вам уже приходилось использовать классы в Ruby, все это будет очень знакомо. Достоинство такого подхода в том, что теперь мы получили поддержку модулей. Модулями называются повторно используемые фрагменты кода, которые могут стать альтернативой наследованию для совместного использования классами общих свойств.

```
var ORMModule = {
  save: function(){
    // Обшая функция
  }
};
var Person = new Class;
var Asset  = new Class;
Person.include(ORMModule);
Asset.include(ORMModule);
```

Наследование в классе с использованием свойства Prototype

Мы уже много раз использовали свойство `prototype`, но так до сих пор и не дали ему достаточного объяснения. Давайте присмотримся к тому, что оно собой представляет и как им можно воспользоваться для реализации формы наследования в наших классах.

JavaScript является языком, основанным на прототипах, заменяющих в нем различия между классами и экземплярами, т. е. в нем имеется такое понятие как *объект-прототип*: это объект, используемый в качестве шаблона, из которого берутся исходные свойства для нового объекта. Любой объект может рассматриваться в качестве прототипа для другого объекта, выделяя ему свои свойства. С практической точки зрения это можно рассматривать в качестве своеобразной формы наследования.

При извлечении свойства объекта JavaScript будет искать его в локальном объекте. Если оно не будет найдено, JavaScript приступит к поиску в прототипе объекта и продолжит поиск вверх по дереву прототипа, доходя в конечном итоге до `Object.prototype`. Если свойство найдено, возвращается его значение, а если нет, будет возвращено значение `undefined` (не определено).

Иными словами, если приступить к добавлению свойств к `Array.prototype`, это отразится на каждом массиве JavaScript.

Чтобы создать подкласс на основе класса с унаследованием свойств, сначала нужно определить функцию-конструктор. Затем нужно назначить новый экземпляр родительского класса в качестве прототипа для вашей функции-конструктора. Это выглядит следующим образом:

```
var Animal = function(){};

Animal.prototype.breath = function(){
  console.log('дыхание');
};

var Dog = function(){};

// Dog (собака) наследует свойства у Animal (животного)
Dog.prototype = new Animal;
Dog.prototype.wag = function(){
  console.log('виляющий хвост');
};
```

А теперь работу наследования можно проверить:

```
var dog = new Dog;
dog.wag();
dog.breath(); // унаследованное свойство
```

Добавление наследования к нашей библиотеке класса

Давайте добавим наследование к разрабатываемой нами библиотеке класса. При создании нового класса мы пройдем через необязательный родительский класс (parent):

```
var Class = function(parent){
  var klass = function(){
    this.init.apply(this, arguments);
  };

  // Изменение прототипа для klass
  if (parent) {
    var subclass = function() { };
    subclass.prototype = parent.prototype;
    klass.prototype = new subclass;
  };

  klass.prototype.init = function(){};

  // Сокращения
  klass.fn = klass.prototype;
  klass.fn.parent = klass;
  klass._super = klass.__proto__;

  /* код включений (include) и расширений (extend)... */

  return klass;
};
```

Если конструктору Class передан аргумент parent, любым подклассам обеспечивается передача одних и тех же свойств. Эта небольшая пляска вокруг создания временных безымянных функций освобождает при наследовании класса от создания экземпляров. Но при этом следует знать, что наследуются только свойства экземпляра, а не свойства класса. Пока еще нет кроссбраузерного способа установки объектного __proto__;. Такие библиотеки, как Super.js, обходят эту проблему, копируя свойства вместо реализации надлежащего динамического наследования.

Теперь мы можем осуществить простое наследование, передавая родительские классы классу Class:

```
var Animal = new Class;

Animal.include({
  breath: function(){
    console.log('дыхание');
  }
```

```
});

var Cat = new Class(Animal)

// Использование
var tommy = new Cat;
tommy.breath();
```

Вызов функции

Как и все остальное в JavaScript, функции являются просто объектами. Но, в отличие от других объектов, их можно вызывать. Содержимое функции, т. е. значение this, зависит от того, где и как она вызвана.

Кроме использования скобок, есть еще два способа вызова функции: apply() и call(). Разница между ними заключается в аргументах, передаваемых функции.

Функция apply() получает два аргумента: контекст и массив аргументов. Если контекст имеет нулевое значение, используется глобальный контекст. Например:

```
function.apply(this, [1, 2, 3])
```

У функции call() точно такое же поведение, но она используется по-другому. Первым аргументом является контекст, а вот каждый последующий аргумент делегируется вызову. Иными словами, для передачи аргументов функции используется не массив, как в случае с apply(), а несколько аргументов.

```
function.call(this, 1, 2, 3);
```

Зачем может понадобиться изменение контекста? Это резонный вопрос, потому что другие языки неплохо обходятся и без предоставления возможности явных изменений контекста. В JavaScript изменение контекста применяется для совместного использования состояния, особенно в процессе внешнего вызова функций обработки событий. (Лично мне это представляется ошибкой в конструкции языка, поскольку это может сбивать с толку новичков. Но теперь уже слишком поздно что-либо менять, остается только изучить, как это работает.)

Библиотека jQuery пользуется возможностями изменения контекста с помощью функций apply() и call() во многих местах своего API, например при использовании обработчиков событий или при последовательном переборе с помощью функции each(). Поначалу это может сбить с толку, но, разобравшись с происходящим, из всего можно извлечь пользу:

```
$('.clicky').click(function(){
  // 'this' ссылается на элемент
  $(this).hide();
});
$('p').each(function(){
  // 'this' ссылается на текущую итерацию
  $(this).remove();
});
```

Для обращения к исходному контексту часто применяется прием сохранения значения `this` в локальной переменной. Например:

```
var clicky = {
  wasClicked: function(){
    /* ... */
  },

  addListeners: function(){
    var self = this;
    $('.clicky').click(function(){
      self.wasClicked()
    });
  }
};
clicky.addListeners();
```

Но чтобы прояснить эту ситуацию, мы можем воспользоваться `apply`, заключая внешний вызов в другую безымянную функцию, которая сохраняет исходный контекст:

```
var proxy = function(func, thisObject){
  return(function(){
    return func.apply(thisObject, arguments);
  });
};

var clicky = {
  wasClicked: function(){
    /* ... */
  },

  addListeners: function(){
    var self = this;
    $('.clicky').click(proxy(this.wasClicked, this));
  }
};
```

В показанном выше примере мы задали контекст, используемый внутри функции внешнего вызова `click`; контекст jQuery, вызывающий функцию, игнорируется. Фактически в API библиотеки jQuery входит кое-что, чтобы сделать абсолютно то же самое (как вы уже, наверное, догадались, это `jQuery.proxy()`):

```
$('.clicky').click($.proxy(function(){ /* ... */ }, this));
```

Есть и другие веские причины для использования `apply()` и `call()` — например делегирование. Мы можем делегировать вызовы одной функции другой и даже изменять переданные аргументы:

```
var App {
  log: function(){
    if (typeof console == "undefined") return;
```

```
    // Превращение аргументов в надлежащий массив
    var args = jQuery.makeArray(arguments);

    // Вставка нового аргумента
    args.unshift("(App)");

    // Делегирование вызова консоли
    console.log.apply(console, args);
  }
};
```

В приведенном выше коде мы создали массив аргументов, затем добавили свой собственный аргумент, и, наконец, вызов был делегирован `console.log()`. Если вы не знакомы с переменной `arguments`, то значение ей присваивает интерпретатор, и в ней содержится массив аргументов, с которыми была вызвана текущая область действия. Но это не настоящий массив, например он не может быть изменен, поэтому нам нужно превратить его в нечто более полезное с помощью `jquery.makeArray()`.

Управление областью действия в нашей библиотеке классов

Прокси-функция, описанная в предыдущем разделе, является настолько полезным образцом, что мы должны добавить ее к нашей библиотеке классов. Мы добавим прокси-функцию как к классам, так и к экземплярам классов, предоставив себе возможность сохранять область действия класса, передавая функции обработчикам событий и т. п.:

```
var Class = function(parent){
  var klass = function(){
    this.init.apply(this, arguments);
  };
  klass.prototype.init = function(){};
  klass.fn = klass.prototype;

  // Добавление прокси-функции
  klass.proxy = function(func){
    var self = this;
    return(function(){
      return func.apply(self, arguments);
    });
  }

  // Добавление функции также и к экземплярам
  klass.fn.proxy = klass.proxy;
  return klass;
};
```

Теперь функцию proxy() можно использовать для охвата функций, обеспечивая тем самым их вызов в нужной области действия:

```
var Button = new Class;

Button.include({
  init: function(element){
    this.element = jQuery(element);

    // Передача области действия функции щелчка (click)
    this.element.click(this.proxy(this.click));
  },

  click: function(){ /* ... */ }
});
```

Если мы не охватим вызов внешней функции click() с помощью прокси-функции, она будет вызвана в контексте this.element, а не в контексте Button, что приведет к разного рода проблемам. В новой спецификации JavaScript — ECMAScript, пятое издание (ES5) — также добавлена поддержка для управления области действия вызова с помощью функции привязки bind(). Эта функция вызывается в отношении другой функции, гарантируя ее вызов в контексте указанного значения this. Например:

```
Button.include({
  init: function(element){
    this.element = jQuery(element);

   // привязка функции click
    this.element.click(this.click.bind(this));
  },

  click: function(){ /* ... */ }
});
```

Этот пример является эквивалентом нашей функции proxy() и обеспечивает вызов функции click() в правильном контексте. Устаревшие браузеры не поддерживают bind(), но поддержка при необходимости может быть легко вставлена и реализована своими силами. Вставляемая прослойка реализует уровень совместимости на устаревших браузерах, расширяя соответствующие свойства объектов и позволяя тем самым использовать возможности ES5 уже сегодня, не обращая внимание на устаревшие браузеры. Например, прослойка, поддерживающая bind(), будет иметь следующий вид:

```
if ( !Function.prototype.bind ) {
  Function.prototype.bind = function( obj ) {
    var slice = [].slice,
        args = slice.call(arguments, 1),
        self = this,
        nop = function () {},
```

```
        bound = function () {
          return self.apply( this instanceof nop ? this : ( obj || {} ),
                             args.concat( slice.call(arguments) ) );
        };

    nop.prototype = self.prototype;

    bound.prototype = new nop();

    return bound;
  };
}
```

Принадлежащее `Function` свойство `prototype` переписывается, только если нужная возможность не существует: новейшие браузеры будут по-прежнему использовать свои собственные реализации. Вставка прослоек (shimming) особенно полезна для массивов, которые получили массу новых свойств, добавленных в последних версиях JavaScript. Лично я использую проект *es5-shim*, потому что он охватывает максимально возможное количество нововведений, появившихся в ES5.

Добавление закрытых функций

До сих пор любое свойство, добавляемое к нашим классам, было открыто всему миру и могло быть изменено в любой момент. А теперь давайте исследуем возможность добавления к нашим классам закрытых свойств.

Многие разработчики склонны использовать для закрытых свойств префикс в виде знака подчеркивания (_). Хотя эта склонность все еще может претерпеть изменения, она является явным свидетельством принадлежности к части закрытого API. Я пытаюсь избегать такого подхода, поскольку он выглядит довольно уродливо.

В JavaScript имеется поддержка неизменяемых свойств, но она не реализована на основных браузерах, поэтому нам приходится ждать, когда можно будет воспользоваться этим методом. Вместо него для создания закрытой области действия, имеющей только внутренний доступ, мы воспользуемся безымянной функцией JavaScript:

```
var Person = function(){};

(function(){

  var findById = function(){ /* ... */ };

  Person.find = function(id){
    if (typeof id == "integer")
      return findById(id);
  };
})();
```

Мы заключаем все свойства нашего класса в безымянную функцию, затем создаем локальные переменные (`findById`), которые могут быть доступны только в текущей области действия. Переменная `Person` определена в глобальной области действия, поэтому к ней можно обращаться из любого места.

Никогда не нужно определять переменную без использования оператора `var`, поскольку она всегда будет создаваться в качестве глобальной. Если нужно определить глобальную переменную, это следует сделать в глобальной области действия или в качестве свойства, принадлежащего окну:

```
(function(exports){
  var foo = "bar";

  // Превращение переменной в видимую
  exports.foo = foo;
})(window);
assertEqual(foo, "bar");
```

Библиотеки классов

При наличии в данной книге большого количества понятий, неплохо бы разобраться с теорией, лежащей в основе классов, но зачастую на практике вы будете пользоваться библиотекой. jQuery изначально не имеет поддержки классов, но она легко может быть добавлена с помощью дополнительного модуля HJS. Этот модуль позволяет определить классы, передавая набор свойств функции `$.Class.create`:

```
var Person = $.Class.create({
  // конструктор
  initialize: function(name) {
    this.name = name;
  }
});
```

Для наследования классов нужно при их создании передать им родительский класс:

```
var Student = $.Class.create(Person, {
  price: function() { /* ... */ }
});
var alex = new Student("Alex");
alex.pay();
```

Для добавления свойств класса их нужно присвоить непосредственно классу:

```
Person.find = function(id){ /* ... */ };
```

API дополнительного модуля HJS включает ряд полезных функций, таких как `clone()` и `equal()`:

```
var alex = new Student("Alex");
var bill = alex.clone();
assert( alex.equal(bill) );
```

Но HJS не является единственным выбором — реализация класса есть также и в дополнительном модуле Spine. Для его использования нужно просто включить в страницу файл spine.js:

```
<script src="http://maccman.github.com/spine/spine.js"> </script>
<script>
  var Person = Spine.Class.create();

  Person.extend({
    find: function() { /* ... */ }
  });

  Person.include({
    init: function(atts){
      this.attributes = atts || {};
    }
  });
  var person = Person.init();
</script>
```

В библиотеке классов Spine есть API, похожий на имеющийся у той библиотеки, которую мы разрабатывали при изучении материалов данной главы. Для добавления свойств класса следует использовать функцию `extend()`, а для добавления свойств экземпляра следует использовать функцию `include()`. Для того чтобы унаследовать свойства, нужно передать родительские классы методу создания экземпляра `Spine.Class`.

Если расширять свой кругозор за пределы jQuery, определенно следует обратить внимание на библиотеку *Prototype*. У нее имеется превосходный *API класса*, вдохновивший на создание множества других библиотек.

У создателя jQuery Джона Рейсига (John Resig) есть интересная статья по реализации *классического наследования с помощью библиотеки* (classical inheritance with the library). Ее стоит прочитать, особенно если вы интересуетесь особенностями, положенными в основу системы прототипов JavaScript.

2 События и наблюдение

События положены в основу вашего JavaScript-приложения, являясь движущей силой происходящего и предоставляя первую точку соприкосновения при взаимодействии пользователя с вашим приложением. Но именно в них проявляется вся уродливая сторона нестандартизированного рождения JavaScript. На пике браузерных войн Netscape и Microsoft преднамеренно выбрали разные, несовместимые модели событий. Хотя позже они были стандартизированы консорциумом W3C, Internet Explorer сохранил свою отличную от других реализацию вплоть до своего самого последнего выпуска IE9.

К счастью, у нас есть такие великолепные библиотеки, как jQuery и Prototype, которые устраняют путаницу, предоставляя вам единый API, который будет работать со всеми реализациями модели событий. Тем не менее все же стоит разобраться с тем, что происходит за сценой, поэтому перед показом примеров для различных популярных библиотек я собираюсь рассмотреть модель, принятую консорциумом W3C.

Прослушивание событий

Все события вращаются вокруг функции, которая называется `addEventListener()` и получает три аргумента: тип события (например, `click`), слушатель, т. е. функцию внешнего (обратного) вызова, и аргумент `useCapture` (на котором мы остановимся чуть позже). Используя первые два аргумента, мы можем прикрепить к DOM-элементу функцию, вызываемую, когда в отношении этого элемента произойдет конкретное событие, например щелчок на нем кнопкой мыши:

```
var button = document.getElementById("createButton");

button.addEventListener("click", function(){ /* ... */ }, false);
```

Слушатель можно удалить, воспользовавшись функцией `removeEventListener()` и передав ей те же самые аргументы, которые использовались при вызове функции `addEventListener()`. Если функция-слушатель была безымянной и на нее нет ссылок, она не может быть удалена без удаления самого элемента:

```
var div = document.getElementById("div");

var listener = function(event) { /* ... */ };
div.addEventListener("click", listener, false);
div.removeEventListener("click", listener, false);
```

В качестве первого аргумента функции-слушателю (listener) передается объект события (event), который можно использовать для получения информации о событии, например метки времени, координат и целевого объекта. В нем также содержатся различные функции для остановки распространения события и предотвращения действия по умолчанию.

Что касается типов событий, то их поддержка варьируется от браузера к браузеру, но все современные браузеры поддерживают следующие события:

- `click` (щелчок)
- `dblclick` (двойной щелчок)
- `mousemove` (перемещение указателя мыши)
- `mouseover` (прохождение указателя мыши над объектом)
- `mouseout` (выход указателя мыши за пределы объекта)
- `focus` (получение объектом фокуса)
- `blur` (потеря объектом фокуса)
- `change` (изменение, касающееся ввода данных в форму)
- `submit` (отправка данных формы)

Полную таблицу совместимости событий можно найти на веб-ресурсе Quirksmode.

Порядок обработки событий

Прежде чем пойти дальше, важно рассмотреть порядок обработки событий. Если элемент и один из его прародителей имеет обработчик для одного и того же типа события, который из них должен быть активирован первым при наступлении события? Наверное, вы не удивитесь, узнав, что у Netscape и у Microsoft на этот счет были разные идеи.

В Netscape 4 была реализована поддержка *захвата* событий, инициирующего слушателей событий, начиная с самого старшего элемента-прародителя и заканчивая рассматриваемым элементом, т. е. снаружи вовнутрь.

Компанией Microsoft была реализована поддержка *всплытия* событий, распространяя вызов слушателей событий от элемента вверх, через его прародителей, т. е. изнутри наружу.

Я считаю, что во всплытии событий больше смысла и оно, вероятно, станет моделью, используемой в повседневной разработке. Консорциум W3C пошел на компромисс и оговорил в своей спецификации поддержку обеих моделей событий. События, в соответствии с моделью W3C, сначала проходят стадию захвата, пока не достигнут целевого элемента, а затем они опять всплывают.

Можно выбрать тип требуемого регистрируемого обработчика события по признаку захвата или всплытия, и здесь на сцену выходит аргумент `useCapture` функции `addEventListener()`. Если последний аргумент, передаваемый функции `addEventListener()`, имеет значение `true`, обработчик событий настраивается на

фазу захвата, а если он имеет значение `false`, обработчик события настраивается на фазу всплытия:

```
// Использование всплытия путем передачи последним аргументом значения false
button.addEventListener("click", function(){ /* ... */ }, false);
```

Основную часть времени, скорее всего, будет использоваться всплытие. Если вы в этом будете не уверены, передайте функции `addEventListener()` в качестве последнего аргумента значение `false`.

Отмена событий

При всплытии события его продвижение можно остановить, воспользовавшись функцией `stopPropagation()`, вызванной в отношении объекта события. Тем самым будет предотвращен вызов любых обработчиков, принадлежащих элементам-прародителям:

```
button.addEventListener("click", function(e){
  e.stopPropagation();
  /* ... */
}, false);
```

Кроме этого, такие библиотеки, как jQuery, поддерживают функцию `stopImmediatePropagation()`, полностью препятствующую вызову любых последующих обработчиков, даже если они относятся к тому же самому элементу.

Браузеры также назначают событиям действия по умолчанию. Например, при щелчке на ссылке действием, назначенным браузером по умолчанию, будет загрузка новой страницы. При щелчке на поле флажка браузер устанавливает этот флажок. Эти действия по умолчанию выполняются после фаз распространения события и могут быть отменены в процессе любой из этих обработок события. Действию по умолчанию можно воспрепятствовать с помощью функции `preventDefault()`, вызванной в отношении объекта события. Вместо этого можно вернуть из обработчика значение `false`:

```
form.addEventListener("submit", function(e){
  /* ... */
  return confirm("Вы абсолютно уверены?");
}, false);
```

Если вызов функции `confirm()` вернет `false`, т. е. если пользователь щелкнет на кнопке отмены в диалоговом окне подтверждения, функция внешнего вызова, обрабатывающая событие, вернет `false`, отменяя событие и отправку формы.

Объект события

Как и вышеупомянутые функции — `stopPropagation()` и `preventDefault()`, — объект события содержит массу полезных свойств. Большинство свойств, описан-

ных в спецификации W3C, рассмотрены ниже, а для дополнительных сведений следует обратиться к полной спецификации.

Тип события:

bubbles

Булево значение, свидетельствующее о всплытии события вверх по дереву DOM (объектной модели документа)

Свойства, отображающие состояние среды окружения при наступлении события:

button

Значение, показывающее, которая (или которые) из кнопок мыши была нажата (если вообще была нажата какая-либо из кнопок)

ctrlKey

Булево значение, показывающее, была ли нажата клавиша Ctrl

altKey

Булево значение, показывающее, была ли нажата клавиша Alt

shiftKey

Булево значение, показывающее, была ли нажата клавиша Shift

metaKey

Булево значение, показывающее, была ли нажата клавиша Meta

Свойства, относящиеся к событиям клавиатуры:

isChar

Булево значение, показывающее, относится ли событие к нажатию алфавитно-цифровой клавиши

charCode

Значение нажатой клавиши в формате Юникод (только для событий нажатия клавиш — `keypress`)

keyCode

Значение нажатой клавиши в формате Юникод для несимвольных клавиш

which

Значение нажатой клавиши в формате Юникод независимо от того, является ли она символьной

Место, где произошло событие:

pageX, pageY

Координаты события относительно страницы (т. е. области просмотра)

screenX, screenY

Координаты события относительно экрана

Элементы, связанные с событием:

currentTarget

Текущий DOM-элемент в фазе всплытия события

target, originalTarget

Исходный DOM-элемент

relatedTarget

Другой DOM-элемент, вовлеченный в событие (если таковой имеется)

Эти свойства в браузерах, особенно несовместимых с требованиями W3C, могут варьироваться. К счастью, такие библиотеки, как jQuery и Prototype, сгладят любые различия.

Библиотеки событий

В конечном итоге для управления событиями вы, вероятнее всего, воспользуетесь библиотекой JavaScript, иначе вам придется сталкиваться с многочисленными случаями несогласованности браузеров. Я собираюсь показать вам, как использовать API управления событиями библиотеки jQuery, хотя существует множество других хороших вариантов выбора, таких как Prototype, MooTools и YUI. Для получения более подробной документации нужно обратиться к API, соответствующим этим библиотекам.

Для добавления кроссбраузерных слушателей событий в API библиотеки jQuery имеется функция `bind()`. Эту функцию нужно вызвать в отношении экземпляров класса `jQuery`, передавая ей имя события и функцию его обработки:

```
jQuery("#element").bind(eventName, handler);
```

Например, зарегистрировать обработчик щелчка на элементе можно следующим образом:

```
jQuery("#element").bind("click", function(event) {
  // ...
});
```

В jQuery имеются сокращения для таких типов событий, как щелчок (click), отправка (submit) и прохождение указателя мыши (mouseover). Такое сокращение имеет следующий вид:

```
$("#myDiv").click(function(){
  // ...
});
```

Важно заметить, что элемент должен существовать до начала добавления к нему события, т. е. добавление нужно осуществлять после того, как произойдет загрузка страницы. Для этого нужно всего лишь прислушаться к событию загрузки окна (load) и затем приступить к добавлению слушателей:

```
jQuery(window).bind("load", function() {
  $("#signinForm").submit(checkForm);
});
```

Но лучше прислушиваться не к событию загрузки окна, а к событию загрузки содержимого DOM — `DOMContentLoaded`. Оно наступает при готовности модели DOM, но до того как завершится загрузка имеющихся на странице изображений и таблиц стилей. Это означает, что событие всегда будет наступать до того, как пользователь сможет взаимодействовать со страницей.

Событие `DOMContentLoaded` поддерживается не всяким браузером, поэтому jQuery получает его с помощью функции `ready()`, имеющей кроссбраузерную поддержку:

```
jQuery.ready(function($){
  $("#myForm").bind("submit", function(){ /* ... */ });
});
```

Фактически функцию `ready()` можно опустить и передать обработчик непосредственно объекту jQuery:

```
jQuery(function($){
  // Вызывается, когда страница будет готова к работе
});
```

Изменение контекста

Одним из вопросов, вокруг которого часто возникает путаница, является способ изменения контекста при вызове обработчика. При использовании принадлежащей браузеру функции `addEventListener()` локальный контекст заменяется контекстом целевого HTML-элемента:

```
new function(){
  this.appName = "wem";
  document.body.addEventListener("click", function(e){
    // Контекст был изменен, поэтому appName не будет определен
    alert(this.appName);
  }, false);
};
```

Для сохранения исходного контекста нужно заключить обработчик в безымянную функцию, сохранив на нее ссылку. Эта схема уже рассматривалась в главе 1, где мы использовали прокси-функцию для поддержки текущего контекста. Это настолько широко востребованная схема, что в jQuery включена функция `proxy()`, ей нужно только передать функцию и контекст, в котором эту функцию нужно вызвать:

```
$("signinForm").submit($.proxy(function(){ /* ... */ }, this));
```

Делегирование событий

Возможно, вы уже сталкивались с такими случаями, когда при проверке наступления события на дочерних элементах можно воспользоваться всплытием события и добавить слушатель только к их родительскому элементу. Именно такая

технология используется в таких структурах, как SproutCore, для сокращения в приложении количества слушателей событий:

```
// Делегирование событий списку ul
list.addEventListener("click", function(e){
  if (e.currentTarget.tagName == "li") {
    /* ... */
    return false;
  }
}, false);
```

В jQuery для этого имеется очень хороший способ: нужно просто передать функции `delegate()` селектор дочернего элемента, тип события и обработчик. Альтернативой этому подходу может стать добавление события `click` каждому элементу `li`. Но, используя `delegate()`, вы сокращаете количество слушателей событий, улучшая производительность программы:

```
// Не делайте этого! слушатель будет добавлен к каждому 'li' (а это затратно)
$("ul li").click(function(){ /* ... */ });

// Этот код добавит только один слушатель события
$("ul").delegate("li", "click", /* ... */);
```

Еще одно преимущество делегирования события заключается в том, что любой дочерний элемент, добавленный в динамическом режиме к родительскому элементу, уже будет иметь слушатель события. Поэтому в показанном выше примере любые элементы `li`, добавленные к списку после загрузки страницы, все равно будут вызывать обработчик события `click`.

Пользовательские события

Кроме событий, присущих браузеру, можно инициировать и привязывать свои собственные события. Вне всякого сомнения, это отличный способ проектирования библиотек, именно такой схемой пользуются многие дополнительные модули библиотеки jQuery. Спецификация W3C, касающаяся пользовательских событий, производителями браузеров была дружно проигнорирована, поэтому для реализации этого свойства нужно пользоваться такими библиотеками, как jQuery или Prototype.

jQuery позволяет запускать пользовательские события, используя функцию `trigger()`. Вы можете указать пространство имен для имен событий, но эти пространства имен должны быть разделены точками и указаны в обратном порядке. Например:

```
// Привязка пользовательского события
$(".class").bind("refresh.widget", function(){});

// Запуск пользовательского события
$(".class").trigger("refresh.widget");
```

Для передачи данных обработчику события их нужно просто передать в качестве дополнительного аргумента функции `trigger()`. Данные будут отправлены функции внешнего вызова в качестве дополнительных аргументов:

```
$(".class").bind("frob.widget", function(event, dataNumber){
  console.log(dataNumber);
});

$(".class").trigger("frob.widget", 5);
```

Подобно обычным событиям, пользовательские события будут распространяться вверх по дереву DOM.

Пользовательские события и дополнительные модули jQuery

Пользовательские события, часто используемые для повышения эффективности в дополнительных модулях jQuery, являются отличным способом для разработки любых фрагментов логики, взаимодействующей с DOM. Если вы незнакомы с дополнительными модулями jQuery, перейдите к Приложению Б, включающему букварь jQuery.

При добавлении какой-либо функциональной возможности к своему приложению всегда нужно прикидывать, нельзя ли реализующий ее фрагмент кода извлечь и выделить в дополнительный модуль. Это поможет уменьшить количество связей и создать готовую к повторному использованию библиотеку.

Например, давайте посмотрим на простой дополнительный модуль jQuery для поддержания вкладок. Мы собираемся получить `ul`-список, реагирующий на события щелчка (`click`). Когда пользователь щелкает на элементе списка, мы будем добавлять к этому элементу класс `active`, удаляя класс `active` из других элементов списка:

```
<ul id="tabs">
  <li data-tab="users">Users</li>
  <li data-tab="groups">Groups</li>
</ul>

<div id="tabsContent">
  <div data-tab="users"> ... </div>
  <div data-tab="groups"> ... </div>
</div>
```

Кроме этого, у нас имеется `div`-контейнер содержимого вкладок `tabsContent`, который содержит текущее наполнение вкладок. Мы также будем добавлять и удалять класс `active` из дочерних элементов `div`-контейнера, в зависимости от того, на какой вкладке был сделан щелчок. Текущее отображение и скрытие вкладок

будет осуществляться с помощью CSS, а наш дополнительный модуль будет только переключать класс active:

```
jQuery.fn.tabs = function(control){
  var element = $(this);
  control = $(control);

  element.find("li").bind("click", function(){
    // Добавление класса active к элементу списка и удаление из него этого
класса
    element.find("li").removeClass("active");
    $(this).addClass("active");

    // Добавление класса active к tabContent и удаление из него этого класса
    var tabName = $(this).attr("data-tab");
    control.find(">[data-tab]").removeClass("active");
    control.find(">[data-tab='" + tabName + "']").addClass("active");
  });

  // Активация первой вкладки
  element.find("li:first").addClass("active");

  // Возвращение 'this' чтобы допустить выстраивание цепочки
  return this;
};
```

Дополнительный модуль является свойством prototype класса jQuery, поэтому он может быть вызван в отношении экземпляров класса jQuery:

```
$("ul#tabs").tabs("#tabContent");
```

Что на данный момент сделано в дополнительном модуле не так? Обработчик события щелчка был добавлен ко всем элементам списка, и это было нашей первой ошибкой. Вместо этого нужно было воспользоваться ранее рассмотренной в этой главе функцией delegate(). К тому же поскольку обработчик щелчка слишком большой, трудно понять, что происходит. Более того, если другой разработчик захочет расширить дополнительный модуль, ему, наверное, придется его переписать.

Давайте посмотрим, как можно применить пользовательские события, чтобы сделать наш код чище. Мы запустим событие change.tabs при щелчке на вкладке и привяжем несколько обработчиков, чтобы изменить соответствующим образом значение active для атрибута класс:

```
jQuery.fn.tabs = function(control){
  var element = $(this);
  control = $(control);

  element.delegate("li", "click", function(){
    // Извлечение имени вкладки
    var tabName = $(this).attr("data-tab");
```

```
    // Запуск пользовательского события при щелчке на вкладке
    element.trigger("change.tabs", tabName);
  });

  // Привязка к пользовательскому событию
  element.bind("change.tabs", function(e, tabName){
    element.find("li").removeClass("active");
    element.find(">[data-tab='" + tabName + "']").addClass("active");
  });

  element.bind("change.tabs", function(e, tabName){
    control.find(">[data-tab]").removeClass("active");
    control.find(">[data-tab='" + tabName + "']").addClass("active");
  });

  // Активация первой вкладки
  var firstName = element.find("li:first").attr("data-tab");
  element.trigger("change.tabs", firstName);
  return this;
};
```

Видите, насколько чище стал код с применением пользовательского события? Это означает, что мы можем выделить обработчики смены вкладок, что добавит преимуществ, существенно повышая возможности расширения дополнительного модуля. Например, теперь мы можем программным способом сменить вкладки путем запуска нашего события change.tabs в отношении просматриваемого списка:

```
$("#tabs").trigger("change.tabs", "users");
```

Мы также могли бы связать вкладки с хэшем окна, добавляя поддержку кнопки возврата:

```
$("#tabs").bind("change.tabs", function(e, tabName){
  window.location.hash = tabName;
});
$(window).bind("hashchange", function(){
  var tabName = window.location.hash.slice(1);
  $("#tabs").trigger("change.tabs", tabName);
});
```

Сам факт использования пользовательских событий дает другим разработчикам широкие возможности расширения нашей работы.

Элементы, не имеющие отношения к модели DOM

Программирование, основанное на событиях, может быть весьма эффективным, поскольку оно разъединяет архитектуру вашего приложения, приводя к улучше-

нию обособленности и обслуживаемости компонентов. Но события не ограничиваются только рамками объектной модели документа (DOM), можно без особого труда создать свою собственную библиотеку обработчиков. Такая схема называется *публикацией-подпиской*, и с ней стоит познакомиться.

Публикация-подписка (Publish/Subscribe, или Pub/Sub) — это схема работы с сообщениями, имеющая две категории исполнителей: издателей и подписчиков. Издатели публикуют сообщения в конкретном канале, а подписчики подписываются на каналы, получая уведомления при публикации новых сообщений. Ключевая особенность этой схемы состоит в том, что издатели и подписчики совершенно разобщены, они даже не подозревают о существовании друг друга. Единственное, чем они совместно пользуются — это имя канала.

Разобщенность издателей и подписчиков позволяет вашему приложению расширяться без введения большого объема взаимозависимостей и обобщений, упрощая обслуживание и добавление дополнительных свойств.

Итак, как же приспособить Pub/Sub к приложению? Для этого нужно всего лишь написать обработчики, связанные с именем события, а затем получить способ их вызова. Ниже приводится объект PubSub, который можно использовать для добавления и запуска слушателей событий:

```
var PubSub = {
  subscribe: function(ev, callback) {
    // Создание объекта _callbacks, если только он уже не существует
    var calls = this._callbacks || (this._callbacks = {});

    // Создание массива для заданного ключа события, если только он уже не
    // существует, а затем добавление внешнего вызова к массиву
    (this._callbacks[ev] || (this._callbacks[ev] = [])).push(callback);
    return this;
  },

  publish: function() {
    // Превращение аргументов объекта в настоящий массив
    var args = Array.prototype.slice.call(arguments, 0);

    // Извлечение первого аргумента (имени события)
    var ev   = args.shift();

    // Возврат управления, если это не объект _callbacks или
    // если он не содержит массив для заданного события
    var list, calls, i, l;
    if (!(calls = this._callbacks)) return this;
    if (!(list  = this._callbacks[ev])) return this;

    // Вывоз внешних функций
    for (i = 0, l = list.length; i < l; i++)
      list[i].apply(this, args);
    return this;
```

```
  }
};

// Пример использования
PubSub.subscribe("wem", function(){
  alert("Wem!");
});
PubSub.publish("wem");
```

Можно указать пространство имен событий, воспользовавшись таким разделителем, как двоеточие (`:`).

```
PubSub.subscribe("user:create", function(){ /* ... */ });
```

Если используется jQuery, то в ней есть еще более простая библиотека, созданная Беном Алманом (Ben Alman). Она настолько простая, что ее можно просто поместить в программу:

```
/*!
 * jQuery Tiny Pub/Sub - v0.3 - 11/4/2010
 * http://benalman.com/
 *
 * Copyright (c) 2010 "Cowboy" Ben Alman
 * Dual licensed under the MIT and GPL licenses.
 * http://benalman.com/about/license/
 */

(function($){
  var o = $({});

  $.subscribe = function() {
    o.bind.apply( o, arguments );
  };

  $.unsubscribe = function() {
    o.unbind.apply( o, arguments );
  };

  $.publish = function() {
    o.trigger.apply( o, arguments );
  };
})(jQuery);
```

Ее API получает те же самые аргументы, что и функции jQuery `bind()` и `trigger()`. Единственное отличие заключается в том, что функции размещены непосредственно в объекте `jQuery` и называются `publish()` и `subscribe()`:

```
$.subscribe( "/some/topic", function( event, a, b, c ) {
  console.log( event.type, a + b + c );
});

$.publish( "/some/topic", "a", "b", "c" );
```

Мы использовали Pub/Sub для глобальных событий, но для этой схемы нетрудно определить и область действия. Давайте возьмем ранее созданный объект PubSub и определим область его действия в пределах объекта:

```
var Asset = {};

// Добавление PubSub
jQuery.extend(Asset, PubSub);

// Теперь у нас есть функции publish и subscribe
Asset.subscribe("create", function(){
  // ...
});
```

Для копирования в наш объект Asset свойств, принадлежащих объекту PubSub, мы воспользовались методом extend(), принадлежащим объекту jQuery. Теперь все вызовы publish() и subscribe() ограничены пределами объекта Asset. Такой прием пригодится во многих сценариях, включая события в объектно-реляционном отображении (ORM), изменения в конечном автомате или внешние (обратные) вызовы по завершении Ajax-запроса.

3 Модели и данные

Одной из проблем перемещения состояния на сторону клиента является управление данными. Традиционно можно извлечь данные непосредственно из базы данных в процессе запроса страницы, вставляя результат непосредственно в страницу в результате взаимодействия сервера и клиента. Но управление данными в структурированных JavaScript-приложениях является совершенно другим процессом. Здесь нет модели запрос-ответ, и у вас нет доступа к переменным на стороне сервера. Вместо этого данные извлекаются удаленно и временно сохраняются на стороне клиента.

Хотя такой переход может создать определенные трудности, он принесет ряд преимуществ. Например, доступ к данным на стороне клиента происходит практически мгновенно, поскольку данные просто извлекаются из памяти. Это может действительно изменить интерфейс вашего приложения — реакция на любое действие с приложением будет мгновенной, что зачастую существенно улучшает пользовательские впечатления от работы с ним.

Тут следует подумать над тем, как выстроить архитектуру хранилища данных на стороне клиента. В этой области легко просчитаться и впасть в заблуждение, что характерно для малоопытных разработчиков, особенно если их приложения становятся крупнее. В данной главе будут рассмотрены лучшие способы организации такого перехода и даны некоторые рекомендованные схемы и инструкции.

MVC и организация пространства имен

Обеспечение в вашем приложении четких границ между представлениями, состоянием и данными играет основную роль в поддержании порядка и жизнеспособности его архитектуры. Если придерживаться модели MVC, управление данными осуществляется в моделях (буква «M» в MVC). Модели должны быть отделены от представлений и контроллеров. Любая логика, связанная с работой с данными и поведением, должна находиться в моделях и иметь правильную организацию пространства имен.

В JavaScript можно определить пространство имен функций и переменных, сделав их свойством объекта. Например:

```
var User = {
  records: [ /* ... */ ]
};
```

Для массива пользователей определено должное пространство имен под свойством `User.records`. Функции, связанные с пользователями, также могут быть заключены в пространство имен модели `User`. Например, у нас может быть функция `fetchRemote()` для извлечения пользовательских данных из сервера:

```
var User = {
  records: [],
  fetchRemote: function(){ /* ... */ }
};
```

Содержание всех свойств модели в пространстве имен гарантирует отсутствие каких-либо конфликтов и совместимость этой модели со схемой MVC. Это также оберегает ваш код от скатывания вниз к мешанине из функций и внешних (обратных) вызовов.

В организации пространства имен можно пойти еще дальше и хранить любые функции, относящиеся к экземплярам пользователей, в фактических объектах пользователей. Представим, что у нас есть функция `destroy()` для работы с записями пользователей. Она ссылается на конкретного пользователя, стало быть, она должна быть в экземпляре `User`:

```
var user = new User;
user.destroy()
```

Чтобы добиться этого, нам нужно сделать `User` классом, а не обычным объектом:

```
var User = function(atts){
  this.attributes = atts || {};
};
User.prototype.destroy = function(){
  /* ... */
};
```

Любые функции и переменные, не относящиеся к конкретным пользователям, могут быть свойствами, непосредственно относящимися к объекту `User`:

```
User.fetchRemote = function(){
  /* ... */
};
```

Чтобы получить дополнительную информацию об организации пространства имен, посетите блог Питера Мишо (Peter Michaux), где можно прочитать статью на эту тему.

Создание ORM

Средства объектно-реляционного отображения (ORM) обычно используются в языках, отличающихся от JavaScript. И тем не менее это очень полезная технология для управления данными, а также отличный способ использования моделей в вашем JavaScript-приложении. Используя ORM, можно, к примеру, связать модель с удаленным сервером — любые изменения, внесенные в экземпляры модели, вызовут отправку в фоновом режиме Ajax-запросов к серверу. Или же можно связать экземпляр модели с элементом HTML — любые изменения, относящиеся к экземпляру, будут отражаться в представлении. Все это будет конкретизировано в примерах, а сейчас давайте посмотрим на создание специализированного ORM.

В конечном итоге ORM — это всего лишь объектный уровень, заключающий в себя некие данные. Обычно средства ORM используются для реферирования баз данных SQL, но в нашем случае ORM будет всего лишь реферированием типов данных JavaScript. Преимущество наличия этого дополнительного уровня состоит в том, что мы можем улучшить исходные данные расширенной функциональностью путем добавления наших собственных специализированных функций и свойств. Это позволит нам добавить такие ценные качества, как проверка допустимости, использование наблюдателей, обеспечение постоянства и использование серверных обратных вызовов, сохраняя при этом возможность повторного использования большого объема кода.

Прототипное наследование

Для создания нашего ORM мы собираемся воспользоваться функцией `Object.create()`, которая немного отличается от рассмотренных в главе 1 примеров, основанных на классах. Это позволит нам вместо функций-конструкторов и ключевого слова `new` использовать наследование, основанное на прототипах.

Функции `Object.create()` передается один аргумент, объект-прототип, а она возвращает новый объект с указанным объектом-прототипом. Иными словами, вы даете ей объект, а она возвращает новый объект, унаследованный от указанного вами объекта.

Функция `Object.create()` была недавно добавлена к ECMAScript, пятое издание, поэтому в некоторых браузерах, таких как IE, она не реализована. Но это не проблема, поскольку мы при необходимости запросто можем добавить ее поддержку:

```
if (typeof Object.create !== "function")
    Object.create = function(o) {
      function F() {}
      F.prototype = o;
      return new F();
    };
```

Показанный выше пример был позаимствован у Дугласа Крокфорда (Douglas Crockford) из его статьи «Prototypal Inheritance». Почитайте эту статью, если вы хотите глубже изучить основы прототипов и наследования в JavaScript.

Мы собираемся создать объект `Model`, который будет отвечать за создание новых моделей и экземпляров:

```
var Model = {
  inherited: function(){},
  created: function(){},

  prototype: {
    init: function(){}
  },

  create: function(){
    var object = Object.create(this);
    object.parent = this;
    object.prototype = object.fn = Object.create(this.prototype);

    object.created();
    this.inherited(object);
    return object;
  },

  init: function(){
    var instance = Object.create(this.prototype);
    instance.parent = this;
    instance.init.apply(instance, arguments);
    return instance;
  }
};
```

Если вы незнакомы с функцией `Object.create()`, этот код может показаться отпугивающим, поэтому давайте разберем его по частям. Функция `create()` возвращает новый объект, унаследованный у объекта `Model`, — мы воспользуемся этим для создания новых моделей. Функция `init()` возвращает новый объект, унаследованный у `Model.prototype`, — т. е. экземпляр объекта `Model`:

```
var Asset = Model.create();
var User  = Model.create();

var user = User.init();
```

Добавление свойств ORM

Теперь, если мы добавим свойства к `Model`, они будут доступны во всех унаследованных моделях:

```
// Добавление свойств объекта
jQuery.extend(Model, {
  find: function(){}
});
```

```
// Добавление свойств экземпляра
jQuery.extend(Model.prototype, {
  init: function(atts) {
    if (atts) this.load(atts);
  },

  load: function(attributes){
    for(var name in attributes)
      this[name] = attributes[name];
  }
});
```

Функция `jQuery.extend()` является всего лишь более коротким способом использования цикла `for` для самостоятельного копирования с перебором всех свойств, аналогично тому, что мы делали в функции `load()`. Теперь наши свойства объекта и экземпляра распространяются вниз на наши отдельные модели:

```
assertEqual( typeof Asset.find, "function" );
```

Фактически мы собираемся добавить множество свойств, поэтому мы можем также сделать `extend()` и `include()` частью объекта `Model`:

```
var Model = {
  /* ... фрагмент ... */

  extend: function(o){
    var extended = o.extended;
    jQuery.extend(this, o);
    if (extended) extended(this);
  },

  include: function(o){
    var included = o.included;
    jQuery.extend(this.prototype, o);
    if (included) included(this);
  }
};

// Добавление свойств объекта
Model.extend({
  find: function(){}
});

// Добавление свойств экземпляра
Model.include({
  init: function(atts) { /* ... */ },
  load: function(attributes){ /* ... */ }
});
```

Теперь мы можем создать новые активы (assets) и установить атрибуты:

```
var asset = Asset.init({name: "foo.png"});
```

Удерживание записей

Нам нужен способ удерживания записей, т. е. сохранения ссылки на созданные экземпляры, чтобы потом иметь к ним доступ. Мы сделаем это с использованием объекта `records`, установленного в отношении `Model`. При сохранении экземпляра мы будем добавлять его к этому объекту, а при удалении экземпляров мы будем удалять их из этого объекта:

```
// Объект сохраненных активов
Model.records = {};

Model.include({
  newRecord: true,

  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this;
  },

  destroy: function(){
    delete this.parent.records[this.id];
  }
});
```

А что насчет обновления существующего экземпляра? Это нетрудно, достаточно обновить ссылку на объект:

```
Model.include({
  update: function(){
    this.parent.records[this.id] = this;
  }
});
```

Давайте создадим удобную функцию для сохранения экземпляра, чтобы нам не приходилось проверять, был ли экземпляр ранее сохранен и нужно ли его создавать:

```
// Сохранение объекта в хэше записей с сохранением ссылки на него
Model.include({
  save: function(){
    this.newRecord ? this.create() : this.update();
  }
});
```

А как насчет реализации функции `find()`, чтобы можно было искать активы по ID?

```
Model.extend({
  // Поиск по ID или выдача исключения
  find: function(id){
    return this.records[id] || throw("Неизвестная запись ");
  }
});
```

Теперь, преуспев в создании основного ORM, давайте испытаем его в работе:

```
var asset  = Asset.init();
asset.name = "same, same";
asset.id   = 1
asset.save();

var asset2  = Asset.init();
asset2.name = "but different";
asset2.id   = 2;
asset2.save();

assertEqual( Asset.find(1).name, "same, same" );
asset2.destroy();
```

Добавление поддержки ID

Пока при каждом сохранении записи нам нужно указывать ID самостоятельно. Нехорошо, но, к счастью, именно это мы можем автоматизировать. Сначала нам нужно найти способ генерирования идентификаторов – это можно сделать с помощью GUID-генератора (Globally Unique Identifier – глобально уникальный идентификатор). Итак, технически JavaScript не способен генерировать официальные полноценные 128-разрядные GUID-идентификаторы для использования в API, он может только генерировать псевдослучайные числа. Генерирование по-настоящему случайных GUID-идентификаторов является печально известной и весьма трудной задачей, и операционная система вычисляет их, используя MAC-адрес, позицию мыши и контрольные суммы BIOS, путем измерения электрического шума или радиоактивного распада, и даже с помощью ламп со вплывающими и тонущими в жидкости каплями вязкого вещества, постоянно меняющими свои формы! Но нам вполне хватит и возможностей имеющейся в JavaScript функции `Math.random()`.

Роберт Киффер (Robert Kieffer) написал легкий и совсем небольшой GUID-генератор, использующий `Math.random()` для генерации псевдослучайных GUID-идентификаторов. Он настолько прост, что его можно включить непосредственно в текст программы:

```
Math.guid = function(){
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g,
function(c) {
    var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
    return v.toString(16);
  }).toUpperCase();
};
```

Теперь, когда у нас есть функция для генерации GUID-идентификаторов, ее интеграция в наш ORM не составит труда. Для этого нужно лишь изменить функцию `create()`:

```
Model.extend({
  create: function(){
    if ( !this.id ) this.id = Math.guid();
    this.newRecord = false;
    this.parent.records[this.id] = this;
  }
});
```

Теперь любые только что созданные записи имеют в качестве ID GUID-идентификаторы:

```
var asset = Asset.init();
asset.save();
asset.id //=> "54E52592-313E-4F8B-869B-58D61F00DC74"
```

Адресация ссылок

Если пристально приглядеться к нашему ORM, в нем можно заметить ошибку, связанную со ссылками. Мы не создаем копии экземпляров, когда они возвращаются функцией `find()` или когда мы их сохраняем, поэтому, если мы изменили любые свойства, они изменяются на исходном активе.

Проблема в том, что обновление активов нам требуется только при вызове функции `update()`:

```
var asset = new Asset({name: "foo"});
asset.save();

// Утверждение соответствует истине
assertEqual( Asset.find(asset.id).name, "foo" );

// Давайте изменим свойство, но не будем вызывать update()
asset.name = "wem";

// Надо же! Это утверждение дает ложный результат, поскольку теперь актив
// называется "wem"
assertEqual( Asset.find(asset.id).name, "foo" );
```

Давайте исправим ситуацию, создав новый объект в процессе операции `find()`. Нам также нужно создать дубликат объекта при каждом создании или обновлении записи:

```
Asset.extend({
  find: function(id){
    var record = this.records[id];
    if ( !record ) throw("Неизвестная запись");
    return record.dup();
  }
```

```
});

Asset.include({
  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this.dup();
  },

  update: function(){
    this.parent.records[this.id] = this.dup();
  },

  dup: function(){
    return jQuery.extend(true, {}, this);
  }
});
```

Есть еще одна проблема: `Model.records` является объектом, который совместно используется каждой моделью:

```
assertEqual( Asset.records, Person.records );
```

Это имеет досадный побочный эффект смешивания всех записей:

```
var asset = Asset.init();
asset.save();

assert( asset in Person.records );
```

Решение состоит в создании нового объекта `records` при каждом создании новой модели. Функция `Model.created()` является внешним вызовом для создания нового объекта, поэтому мы можем создавать любые объекты, которые там определены для модели:

```
Model.extend({
  created: function(){
    this.records = {};
  }
});
```

Загрузка в данные

Если ваше веб-приложение не работает исключительно в пределах браузера, вам нужно загружать в него удаленные данные с сервера. Обычно поднабор данных загружается при запуске приложения, а дополнительные данные загружаются после взаимодействия пользователя с приложением. В зависимости от типа приложения и объема данных, вы можете загрузить все необходимое при загрузке начальной страницы. Это идеальный вариант, поскольку пользователям никогда не приходится ждать загрузки дополнительных данных. Но для многих приложений это нереально, поскольку у них слишком много данных и они не могут поместиться в памяти браузера.

Предварительная загрузка данных является ключевым условием для создания у пользователей ощущения гладкости и быстроты работы вашего приложения, и для сведения к минимуму любого времени ожидания. Но есть разумная грань между предварительной загрузкой действительно нужных данных и загрузкой излишних данных, которые никогда не будут использованы. Необходимо предсказать, данные какого сорта будут востребованы вашими пользователями (или воспользоваться показателями по результатам использования вашего приложения).

Почему бы при отображении списка, разбитого на страницы, не провести предварительную загрузку следующей страницы для обеспечения мгновенных переходов? Или, лучше того, просто вывести длинный список и автоматически загружать и вставлять данные по мере его прокрутки (шаблон бесконечной прокрутки). Чем меньше пользователь будет ждать, тем лучше.

При извлечении новых данных нужно убедиться в том, что пользовательский интерфейс не заблокирован. Выведите какой-нибудь индикатор загрузки, но при этом гарантируйте возможность пользования интерфейсом. Сценарии, блокирующие пользовательский интерфейс, должны быть сведены к минимуму или вообще не использоваться.

Данные должны быть представлены в составе кода начальной страницы или загружаться с помощью отдельных HTTP-запросов с применением Ajax или JSONP. Лично я хочу порекомендовать последние две технологии, поскольку включение большого объема данных в состав кода увеличивает размер страницы, в то время как загрузка с помощью параллельных запросов осуществляется быстрее. AJAX и JSON также позволяют вам кэшировать HTML-страницу, вместо динамического вывода по каждому запросу.

Включение данных в код страницы

Я не приветствую данный подход в силу причин, высказанных в предыдущем абзаце, но в определенных ситуациях он может пригодиться, особенно для загрузки очень небольших объемов данных. Преимуществом данной технологии является простота реализации.

Для этого нужно лишь вывести JSON-объект непосредственно в страницу. Например, в Ruby on Rails это делается следующим образом:

```
<script type="text/javascript">
  var User = {};
  User.records = <%= raw @users.to_json %>;
</script>
```

ERB-теги используются для вывода JSON-интерпретации данных пользователя. Метод `raw` используется просто для приостановки нейтрализации JSON. При выводе страницы HTML выглядит следующим образом:

```
<script type="text/javascript">
  var User = {};
  User.records = [{"first_name": "Alex"}];
</script>
```

JavaScript может просто обработать JSON, поскольку этот формат имеет такую же структуру, что и объект JavaScript.

Загрузка данных с помощью Ajax

Наверное, это первый метод загрузки удаленных данных, который приходит в голову, когда слышишь о фоновых запросах, и для этого есть веская причина: он опробован, протестирован и поддерживается во всех современных браузерах. Но это не говорит о том, что Ajax лишен недостатков — его нетипичная история вылилась в несогласованный API и в сложности загрузки данных из разных доменов, связанные с браузерной безопасностью.

Если вам нужен краткий учебник по Ajax и по классу XMLHttpRequest, прочитайте статью «Getting Started» на сайте разработчиков Mozilla. Но, по всей видимости, вы, в конце концов, остановитесь на использовании такой библиотеки, как jQuery, которая абстрагирует Ajax API, стирая разницу между браузерами. Именно поэтому мы рассматриваем здесь API библиотеки jQuery, а не класс XMLHttpRequest как таковой.

Имеющийся в jQuery Ajax API состоит из одной низкоуровневой функции `jQuery.ajax()` и нескольких ее абстракций, находящихся на более высоком уровне, сокращающих объем набираемого кода. Функция `jQuery.ajax()` получает кроме всего прочего хэш установок для параметров запроса, тип содержимого и функции обратных вызовов. Как только функция будет вызвана, запрос будет асинхронно отправлен в фоновом режиме.

url

Запрашиваемый URL. По умолчанию — текущая страница.

success

Функция, вызываемая при удачном завершении запроса. В качестве аргумента ей передаются любые данные, возвращенные сервером.

contentType

Устанавливает для запроса заголовок Content-Type. Если запрос содержит данные, то по умолчанию устанавливается заголовок `application/x-www-form-urlencoded`, который подходит для большинства случаев.

data

Данные, отправляемые на сервер. Если они еще не в строковом виде, jQuery их преобразует в последовательную форму и закодирует в URL.

type

Используемый HTTP-метод: GET, POST или DELETE. По умолчанию используется метод GET.

dataType

Тип данных, ожидаемых с сервера. jQuery нужно это знать, чтобы понимать, что делать с результатом. Если `dataType` не указан, jQuery выстроит догадку на основе MIME-типа полученного ответа. Поддерживаются следующие значения:

text

Ответ в виде простого текста; обработка не требуется.

script

jQuery обрабатывает ответ как JavaScript.

json

jQuery обрабатывает ответ как JSON, используя строгий анализатор.

jsonp

Для JSONP-запросов, подробное рассмотрение которых нам еще предстоит.

Давайте, к примеру, сделаем простой Ajax-запрос, который выводит извещение о любых данных, возвращенных сервером:

```
jQuery.ajax({
  url: "/ajax/endpoint",
  type: "GET",
  success: function(data) {
    alert(data);
  }
});
```

Но все эти настройки являются слишком многословными. К счастью, в jQuery имеется ряд сокращений. Функция `jQuery.get()` получает URL, а также необязательные данные и функцию обратного вызова:

```
jQuery.get("/ajax/endpoint", function(data){
  $(".ajaxResult").text(data);
});
```

Или же если нужно отправить ряд параметров запроса с помощью GET-метода:

```
jQuery.get("/ajax/endpoint", {foo: "bar"}, function(data){
  /* ... */
});
```

Если с сервера ожидается возвращение данных в формате JSON, нужно вместо этого вызвать функцию `jQuery.getJSON()`, которая установит для параметра `dataType` запроса значение `"json"`:

```
jQuery.getJSON("/json/endpoint", function(json){
  /* ... */
});
```

Кроме этого есть еще функция `jQuery.post()`, которая также получает URL, данные и функцию обратного вызова:

```
jQuery.post("/users", {first_name: "Alex"}, function(result){
  /* Ajax POST завершен успешно */
});
```

Если нужно воспользоваться другими HTTP-методами — DELETE, HEAD и OPTIONS, — придется воспользоваться имеющей более низкий уровень функцией `jQuery.ajax()`.

Этот был краткий обзор Ajax API, имеющегося в jQuery, но если нужно получить дополнительную информацию, следует обратиться к полной документации.

Ограничением для Ajax служит *политика исходного источника* (same origin policy), которая ограничивает запросы тем же доменом, субдоменом и портом, что и у адреса страницы, с которой сделаны эти запросы. На это есть веские причины: при отправке Ajax-запроса с ним вместе отправляется вся информация из cookie-файлов домена. Это означает, что для удаленного сервера запрос окажется от вошедшего в систему пользователя. Без применения политики исходного источника злоумышленник потенциально может скопировать все ваши адреса электронной почты из Gmail, обновить ваш статус на Facebook или направить сообщение читателям вашего Твиттера, что считается существенной прорехой в системе безопасности.

Но когда политика исходного источника встроена в систему безопасности Интернета, это также создает неудобства для тех разработчиков, которые пытаются обратиться к допустимым удаленным ресурсам. В других технологиях, например в Adobe Flash и в Java, были реализованы методы, позволяющие обойти проблему междоменных файлов политики, и теперь Ajax связан стандартом, который называется междоменным совместным использованием исходных ресурсов — CORS (cross-origin resource sharing).

CORS позволяет преодолеть барьер политики исходного источника, предоставляя доступ к авторизованным удаленным серверам. Эта спецификация полностью поддерживается всеми серьезными браузерами, поэтому если вы не пользуетесь IE6, все будет в порядке.

CORS поддерживается следующими браузерами:

- IE >= 8 (с оговорками)
- Firefox >= 3
- Safari: полная поддержка
- Chrome: полная поддержка
- Opera: не поддерживает

Использовать CORS довольно просто. Если нужно получить авторизованный доступ к вашему серверу, следует просто добавить несколько строк к HTTP-заголовку возвращенных ответов:

```
Access-Control-Allow-Origin: example.com
Access-Control-Request-Method: GET,POST
```

Показанный выше заголовок проведет авторизацию междоменных GET- и POST-запросов с example.com. Несколько значений нужно отделять друг от друга запятыми, как это было в нашем примере со значениями `GET,POST`. Чтобы

открыть доступ с дополнительных доменов, их нужно просто перечислить в заголовке `Access-Control-Allow-Origin`, используя в качестве разделителей запятые. Можно открыть доступ для любого домена, просто указав для заголовка источника (origin) звездочку (*).

Некоторые браузеры, например Safari, сначала сделают запрос OPTIONS, чтобы проверить допустимость запроса. А вот Firefox сделает запрос, и если CORS-заголовки не установлены, просто выдаст защитное исключение. Эту разницу в поведении нужно будет учесть на стороне сервера.

Можно даже авторизовать заголовки специализированного запроса, воспользовавшись заголовком `Access-Control-Request-Headers`:

```
Access-Control-Request-Headers: Authorization
```

Это означает, что клиенты могут добавить к Ajax-запросам специализированные заголовки, например давая запросу подпись открытого протокола авторизации OAuth:

```
var req = new XMLHttpRequest();
req.open("POST", "/endpoint", true);
req.setRequestHeader("Authorization", oauth_signature);
```

К сожалению, несмотря на то что CORS работает с версиями Internet Explorer от 8 и выше, Microsoft решила проигнорировать спецификацию и рекомендации *рабочей группы*. Microsoft создала свой собственный объект, `XDomainRequest`, который также предназначался для использования вместо `XMLHttpRequest` в случае междоменных запросов. Хотя его интерфейс аналогичен интерфейсу `XMLHttpRequest`, у него есть ряд *препятствий и ограничений*. Например, работают только методы GET и POST, не поддерживаются аутентифицирующие или специализированные заголовки, и наконец, что совсем неожиданно, поддерживаются только заголовки «Content-Type: text/plain». Если вы готовы мириться с такими ограничениями, то, используя правильные заголовки `Access-Control`, вы можете воспользоваться CORS при работе в IE.

JSONP

Технология JSONP, или JSON с подкладкой (padding), была создана до стандартизации технологии CORS и представляет собой еще один способ извлечения данных из удаленного сервера. Идея заключается в наличии тега `script`, указывающего на конечный пункт JSON, где возвращенные данные заключаются в вызов функции. Теги `script` не являются поводом для каких-либо междоменных ограничений, и эта технология поддерживается практически в каждом браузере.

Итак, здесь у нас есть тег `script`, указывающий на наш удаленный сервер:

```
<script src="http://example.com/data.json"> </script>
```

Затем конечный пункт, data.json, возвращает JSON-объект, заключенный в вызов функции:

```
jsonCallback({"data": "foo"})
```

Затем мы определяем функцию с глобальным доступом. Как только сценарий будет загружен, будет вызвана эта функция:

```
window.jsonCallback = function(result){
  // Заполнение результата
}
```

И все же это не простой процесс. К счастью, jQuery оформляет его в кратком API:

```
jQuery.getJSON("http://example.com/data.json?callback=?", function(result)
{
  // Заполнение результата
});
```

jQuery ставит вместо последнего вопросительного знака в показанном выше URL случайным образом выбранное имя временной функции, создаваемой этой библиотекой. Вашему серверу нужно прочитать аргумент вызова обратной функции и использовать его в качестве имени возвращаемой функции, в которую заключаются данные.

Безопасность при использовании междоменных запросов

Если вы открываете свой сервер для междоменных запросов или JSONP из любого домена, нужно серьезно подумать о безопасности. Обычно политика междоменных запросов не дает злоумышленнику вызвать, скажем, API Твиттера и извлечь ваши персональные данные. CORS и JSONP полностью изменяют положение вещей. При обычном Ajax-запросе все cookie-файлы вашей сессии отправляются вместе с запросом, значит, вы будете зарегистрированы в API Твиттера. Любые потенциальные злоумышленники получают полный контроль над вашей учетной записью, поэтому вопросы безопасности выходят на первый план.

Если при использовании CORS/JSONP вы не указываете, какой домен может получить доступ к вашему API, нужно брать в расчет ряд ключевых обстоятельств:

- не показывайте никакой ценной информации, к примеру, адреса электронной почты;
- не разрешайте никаких действий (например, «отслеживания» Твиттера).

В качестве альтернативных мер просто создайте исчерпывающий список доменов, которым разрешено подключение, или же воспользуйтесь аутентификацией с помощью открытого протокола авторизации OAuth.

Заполнение нашей модели ORM

Заполнение нашей модели ORM данными является совсем несложным процессом. Для этого нужно всего лишь извлечь данные с сервера, после чего обновить записи нашей модели. Давайте добавим к объекту `Model` функцию `populate()`,

которая будет осуществлять последовательный перебор всех заданных значений, создавать экземпляры объекта и обновлять объект records:

```
Model.extend({
  populate: function(values){
    // Перезапуск model и records
    this.records = {};
    for (var i=0, il = values.length; i < il; i++) {
      var record = this.init(values[i]);
      record.newRecord = false;
      this.records[record.id] = record;
    }
  }
});
```

Теперь функцию Model.populate() можно использовать с результатами нашего запроса на получение данных:

```
jQuery.getJSON("/assets", function(result){
  Asset.populate(result);
});
```

Теперь любые возвращенные сервером данные будут доступны в нашей ORM.

Локальное хранение данных

В прошлом вопрос локального хранилища данных решался нелегко. Единственным доступным вариантом было использование cookie-файлов и дополнительных модулей типа Adobe Flash. Cookie-файлы имели устаревший API и не могли хранить большие объемы данных, а кроме того, они отправляли все данные с каждым запросом назад серверу, увеличивая объем ничем неоправданных издержек. Что касается Flash, то давайте по возможности будем избегать применения дополнительных модулей.

К счастью, поддержка локального хранилища была включена в HTML5, и она реализована во всех основных браузерах. В отличие от cookie-файлов, данные хранятся исключительно на стороне клиента и никогда не отправляются на сервера. Кроме того, вы можете хранить значительно больший объем данных, максимальные значения которого зависят от конкретного браузера (и, как показано ниже, от его версии), но все они предлагают не менее 5 Мб на каждый домен:

- IE >= 8
- Firefox >= 3.5
- Safari >= 4
- Chrome >= 4
- Opera >= 10.6

Хранилища HTML5 подпадают под спецификацию HTML5 интернет-хранилища (HTML5 Web Storage) и состоят из двух типов: *локальное хранилище* (local

storage) и *хранилище сессии* (session storage). Локальное хранилище существует и после закрытия браузера, а хранилище сессии существует только на время существования окна. Любые сохраненные данные имеют область определения в пределах домена и доступны только тем сценариям, которые изначально их сохраняли.

Вы можете получить доступ к локальному хранилищу и хранилищу сессии и работать с ними, используя, соответственно, объекты `localStorage` и `sessionStorage`. API очень похож на задание значений свойствам объекта JavaScript и, за исключением двух объектов, идентичен как для локального хранилища, так и для хранилища сессии:

```
// Установка значения
localStorage["someData"] = "wem";
```

У API интернет-хранилища есть ряд дополнительных свойств:

```
// Количество хранящихся элементов
var itemsStored = localStorage.length;

// Установка элемента (является псевдонимом хэш-синтаксиса)
localStorage.setItem("someData", "wem");

// Получение сохраненного элемента, возвращение null, если элемент
неизвестен
localStorage.getItem("someData"); //=> "wem";

// Удаление элемента, возвращение null, если элемент неизвестен
localStorage.removeItem("someData");

// Удаление всех элементов (очистка хранилища)
localStorage.clear();
```

Данные хранятся в виде строк, поэтому если вы собираетесь сохранять объекты или целые числа, преобразование нужно будет провести собственными силами. Чтобы сделать это с помощью JSON, перед сохранением объектов переведите их в последовательную JSON-форму (сериализуйте их), а затем при их извлечении снова превратите JSON-строки в объекты:

```
var object = {some: "object"};

// Сериализация и сохранение объекта
localStorage.setItem("seriData", JSON.stringify(object));

// Загрузка и обратное преобразование объекта
var result = JSON.parse(localStorage.getItem("seriData"));
```

Если будет превышена квота хранения (обычно она составляет 5 Мб на хост), при сохранении дополнительных данных будет выдана ошибка `QUOTA_EXCEEDED_ERR`.

Добавление локального хранилища к нашей модели ORM

Давайте добавим поддержку локального хранилища к нашей модели ORM, чтобы записи могли существовать между обновлениями страницы. Чтобы использовать объект `localStorage`, нам нужно сериализовать наши записи для получения JSON-строки. Проблема в том, что в настоящее время сериализованные объекты имеют следующий вид:

```
var json = JSON.stringify(Asset.init({name: "foo"}));
json //=> "{"parent":{"parent":{"prototype":{}},"records":[]},"name":"foo"}"
```

Итак, нам нужно откорректировать JSON-сериализацию наших моделей. Сначала нужно определить, какие свойства следует превращать в последовательную форму. Давайте добавим к объекту `Model` массив свойств, который отдельные модели могут использовать для задания своих атрибутов:

```
Model.extend({
  created: function(){
    this.records = {};
    this.attributes = [];
  }
});
Asset.attributes = ["name", "ext"];
```

Поскольку у каждой модели имеются разные атрибуты, в силу чего одна и та же ссылка на массив совместно использоваться не может, свойство `attributes` не было установлено непосредственно в объекте `Model`. Вместо этого мы создаем новый массив при создании модели, подобно тому, что мы делали с объектом `records`.

Теперь давайте создадим функцию `attributes()`, которая будет возвращать объект атрибутов в виде значений:

```
Model.include({
  attributes: function(){
    var result = {};
    for(var i in this.parent.attributes) {
      var attr = this.parent.attributes[i];
      result[attr] = this[attr];
    }
    result.id = this.id;
    return result;
  }
});
```

Теперь мы можем установить массив атрибутов для каждой модели:

```
Asset.attributes = ["name", "ext"];
```

и функция `attributes()` будет возвращать объект с нужными свойствами:

```
var asset = Asset.init({name: "document", ext: ".txt"});
asset.attributes(); //=> {name: "document", ext: ".txt"};
```

Что же касается замены метода `JSON.stringify()`, то нам нужно всего лишь заполучить метод `toJSON()` в экземплярах модели. Библиотека JSON будет использовать эту функцию для поиска объекта сериализации, а не для сериализации объекта `records` в его исходном виде:

```
Model.include({
  toJSON: function(){
    return(this.attributes());
  }
});
```

Давайте попробуем превратить записи в последовательную форму еще раз. Теперь получившаяся JSON-строка будет содержать нужные нам свойства:

```
var json = JSON.stringify(Asset.records);
json //= "{"7B2A9E8D...":"{"name":"document","ext":".
txt","id":"7B2A9E8D..."}"}"
```

После того как мы добились нужной работы от JSON-сериализации, добавление в наши модели локального хранилища особого труда не составит. Добавим в наш объект `Model` две функции: `saveLocal()`и `loadLocal()`. При сохранении мы будем превращать объект `Model.records` в массив, переводить его в последовательную форму и отправлять в `localStorage`:

```
var Model.LocalStorage = {
  saveLocal: function(name){
    // Превращение записей в массив
    var result = [];
    for (var i in this.records)
      result.push(this.records[i])
    localStorage[name] = JSON.stringify(result);
  },
  loadLocal: function(name){
    var result = JSON.parse(localStorage[name]);
    this.populate(result);
  }
};
Asset.extend(Model.LocalStorage);
```

Наверное, будет неплохо считывать записи из локального хранилища при загрузке страницы и сохранять их при закрытии страницы. Но решение этой задачи мы оставим в качестве упражнения для читателя.

Отправка новых записей на сервер

Ранее уже рассматривался вопрос использования jQuery-функции `post()` для отправки данных на сервер. Функции передаются три аргумента: конечный URL-адрес, запрашиваемые данные и функция обратного вызова:

```
jQuery.post("/users", {first_name: "Alex"}, function(result){
  /* Ajax POST увенчался успехом */
});
```

Теперь, при наличии функции `attributes()`, записи на сервере создаются довольно просто, нужно просто отправить POST-запрос с атрибутами записей:

```
jQuery.post("/assets", asset.attributes(), function(result){
  /* Ajax POST увенчался успехом */
});
```

Если соблюдать REST-соглашения, нужно отправлять HTTP POST-запрос при создании записи и PUT-запрос при ее обновлении. Давайте добавим к экземплярам объекта `Model` две функции: `createRemote()` и `updateRemote()`, которые будут отправлять на наш сервер HTTP-запрос нужного типа:

```
Model.include({
  createRemote: function(url, callback){
    $.post(url, this.attributes(), callback);
  },
  updateRemote: function(url, callback){
    $.ajax({
      url:     url,
      data:    this.attributes(),
      success: callback,
      type:    "PUT"
    });
  }
});
```

Теперь если вызвать `createRemote()` в отношении экземпляра `Asset`, его атрибуты будут переданы в виде POST-запроса на сервер:

```
// Порядок использования:
Asset.init({name: "jason.txt"}).createRemote("/assets");
```

4 Контроллеры и состояния

Исторически сложилось так, что состояние обрабатывалось на стороне сервера с помощью cookie-файлов сессии. Поэтому при переходе пользователей на новую страницу состояние предыдущей страницы утрачивалось, и продолжали существовать только cookie-файлы. Но JavaScript-приложения сводятся к одной странице, следовательно, теперь мы можем сохранять состояние в памяти клиентской машины.

Одним из самых весомых преимуществ сохранения состояния на клиентской машине заключается в четко реагирующем интерфейсе. Пользователь получает немедленную реакцию при взаимодействии со страницей, и ему не приходится ждать загрузки следующей страницы в течение нескольких секунд. Скорость существенно улучшает пользовательские восприятия, делая многие JavaScript-приложения приятными в использовании.

Но хранение состояния на стороне клиента вызывает вопросы. Где именно оно должно храниться? В локальных переменных? Может быть, в DOM? Правильное хранение данных является одним из самых важных вопросов, в решении которого нельзя ошибаться.

В первую очередь нужно избегать хранения данных или состояния в DOM. Это весьма скользкий путь, ведущий к путанице и анархии! В нашем случае, поскольку мы используем испытанную на практике архитектуру MVC, состояние хранится внутри контроллеров нашего приложения.

А что такое контроллер? Его можно представить в виде своеобразного клея между представлениями и моделями. Это единственный компонент, знающий о существовании представлений и моделей и связывающий их вместе. При загрузке страницы ваш контроллер подключает обработчики событий к представлениям и обрабатывает соответствующие обратные вызовы, взаимодействуя по необходимости с моделями.

Для создания контроллеров никакие библиотеки не нужны, хотя они могут оказаться полезными. Важными характеристиками контроллеров являются модульность и независимость. В идеале в них не должны определяться глобальные переменные, контроллеры должны функционировать как четко обособленные компоненты. Лучше всего обеспечить это с помощью модульной схемы.

Модульная схема

Модульная схема является отличным способом инкапсуляции логики и предотвращения загрязнения глобального пространства имен. Все это становится возможным за счет использования безымянных функций, которые, возможно, являются единственным лучшим свойством JavaScript. Мы будем просто создавать безымянные функции и тут же запускать их на выполнение. Весь код, помещающийся внутри функции, запускается внутри замкнутого выражения, обеспечивая локальную и закрытую среду окружения для переменных нашего приложения:

```
(function(){
  /* ... */
})();
```

Нам нужно заключить безымянную функцию в круглые скобки () перед тем, как мы сможем ее выполнить. JavaScript требует этого, чтобы правильно интерпретировать инструкции.

Глобальный импорт

Определения переменных внутри модуля являются локальными, поэтому к ним невозможно получить доступ за его пределами в глобальном пространстве имен. Но все глобальные переменные приложения остаются доступными, и внутри модуля к ним можно совершенно свободно обращаться и манипулировать ими. Зачастую непонятно, какие именно глобальные переменные используются в модуле, особенно когда модули становятся крупнее.

Кроме того, предполагаемые глобальные переменные медленнее работают, поскольку интерпретатор JavaScript должен для работы с ними пройти по всей цепочке области действия имен. Доступ к локальным переменным всегда будет быстрее и эффективнее.

К счастью, наши модули предоставляют легкий способ решения этих проблем. Передавая глобальные переменные в качестве параметров нашим безымянным функциям, мы можем импортировать их в наш код, который становится понятнее и быстрее, чем при использовании подразумеваемых глобальных переменных:

```
(function($){
  /* ... */
})(jQuery);
```

В показанном выше примере мы импортировали глобальную переменную `jQuery` в наш модуль, присвоив ей псевдоним `$`. Становится понятно, к каким глобальным переменным происходит обращение внутри модуля, и, кроме того, ускоряется их работа. Фактически использование присущего jQuery сокращения `$` *рекомендуется* как средство обеспечения бесконфликтной работы вашего кода с другими библиотеками.

Глобальный экспорт

Сходную технологию мы можем использовать и при экспортировании глобальных переменных. В идеале нужно использовать как можно меньше глобальных

переменных, но всегда бывает какой-нибудь неординарный случай, когда в них случается потребность. Мы можем импортировать переменную страницы `window` в наш модуль, напрямую установить в нем значения свойств, выставляя тем самым переменные в глобальное пространство имен:

```
(function($, exports){
  exports.Foo = "wem";
})(jQuery, window);
assertEqual( Foo, "wem" );
```

Факт использования переменной, называемой `exports`, для установки значения любой глобальной переменной делает код понятнее и выявляет те глобальные переменные, которые создаются модулем.

Добавление контекста

Использование локального контекста является полезным способом структурирования модулей, особенно когда дело касается регистрации обратных вызовов для событий. В данных условиях контекст внутри нашего модуля является глобальным — `this` приравнивается к `window`:

```
(function(){
  assertEqual( this, window );
})();
```

Если нужно образовать область видимости контекста, следует приступить к добавлению функций к объекту. Например:

```
(function(){
  var mod = {};

  mod.contextFunction = function(){
    assertEqual( this, mod );
  };

  mod.contextFunction();
})();
```

Теперь контекст внутри `contextFunction()` является локальным по отношению к нашему объекту `mod`. Мы можем приступить к использованию `this`, не волнуясь о создании глобальных переменных. Чтобы лучше понять, как это могло бы быть использовано на практике, давайте проведем подробный разбор следующего примера:

```
(function($){

  var mod = {};

  mod.load = function(func){
    $($.proxy(func, this));
  };
```

продолжение ⇨

```
  mod.load(function(){
    this.view = $("#view");
  });

  mod.assetsClick = function(e){
    // Обработка щелчка
  };
  mod.load(function(){
    this.view.find(".assets").click(
      $.proxy(this.assetsClick, this)
    );
  });
})(jQuery);
```

Мы создаем функцию `load()`, которой передается функция обратного вызова, выполняемая ею после загрузки страницы. Обратите внимание, что для обеспечения запуска функции обратного вызова в нужном контексте используется `jQuery.proxy()`.

Затем, при загрузке страницы мы добавляем в элемент обработчик щелчка, передавая ему в качестве функции обратного вызова локальную функцию `assetsClick()`. Не нужно излишне усложнять контроллер. Важно, чтобы все состояния контроллера оставались локальными и четко инкапсулировались в модуле.

Абстрагирование в библиотеку

Давайте абстрагируем все это в библиотеку, чтобы можно было повторно использовать ее с другими модулями и контроллерами. Мы включим туда существующую функцию `load()` и добавим новые функции `proxy()` и `include()`:

```
(function($, exports){
  var mod = function(includes){
    if (includes) this.include(includes);
  };
  mod.fn = mod.prototype;

  mod.fn.proxy = function(func){
    return $.proxy(func, this);
  };

  mod.fn.load = function(func){
    $(this.proxy(func));
  };

  mod.fn.include = function(ob){
    $.extend(this, ob);
  };

  exports.Controller = mod;
})(jQuery, window);
```

Функция `proxy()` обеспечивает выполнение функций в локальном контексте, что является весьма полезной схемой для функций обратного вызова, привязанных к событиям. Функция `include()` является простым сокращением для добавления свойств в контроллер, экономя время на набор кода.

Мы добавляем нашу библиотеку к объекту `exports`, выставляя ее в качестве глобальной переменной `Controller`. Внутри модуля мы можем создать экземпляр объекта `Controller`, используя его функцию-конструктор. Давайте разберем простой пример, который переключает `class`-атрибут элемента при прохождении над этим элементом указателя мыши:

```
(function($, Controller){

  var mod = new Controller;

  mod.toggleClass = function(e){
    this.view.toggleClass("over", e.data);
  };

  mod.load(function(){
    this.view = $("#view");
    this.view.mouseover(this.proxy(this.toggleClass), true);
    this.view.mouseout(this.proxy(this.toggleClass), false);
  });
})(jQuery, Controller);
```

Когда страница загружается, мы создаем переменную `view` и подключаем ряд слушателей событий. Они, в свою очередь, вызывают `toggleClass()`, когда указатель мыши проходит над элементом, переключая `class`-атрибут элемента. Полную версию примера можно увидеть в файлах, сопровождающих эту книгу, а именно в файле **assets/ch04/modules.html**.

При использовании контекста вместо локальных переменных приходится набирать больше кода, базирующегося на использовании ключевого слова `this`. Но такая технология расширяет область повторного использования кода и включения миксинов. Например, мы можем добавить функцию в каждый экземпляр объекта `Controller`, установив значение свойства в отношении его прототипа:

```
Controller.fn.unload = function(func){
  jQuery(window).bind("unload", this.proxy(func));
};
```

Или же мы можем расширить отдельный контроллер путем использования ранее определенной функции `include()`, передавая ей объект:

```
var mod = new Controller;
mod.include(StateMachine);
```

Используемый в данном примере объект `StateMachine` может быть повторно использован множество раз с другими нашими модулями, исключая дублирование кода и сохраняя все в рамках принципа don't repeat yourself — DRY (никогда не повторяться).

Загрузка контроллеров после документа

Получается, что некоторые части наших контроллеров загружаются еще до формирования DOM, а другие их части находятся в функциях обратного вызова, которые вызываются после загрузки документа страницы. Это может привести к путанице, поскольку логика контроллера выполняется под различными состояниями, что приводит к большому количеству возвратных вызовов при загрузке документа.

Мы можем решить эту проблему путем загрузки контроллеров *после* формирования DOM. Я лично ратую за такой подход, поскольку он гарантирует, что при обращении к элементам вам не придется постоянно думать о том, в каком состоянии находится DOM страницы.

Давайте сначала применим все это к нашей библиотеке, немного улучшив наши контролеры. Класс `Controller` не должен быть функцией-конструктором, поскольку здесь будет излишним переключение контекста, необходимое при генерировании подчиненных контроллеров:

```
// Использование вместо объекта window глобального контекста для создания
// глобальных переменных
var exports = this;

(function($){
  var mod = {};

  mod.create = function(includes){
    var result = function(){
      this.init.apply(this, arguments);
    };

    result.fn = result.prototype;
    result.fn.init = function(){};

    result.proxy = function(func){ return $.proxy(func, this); };
    result.fn.proxy = result.proxy;

    result.include = function(ob){ $.extend(this.fn, ob); };
    result.extend = function(ob){ $.extend(this, ob); };
    if (includes) result.include(includes)

    return result;
  };

  exports.Controller = mod;
})(jQuery);
```

Теперь мы можем воспользоваться нашей новой функцией `Controller.create()` для создания контроллеров, передав ей литерал объекта свойств экземпляра. Обратите внимание, что весь контроллер заключен в `jQuery(function(){ /* ... */ })`.

Это псевдоним для `jQuery.ready()`, и он гарантирует, что контроллер загружается только после полного формирования DOM страницы:

```
jQuery(function($){
  var ToggleView = Controller.create({
    init: function(view){
      this.view = $(view);
      this.view.mouseover(this.proxy(this.toggleClass), true);
      this.view.mouseout(this.proxy(this.toggleClass), false);
    },

    this.toggleClass: function(e){
      this.view.toggleClass("over", e.data);
    }
  });

  // Instantiate controller, calling init()
  new ToggleView("#view");
});
```

Другим существенным изменением является передача контроллеру элемента представления в ходе создания экземпляра вместо его жесткого задания внутри кода. Это весьма важная деталь, поскольку она означает, что впредь мы можем повторно использовать контроллеры с различными элементами, сводя к минимуму повторяемость кода.

Доступ к представлениям

Согласно общей схеме, на каждое представление должен приходиться свой контроллер. Поскольку у представления имеется идентификатор (ID), он может быть легко и просто передан контроллерам. А элементы внутри представления вместо ID используют атрибуты `class`, поэтому они не конфликтуют с элементами в других представлениях. Эта схема предоставляет неплохую структуру для общей практики, но от нее невозможно добиться чёткого однообразия.

До сих пор в данной главе мы получали доступ к представлениям с помощью использования селектора `jQuery()`, сохраняя локальную ссылку на представление внутри контроллера. Затем последующие поиски элементов внутри представления получали область видимости за счет этой ссылки на представление, ускоряя тем самым поиск этих элементов:

```
// ...
init: function(view){
  this.view = $(view);
  this.form = this.view.find("form");
}
```

Но это означает, что контроллер наполняется множеством селекторов, требуя от нас постоянно запрашивать DOM. Мы можем сделать код немного чище, если

у нас будет одно место в контроллере, где селекторы отображаются на имена переменных:

```
elements: {
  "form.searchForm": "searchForm",
  "form input[type=text]": "searchInput"
}
```

Тем самым гарантируется, что переменные `this.searchForm` и `this.searchInput` будут созданы в отношении контроллера и настроены на соответствующие им элементы при создании экземпляра этого контроллера. Это обычные объекты jQuery, поэтому мы можем манипулировать ими как обычно, устанавливая обработчики событий и извлекая атрибуты.

Давайте реализуем поддержку для этого отображения элементов внутри наших контроллеров, последовательно перебирая все селекторы и устанавливая локальные переменные. Мы будем делать это внутри нашей функции `init()`, которая вызывается при создании экземпляра контроллера:

```
var exports = this;
jQuery(function($){
  exports.SearchView = Controller.create({
    // Отображение селекторов на имена локальных переменных
    elements: {
      "input[type=search]": "searchInput",
      "form": "searchForm"
    },
    // Вызываются в процессе создания экземпляра
    init: function(element){
      this.el = $(element);
      this.refreshElements();
      this.searchForm.submit(this.proxy(this.search));
    },
    search: function(){
      console.log("Searching:", this.searchInput.val());
    },
    // Закрытая область
    $: function(selector){
      // Востребование свойства `el` и определение области применения запроса
      return $(selector, this.el);
    },
    // Установка локальных переменных
    refreshElements: function(){
      for (var key in this.elements) {
        this[this.elements[key]] = this.$(key);
      }
    }
  });
  new SearchView("#users");
});
```

Функция `refreshElements()` ожидает от каждого контроллера наличия текущего свойства элемента `el`, определяющего область применения любого селектора. При вызове функции `refreshElements()` в отношении контроллера будут установлены свойства `this.searchForm` и `this.searchInput`, которые впоследствии будут доступны для привязки событий и работы с DOM.

Полную версию примера можно увидеть в файлах, сопровождающих эту книгу, а именно в файле assets/ch04/views.html.

Делегирование событий

Мы также должны заняться очисткой всех этих привязок событий и представительств (proxying) за счет использования объекта `events`, отображающего типы событий и селекторы на функции обратного вызова. Это очень похоже на объект `elements`, но в отличие от него, будет использована следующая форма:

```
events: {
  "submit form": "submit"
}
```

Давайте продолжим работу и добавим это к нашему контроллеру `SearchView`. Как и в функции `refreshElements()`, у нас будет функция `delegateEvents()`, которая будет вызываться при создании экземпляра контроллера. Она будет проводить разбор объекта события контроллера, подключая функцию обратного вызова для этого события. В нашем примере `Search View` нам нужна функция `search()`, вызываемая при отправке элемента `<form />` представления:

```
var exports = this;
jQuery(function($){
  exports.SearchView = Controller.create({
    // Отображение всех имен событий,
    // селекторов и функций обратного вызова
    events: {
      "submit form": "search"
    },

  init: function(){
  // ...
    this.delegateEvents();
  },

  search: function(e){ /* ... */ },

  // Закрытая область

  // Разбиение по первому пробельному символу
  eventSplitter: /^(\w+)\s*(.*)$/,

  delegateEvents: function(){
```

продолжение ⇨

```
    for (var key in this.events) {
      var methodName = this.events[key];
      var method = this.proxy(this[methodName]);

      var match = key.match(this.eventSplitter);
      var eventName = match[1], selector = match[2];

      if (selector === '') {
        this.el.bind(eventName, method);
      } else {
        this.el.delegate(selector, eventName, method);
      }
    }
  }
});
```

Обратите внимание, что мы используем функцию `delegate()`, а также функцию `bind()` внутри функции `delegateEvents()`. Если селектор события не предоставлен, событие будет помещено прямо на `el`. В противном случае событие будет делегировано и будет инициировано, если тип события, запущенного в отношении дочернего объекта, соответствует селектору. Преимущество делегирования состоит в том, что оно зачастую сокращает объем требуемых слушателей событий — т. е. слушателей не нужно помещать на каждый выбранный элемент, потому что события перехватываются в динамике по мере их всплывания.

Мы можем вставить все эти контроллерные усовершенствования восходящего потока в нашу библиотеку `Controller`, чтобы они могли повторно использоваться в каждом контроллере. В окончательном варианте пример имеет следующий вид (всю библиотеку контроллера вы можете найти в файле assets/ch04/finished_controller.html):

```
var exports = this;

jQuery(function($){
  exports.SearchView = Controller.create({
    elements: {
      "input[type=search]": "searchInput",
      "form": "searchForm"
    },
    events: {
      "submit form": "search"
    },
    init: function(){ /* ... */ },
    search: function(){
      alert("Searching: " + this.searchInput.val());
      return false;
    },
  });
  new SearchView({el: "#users"});
});
```

Конечные автоматы

Конечные автоматы (State machines, или, в более правильном варианте, Finite State Machines (FSM)) являются отличным способом программирования пользовательских интерфейсов (UI). Используя конечные автоматы, мы можете запросто управлять несколькими контроллерами, по мере необходимости показывать и скрывать представления. А что из себя представляет сам конечный автомат? По своей сути, конечный автомат состоит из двух вещей: состояний и переходов. У него только одно активное состояние, но множество пассивных. При смене активного состояния вызываются переходы между состояниями.

Как это работает на практике? Предположим, у приложения есть несколько представлений, которые нужно показывать независимо друг от друга, скажем, представление для показа контактов и представление для редактирования контактов. Эти два представления нуждаются в эксклюзивном выводе на экран — когда показывается одно из них, второе представление должно быть скрыто. Это отличный сценарий для представления конечного автомата, потому что он будет гарантировать в любой заданный момент времени активность только одного представления. Конечно, если нам нужно добавить дополнительные представления, например представление для настроек, использование конечного автомата делает эту задачу тривиальной.

Давайте разберем практический пример, который даст вам четкое понимание того, как может быть реализован конечный автомат. Это простой пример, не учитывающий разные типы переходов, но для нас он вполне достаточен. Во-первых, мы собираемся создать объект `Events`, который будет использовать API событий, имеющийся в jQuery (см. главу 2), чтобы привязывать и события к нашему конечному автомату и инициировать эти события:

```
var Events = {
bind: function(){
  if ( !this.o ) this.o = $({});
  this.o.bind.apply(this.o, arguments);
},

trigger: function(){
  if ( !this.o ) this.o = $({});
    this.o.trigger.apply(this.o, arguments);
  }
};
```

Объект `Events`, по сути, расширяет существующую в jQuery поддержку событий за пределами DOM, поэтому мы хотим использовать его в нашей собственной библиотеке. А теперь приступим к созданию класса `StateMachine`, у которого будет одна основная функция `add()`:

```
var StateMachine = function(){};
StateMachine.fn = StateMachine.prototype;
```

продолжение ➩

```
// Добавление привязки и инициирования события
$.extend(StateMachine.fn, Events);

StateMachine.fn.add = function(controller){
  this.bind("change", function(e, current){
    if (controller == current)
      controller.activate();
    else
      controller.deactivate();
  });

  controller.active = $.proxy(function(){
    this.trigger("change", controller);
  }, this);
};
```

Имеющаяся в конечном автомате функция `add()` добавляет переданный ей контроллер к списку состояний и создает функцию `active()`. Когда вызывается функция `active()`, активное состояние передается контроллеру. Конечный автомат будет вызывать функцию `activate()` в отношении активного контроллера, а функцию `deactivate()` в отношении всех остальных контроллеров. Мы можем увидеть, как это работает, путем создания двух экземпляров контроллеров, добавив их к конечному автомату, а затем активировав один из них:

```
var con1 = {
activate: function(){ /* ... */ },
deactivate: function(){ /* ... */ }
};

var con2 = {
activate: function(){ /* ... */ },
deactivate: function(){ /* ... */ }
};

// Создание нового экземпляра StateMachine и добавление состояний
var sm = new StateMachine;
sm.add(con1);
sm.add(con2);

// Активация первого состояния
con1.active();
```

Функция `add()`, имеющаяся в конечном автомате, работает путем создания функции обратного вызова для события `change`, вызывающей функцию `activate()` или `deactivate()`, в зависимости от того, какая из них больше подходит. Хотя конечный автомат дает нам функцию `active()`, мы также можем изменить состояние, самостоятельно запуская событие `change`:

```
sm.trigger("change", con2);
```

Внутри функции activate() нашего контроллера мы можем настроить и вывести его представление, добавить и показать элементы. Аналогично, внутри функции deactivate() мы можем отменить что-нибудь, скрыв представление. Классы CSS предлагают отличный способ скрытия и показа представлений. Просто добавьте класс, скажем, .active, когда представление активно, и удалите его в процессе деактивации:

```
var con1 = {
  activate: function(){
    $("#con1").addClass("active");
    },
    deactivate: function(){
    $("#con1").removeClass("active");
  }
};

var con2 = {
  activate: function(){
    $("#con2").addClass("active");
  },
  deactivate: function(){
    $("#con2").removeClass("active");
  }
};
```

Затем в своих таблицах стилей обеспечьте наличие класса .active у представлений, при отсутствии которого они будут скрыты:

```
#con1, #con2 { display: none; }
#con1.active, #con2.active { display: block; }
```

Полные версии примеров можно увидеть в файле assets/ch04/state_machine.html.

Маршрутизация

Теперь наше приложение запускается с отдельной страницы, следовательно ее URL не будет изменяться. Это станет проблемой для наших пользователей, поскольку они уже привыкли к наличию уникального URL для ресурса в Интернете. Кроме того, люди привыкли к путешествию по Интернету с помощью кнопок браузера, прокручивающих историю посещений вперед и назад.

Чтобы решить эту проблему, нам нужно привязать состояние приложения к URL. При смене состояния приложения будет также меняться и URL. Справедливо и обратное утверждение — при смене URL будет соответственно меняться и состояние приложения. В процессе загрузки исходной страницы мы будем проверять URL и устанавливать исходное состояние приложения.

Использование хэшей URL-адресов

Но базовый URL страницы не может быть изменен без запуска обновления страницы, а этого-то как раз мы и стараемся избежать. К счастью, решения на сей счет

имеются. Традиционный способ работы с URL был связан с изменением его хэша. Хэш никогда не отправляется на сервер, поэтому он может быть изменен без запуска обновления страницы. Например, так выглядит URL страницы моего Твиттера с содержимым хэша `#!/maccman`:

```
http://twitter.com/#!/maccman
```

Вы можете извлечь и изменить хэш страницы, используя объект `location`:

```
// Установка хэша
window.location.hash = "foo";
assertEqual( window.location.hash , "#foo" );
// Лишение знака "#"
var hashValue = window.location.hash.slice(1);
assertEqual( hashValue, "foo" );
```

Если URL не имеет хэша, свойство `location.hash` имеет значение пустой строки. В противном случае значение `location.hash` равно хэш-фрагменту URL с префиксом `#`.

Слишком частая установка значения хэша может отрицательно сказаться на производительности, особенно на мобильных браузерах. Поэтому при их частой установке, скажем, при прокрутке пользователем списка, может быть, стоит подумать о регулировке этого процесса.

Обнаружение изменений хэша

Исторически сложилось так, что изменения хэша довольно грубо обнаруживались с помощью последовательного таймерного опроса. Но все меняется к лучшему, и современные браузеры поддерживают событие `hashchange`. Это событие инициируется в отношении окна, и вы можете отследить его, чтобы получить изменения хэша:

```
window.addEventListener("hashchange", function(){ /* ... */ }, false);
```

или же с помощью jQuery:

```
$(window).bind("hashchange", function(event){
  // хэш изменился, меняем состояние
});
```

При запуске события `hashchange` мы должны обеспечить соответствующее состояние приложения. События имеют хорошую кроссбраузерную поддержку с реализацией во всех последних версиях основных браузеров:

- IE >= 8
- Firefox >= 3.6
- Chrome
- Safari >= 5
- Opera >= 10.6

Событие не инициируется на старых браузерах, но *существует полезный дополнительный модуль jQuery*, который добавляет событие hashchange к устаревшим браузерам.

Следует заметить, что это событие не инициируется при начальной загрузке страницы и инициируется только при изменениях хэша. Если вы в своем приложении используете маршрутизацию с помощью хэша, то при загрузке страницы вам может потребоваться самостоятельный запуск события:

```
jQuery(function(){
  var hashValue = location.hash.slice(1);
  if (hashValue)
  $(window).trigger("hashchange");
});
```

Ajax Crawling

Так как агенты поисковых машин (search engine crawlers) JavaScript не выполняют они не могут видеть содержимое, создаваемое в динамическом режиме. Кроме этого ни один из наших хэш-маршрутов не сможет быть проиндексирован: с точки зрения агентов поисковых машин все URL одинаковы, поскольку хэш-фрагмент никогда не отправляется на сервер.

Если нужно, чтобы наше безупречное JavaScript-приложение могло индексироваться и быть доступным таким поисковым машинам, как Google, значит мы точно столкнулись с проблемой. В качестве средства обхода разработчики создают «параллельный мир» содержимого. Агенты будут отсылаться на специальные статические HTML-кадры содержимого, а нормальные браузеры будут продолжать использовать динамическую JavaScript-версию приложения. Это значительно увеличивает нагрузку на разработчиков и вынуждает заниматься мониторингом браузера, чего лучше было бы избежать. К счастью, Google предоставляет альтернативу: *спецификацию Ajax Crawling*.

Давайте еще раз посмотрим на адрес моего профиля в Твиттере (обратите внимание на восклицательный знак после знака решетки):

```
http://twitter.com/#!/maccman
```

Восклицательный знак сигнализирует поисковым агентам Google, что наш сайт соответствует спецификации Ajax Crawling. Вместо того чтобы запрашивать URL в чистом виде, исключая, разумеется, хэш, агент переводит URL в следующий вид:

```
http://twitter.com/?_escaped_fragment_=/maccman
```

Хэш был заменен URL-параметром `_escaped_fragment_`. В спецификации это называется *испорченным* URL (ugly URL), который пользователи никогда не увидят. Затем агент продолжает свою работу и извлекает этот испорченный URL. Поскольку хэш-фрагмент теперь является URL-параметром, ваш сервер знает, какой именно ресурс запрашивается поисковым агентом — в данном случае это моя страница в Твиттере.

Затем сервер может отобразить этот испорченный URL на тот ресурс, который он представляет, и отправить в ответ чистый HTML или текстовый фрагмент, который затем индексируется. Поскольку Твиттер все еще имеет статическую версию своего сайта, поисковый агент просто перенаправляется на эту версию.

```
curl -v http://twitter.com/?_escaped_fragment_=/maccman
302 redirected to http://twitter.com/maccman
```

Так как в Твиттере используется временное перенаправление (302), а не постоянное (301), URL, показываемый в результатах поиска, обычно будет представлен хэш-адресом, т. е. динамической JavaScript-версией сайта (http://twitter.com/#!/maccman). Если у вас нет статической версии вашего сайта, просто предоставьте статический HTML или текстовый фрагмент, когда URL-адреса запрашиваются с параметром `_escaped_fragment_`.

Как только вы добавите к своему сайту поддержку спецификации Ajax Crawling, вы сможете проверить его в работе, воспользовавшись средством Fetch as Googlebot. Если вы не станете заниматься реализацией этой схемы на своем сайте, страницы будут по-прежнему индексироваться «как есть», с высокой степенью вероятности не получить правильного представления в результатах поиска. Но когда-нибудь такие поисковые машины, как Google, добавят поддержку JavaScript к своим поисковым агентам, сделав излишними подобные схемы.

Использование History API HTML5

History API является частью спецификации HTML5 и, по сути, позволяет вам заменять текущее местоположение произвольным URL. Можно также выбрать, нужно ли добавлять новый URL к истории браузера, предоставляя вашему браузеру поддержку «кнопки возврата». Как и при установке хэша местоположения, главное здесь в том, что страница не будет перезагружена, ее состояние будет сохранено заранее.

Эту спецификацию поддерживают следующие браузеры:

- Firefox >= 4.0
- Safari >= 5.0
- Chrome >= 7.0
- IE: не поддерживает
- Opera >= 11.5

Этот API предельно прост и вращается главным образом вокруг функции `history.pushState()`. Ей передаются три аргумента: объект данных, заголовок и новый URL-адрес:

```
// объект данных является необязательным аргументом,
// и он передается вместе с событием popstate
var dataObject = {
  createdAt: '2011-10-10',
```

```
  author: 'donnamoss'
};
var url = '/posts/new-url';
history.pushState(dataObject, document.title, url);
```

Все три аргумента являются необязательными, но они управляют тем, что помещается в стек истории браузера.

Объект `data`

Указывать его совсем не обязательно — указывается любой пользовательский объект на ваше усмотрение. Он будет передан вместе с событием `popstate` (которое чуть позже будет рассмотрено более подробно).

Аргумент `title`

Пока большинством браузеров он игнорируется, но, согласно спецификации, он будет изменять заголовок новой страницы и появляться в истории браузера.

Аргумент `url`

Это строка, определяющая URL для замены текущего местоположения браузера. Если аргумент имеет относительный характер, новый URL вычисляется относительно текущего, с тем же доменом, портом и протоколом. Можно также указать абсолютный URL, но из соображений безопасности он будет ограничен тем же самым доменом, что и URL текущего местоположения.

Проблема в использовании нового History API в JavaScript-приложениях заключается в том, что каждому URL необходимо реальное HTML-отображение. Хотя при вызове функции `history.pushState()` браузеры не будут запрашивать новый URL, он будет запрошен при перезагрузке страницы. Иными словами, каждый URL, передаваемый этому API, должен существовать на самом деле, вы не можете просто выдумать фрагменты, как это было при работе с хэшами.

Если у вашего сайта уже есть статическое HTML-отображение, проблем не будет, но они возникнут, если он построен исключительно на применении JavaScript. Одно из решений заключается в том, чтобы всегда предлагать JavaScript-приложение, независимо от вызванного URL. К сожалению, это нарушит поддержку ошибки 404 (страница не найдена), поскольку каждый URL будет возвращать успешный ответ. Альтернативой может послужить реальное проведение проверок на стороне сервера, чтобы убедиться в допустимости URL и запрошенного ресурса, прежде чем предложить приложение.

В History API содержится ряд других свойств. Функция `history.replaceState()` работает точно так же, как функция `history.pushState()`, но она не добавляет запись в стек истории. Вы можете перемещаться по истории браузера, используя функции `history.back()` и `history.forward()`.

Ранее упомянутое событие `popstate` инициируется при загрузке страницы или при вызове функции `history.pushState()`. В последнем случае в объекте `event`

будет содержаться свойство `state`, содержащее объект `data`, переданный функции `history.pushState()`:

```
window.addEventListener("popstate", function(event){
  if (event.state) {
    // была вызвана функция history.pushState()
  }
});
```

Вы можете отследить событие и гарантировать совместимость состояния приложения с URL. При использовании jQuery вам нужно иметь в виду, что событие приведено к стандарту. Поэтому для доступа к объекту состояния нужно обратиться к исходному событию:

```
$(window).bind("popstate", function(event){
  event = event.originalEvent;
  if (event.state) {
    // была вызвана функция history.pushState()
  }
});
```

5 Представления и использование шаблонов

Представления являются интерфейсом вашего приложения, именно их видит и с ними взаимодействует конечный пользователь. В нашем случае представления — это не содержащие логики HTML-фрагменты, управляемые контроллерами приложения, которые имеют дело с обработчиками событий и вставкой данных. Именно здесь может появиться сильный соблазн сломать абстракцию MVC, включив логику непосредственно в ваши представления. Не поддавайтесь этому соблазну! Все закончится созданием бессмысленной, запутанной программы.

Одно из самых больших архитектурных изменений, которое вы должны сделать, перемещаясь с серверной стороны приложения на его клиентскую сторону, касается представлений. Традиционно вы можете только лишь вставить данные серверной стороны вместе с HTML-фрагментами, создавая новые страницы. Но представления в JavaScript-приложениях несколько отличаются от традиционных.

Во-первых, вы должны передать любые данные, необходимые представлению, в адрес клиента, поскольку у вас нет доступа к переменным на серверной стороне. Обычно это делается с помощью Ajax-запроса, возвращающего JSON-объект, который затем загружается моделями вашего приложения. Вы не должны заранее интерпретировать любой HTML на серверной стороне, все это должно быть делегировано клиенту. Тем самым гарантируется, что ваше приложение на стороне клиента не полагается на сервер для интерпретации представлений, сохраняя мгновенную реакцию своего интерфейса.

Затем вы загружаете данные в ваши представления либо путем динамического создания DOM-элементов с помощью JavaScript, либо путем использования шаблонов. Ниже я остановлюсь на этих двух возможностях более подробно.

Динамически интерпретируемые представления

Один из способов создания представлений основан на практическом применении JavaScript. Вы можете создать DOM-элементы, воспользовавшись функцией `document.createElement()`, установив их содержимое и добавив их к странице.

Когда наступит время перерисовки представления, его нужно будет опустошить и повторить процесс:

```
var views = document.getElementById("views");
views.innerHTML = ""; // Опустошение элемента

var container = document.createElement("div");
container.id = "user";
var name = document.createElement("span");
name.innerHTML = data.name;
container.appendChild(name);
views.appendChild(container);
```

Или для более лаконичного API при использовании jQuery:

```
$("#views").empty();
var container = $("<div />").attr({id: "user"});
var name      = $("<span />").text(data.name);
$("#views").append(container.append(name));
```

Я мог бы с этим согласиться только в том случае, если представление, которое нужно интерпретировать, очень маленькое, состоящее, возможно, всего из пары элементов. Помещение элементов представления в ваши контроллеры или состояния компрометирует MVC-архитектуру приложения.

Вместо создания элементов с нуля я советую включать статический HTML в страницу, скрывая или показывая его по мере необходимости. Тогда весь код в ваших контроллерах, относящийся к представлению, будет сведен к абсолютному минимуму, и вы сможете просто обновлять содержимое элемента по мере необходимости.

Давайте, к примеру, создадим HTML-фрагмент, который будет служить в качестве нашего представления:

```
<div id="views">
  <div class="groups"> ... </div>
  <div class="user">
    <span></span>
  </div>
</div>
```

Теперь для обновления представления и переключения отображения различных элементов мы можем воспользоваться селекторами jQuery:

```
$("#views div").hide();
var container = $("#views .user");
container.find("span").text(data.name);
container.show();
```

Для генерации элементов этот метод более предпочтителен, поскольку он в максимальной степени обособляет представление и контроллер.

Шаблоны

Если вы привыкли вставлять серверные переменные в HTML, то использование шаблонов должно быть вам знакомо. Существует множество библиотек шаблонов, и ваш выбор будет, наверное, зависеть от используемой вами DOM-библиотеки. И тем не менее большинство библиотек используют похожий синтаксис, который будет рассмотрен ниже.

Суть JavaScript-шаблонов заключается в том, что вы можете взять HTML-фрагмент, в который вставлены переменные шаблона, и скомпоновать его с объектом JavaScript, заменяя эти переменные шаблона значениями из объекта. В целом работа с шаблонами с помощью JavaScript во многом похожа на библиотеки для работы с шаблонами, имеющиеся в других языках, такие как Smarty в PHP, ERB в Ruby, и строковое форматирование в Python.

В качестве основы для примеров работы с шаблонами мы собираемся воспользоваться библиотекой jQuery.tmpl. Если вы не используете jQuery или если вы хотите использовать другую библиотеку для работы с шаблонами, примеры вам все равно пригодятся, поскольку синтаксис работы с шаблонами для большинства библиотек если не идентичен, то очень похож. Если вы ищете хорошую альтернативу, присмотритесь к библиотеке Mustache, у которой есть реализации во многих языках, включая JavaScript.

Созданная Microsoft библиотека jQuery.tmpl является дополнительным модулем для работы с шаблонами, основанным на оригинальной авторской работе Джона Ресига (John Resig). Эта библиотека имеет хорошую поддержку и полную документацию на сайте jQuery. В библиотеке есть одна основная функция `jQuery.tmpl()`, которой можно передать шаблон и данные. Она интерпретирует элемент шаблона, который вы можете добавить к документу. Если данные являются массивом, шаблон интерпретируется по одному разу для каждого элемента данных, имеющегося в массиве; в противном случае интерпретируется только один шаблон:

```
var object = {
  url: "http://example.com",
  getName: function(){ return "Trevor"; }
};
var template = '<li><a href="${url}">${getName()}</a></li>';
var element = jQuery.tmpl(template, object);
// В результате получается:
<li><a href="http://example.com">Trevor</a></li>
$("body").append(element);
```

Как видите, мы вставляем переменные с помощью синтаксиса `${}`. Все, что находится внутри скобок, вычисляется в контексте объекта, переданного функции `jQuery.tmpl()`, независимо от того, является ли это свойством или функцией.

Но шаблоны — намного более мощный инструмент, способный не только на вставку. Большинство библиотек для работы с шаблонами имеет такие расширенные

возможности, как условное изменение хода интерпретации и применение итераций. Вы можете управлять ходом интерпретации, используя инструкции `if` и `else`, аналогичные тем, которые используются в самом языке JavaScript. Разница в том, что здесь нам нужно заключать ключевые слова в двойные скобки, чтобы они были соответствующим образом восприняты механизмом работы с шаблонами:

```
{{if url}}
  ${url}
{{/if}}
```

Блок `if` будет выполнен, если указанное значение атрибута не вычисляется в `false`, `0`, `null`, `""`, `Nan` или `undefined`. Как видите, блок закрывается с помощью ключевого слова `{{/if}}`, поэтому не забудьте его включить! Довольно распространенной схемой является вывод сообщения, когда массив, скажем, сообщений в чате, пуст:

```
{{if messages.length}}
  <!-- Вывод сообщения... -->
{{else}}
  <p>Извините, сообщений нет</p>
{{/if}}
```

Никакая библиотека для работы с шаблонами не может позволить себе отсутствие итерации. Используя библиотеки JS, вы можете осуществлять последовательный перебор любого JavaScript-типа — `Object` или `Array`, — используя ключевое слово `{{each}}`. Если `Object` передать `{{each}}`, то блок кода будет применен к последовательно перебираемым свойствам объекта. Точно так же передача массива приводит к применению блока к каждому элементу массива.

Внутри блока можно получить доступ к значению, полученному в результате текущей итерации, воспользовавшись для этого переменной `$value`. Отображение значения осуществляется так же, как и в предыдущем примере со вставкой, для чего используется синтаксис `${$value}`. Рассмотрим следующий объект:

```
var object = {
  foo: "bar",
  messages: ["Hi there", "Foo bar"]
};
```

Затем воспользуемся следующим шаблоном для последовательного перебора массива сообщений, отображая каждое сообщение. Дополнительно с помощью переменной `$index` будет показан текущий индекс итерации.

```
<ul>
  {{each messages}}
    <li>${$index + 1}: <em>${$value}</em></li>
  {{/each}}
</ul>
```

Как видите, имеющийся в jQuery.tmpl API для работы с шаблонами весьма прост в использовании. Как уже ранее упоминалось, большинство альтернативных библи-

отек для работы с шаблонами имеют похожий API, хотя многие из них предлагают расширенные возможности, например lambda-функции, парциалы и комментарии.

Шаблонные помощники

Иногда полезно будет внутри шаблонов воспользоваться общими функциями-помощниками, например для форматирования дат или чисел. Но при этом важно помнить об использовании MVC-архитектуры и не вставлять в представление функции без достаточных на то оснований. Давайте, к примеру, заменим ссылки в некотором простом тексте тегами <a>. Несомненно, следующий код демонстрирует совершенно неправильный подход к решению этой задачи:

```
<div>
  ${ this.data.replace(
  /((http|https|ftp):\/\/[\w?=&.\/-;#~%-]+(?![\w\s?&.\/;#~%"=-]*>))/g,
  '<a target="_blank" href="$1">$1</a> ') }
  </div>
```

Вместо того чтобы внедрять функцию непосредственно в представление, мы должны ее абстрагировать и предоставить ей пространство имен, сохраняя обособленность логики от представлений. В данном случае мы собираемся создать отдельный файл helpers.js, содержащий все функции-помощники, такие как функция `autoLink()`. Затем мы можем с помощью нашей функции-помощника навести порядок в представлении:

```
// helper.js
var helper = {};
helper.autoLink = function(data){
  var re = /((http|https|ftp):\/\/[\w?=&.\/-;#~%-]+(?![\w\s?&.\/;#~%"=-]*>))/g;
  return(data.replace(re, '<a target="_blank" href="$1">$1</a> ') );
};
// template.html
<div>
  ${ helper.autoLink(this.data) }
</div>
```

Здесь есть еще и дополнительное преимущество: теперь функция `autoLink()` является общедоступной и может быть повторно использована в любом месте приложения.

Хранение шаблонов

У решения вопроса о хранении шаблонов представления есть несколько вариантов.

- Встроить их в JavaScript
- Встроить их в пользовательский `script`-тег
- Загрузить с удаленного сервера
- Встроить их в HTML

Некоторые из них лучше вписываются в MVC-архитектуру. Лично я за хранение шаблонов путем встраивания их в пользовательские `script`-теги, и аргументы в пользу такого выбора будут рассмотрены ниже.

Вы можете хранить шаблоны внутри ваших файлов JavaScript. Но я не рекомендую так поступать, потому что это приводит к размещению кода представления внутри контроллера, нарушая тем самым MVC-архитектуру.

Отправляя Ajax-вызовы, вы можете динамически подгружать шаблоны по мере надобности. Преимуществом такого подхода является малый размер загружаемой исходной страницы, а недостаток заключается в том, что во время загрузки шаблона вы можете замедлить работу пользовательского интерфейса. Одна из основных причин создания JavaScript-приложений заключается в повышении скорости их работы, поэтому при загрузке удаленных ресурсов нужно подумать об утрате этого преимущества.

Можно хранить шаблоны встроенными в HTML-код страницы. Преимуществом такого подхода является отсутствие проблемы с медленной загрузкой, имеющейся у удаленного извлечения шаблонов. Исходный код намного более очевиден — шаблоны встроены туда, где они будут отображаться и использоваться. Явный недостаток заключается в том, что этот подход влечет за собой большой размер страницы. Но справедливости ради стоит заметить, что разница в скорости может быть незначительной, особенно если используется сжатие и кэширование страниц.

Я же рекомендую воспользоваться пользовательскими `script`-тегами, ссылаясь на них из JavaScript по идентификаторам ID. Это весьма удобный способ хранения шаблонов, особенно если вы хотите использовать их в нескольких местах. Пользовательские `script`-теги имеют также то преимущество, что они не интерпретируются браузером, который принимает их содержимое в качестве текста.

Если шаблон определен встроенным в страницу, вы можете воспользоваться функцией `jQuery.fn.tmpl(data)` — т. е. вызвать `tmpl()` в отношении элемента jQuery:

```
<script type="text/x-jquery-tmpl" id="someTemplate">
  <span>${getName()}</span>
</script>
<script>
  var data = {
    getName: function(){ return "Bob" }
  };
  var element = $("#someTemplate").tmpl(data);
  element.appendTo($("body"));
</script>
```

Кроме всего прочего, jQuery.tmpl обеспечивает кэширование скомпилированного шаблона после его генерирования. Это ускоряет работу приложения, поскольку шаблон при следующем его использовании не нужно повторно компилировать. Обратите внимание, что мы генерируем элемент перед добавлением его к странице. Этот метод имеет более высокую производительность по сравнению с манипу-

ляцией элементами, уже встроенными в страницу, и мы рекомендуем пользоваться им в практической работе.

Даже если вы интерпретируете все шаблоны встроенными в страницу, это не означает, что ваша серверная сторона должна быть структурирована подобным образом. Старайтесь хранить каждый шаблон в отдельном файле (или парциале), а затем, при запросе страницы, объединять их в один документ. Некоторые средства управления зависимостями (такие как RequireJS, которые будут рассмотрены в главе 6) сделают все это для вас.

Связывание

Связывание позволяет увидеть реальные преимущества от интерпретации представлений на стороне клиента. По своей сути, связывание привязывает друг к другу элемент представления и объект JavaScript (обычно модель). При изменении объекта JavaScript автоматически обновляется и представление, чтобы отобразить только что модифицированный объект. Иными словами, как только у вас представления становятся связанными с моделями, при обновлении моделей приложения представления будут автоматически интерпретироваться заново.

Связывание — действительно хорошая вещь. Оно означает, что ваши контроллеры не должны заниматься обновлением представлений при изменении записей, поскольку все это происходит автоматически в фоновом режиме. Структурирование вашего приложения с помощью инструментов связывания также прокладывает путь к приложениями реального времени, которые будут более подробно рассматриваться в главе 8.

Для связывания объектов JavaScript и представлений нам нужно получить функцию обратного вызова, которая заставляет представление обновиться при изменении свойства объекта. Затруднения заключаются в том, что JavaScript не предоставляет для этого свой собственный метод. В языке отсутствует функциональное средство обработки отсутствия метода, такого как `method_missing` в Ruby или Python, и пока в JavaScript невозможно имитировать его поведение с помощью получателей *геттеров* (getter) и предоставителей *сеттеров* (setter). Но поскольку JavaScript язык очень динамический, мы можем создать свою собственную функцию обратного вызова `change`:

```
var addChange = function(ob){
  ob.change = function(callback){
    if (callback) {
      if ( !this._change ) this._change = [];
      this._change.push(callback);
    } else {
      if ( !this._change ) return;
      for (var i=0; i < this._change.length; i++)
        this._change[i].apply(this);
    }
  };
};
```

Функция addChange() добавляет функцию change() к любому переданному ей объекту. Функция change() работает точно так же, как и событие change в jQuery. Вы можете добавить функции обратного вызова, запуская change() с функцией, или инициировать событие, вызвав change() без аргументов. Давайте посмотрим на нее в практическом применении:

```
var object = {};
object.name = "Foo";
addChange(object);
object.change(function(){
  console.log("Изменено!", this);
  // Потенциально здесь происходит обновление представления
});
object.change();
object.name = "Bar";
object.change();
```

Итак, мы добавили функцию обратного вызова change() к объекту, что позволило нам привязать и инициировать события change.

Привязка моделей

Теперь давайте разовьем дальше пример связывания и применим его к моделям. При каждом создании, обновлении или удалении записи модели мы будем инициировать событие change, заново интерпретируя представление. В приводимом ниже примере мы создаем базовый класс User, устанавливая связывание и инициирование события, и в заключение прислушиваемся к событию change, заново интерпретируя представление, как только оно будет инициировано:

```
<script>
  var User = function(name){
    this.name = name;
  };
  User.records = []
  User.bind = function(ev, callback) {
    var calls = this._callbacks || (this._callbacks = {});
    (this._callbacks[ev] || (this._callbacks[ev] = [])).push(callback);
  };
  User.trigger = function(ev) {
    var list, calls, i, l;
    if (!(calls = this._callbacks)) return this;
    if (!(list  = this._callbacks[ev])) return this;
    jQuery.each(list, function(){ this() })
  };
  User.create = function(name){
    this.records.push(new this(name));
    this.trigger("change")
  };
  jQuery(function($){
```

```
    User.bind("change", function(){
      var template = $("#userTmpl").tmpl(User.records);
      $("#users").empty();
      $("#users").append(template);
    });
  }):
</script>
<script id="userTmpl" type="text/x-jquery-tmpl">
  <li>${name}</li>
</script>
<ul id="users">
</ul>
```

Теперь, как только будет изменена запись объекта User, будет инициировано событие change модели User, запуская нашу функцию обратного вызова, работающую с шаблоном и перерисовывая список пользователей. Это весьма полезная технология, и мы можем работать над созданием и обновлением записей пользователей, нисколько не заботясь об обновлении представления, которое будет происходить автоматически. Давайте, к примеру, создадим новый объект User:

```
User.create("Sam Seaborn");
```

Будет вызвано событие change объекта User, и наш шаблон будет интерпретирован заново, автоматически обновляя представление и показывая нашего нового пользователя. Полную версию примера привязки модели можно увидеть в assets/ch05/model.html.

6 Управление зависимостями

Одним из обстоятельств, удерживающих язык JavaScript в числе отстающих, было отсутствие управления зависимостями и системы модулей. В отличие от других языков, при изучении JavaScript пространству имен и модулям не придавалось особого значения. В действительности такие популярные библиотеки, как jQuery, не навязывали какой-либо прикладной структуры, все это возлагалось на самого разработчика. Слишком часто мне доводилось встречаться с плохо структурированным JavaScript, с сумасшедшим количеством отступов и безымянных функций. Вам знакома такая картина?

```
function() {
  function() {
    function() {
      function() {
      }
    }
  }
}
```

Использование модулей и пространства имен — в порядке вещей, но отсутствие своих собственных систем зависимостей при создании больших приложений беспокоило все больше и больше. Долгое время считалось, что вполне достаточно `script`-тега, поскольку объем кода JavaScript, присутствующего на странице, больше ничем не регулировался. Но при создании сложных JavaScript-приложений без системы зависимостей никак не обойтись. Совершенно непрактично отслеживать зависимости самостоятельно, вручную добавляя `script`-теги на страницу. Зачастую в результате получалась сплошная неразбериха, похожая на эту:

```
<script src="jquery.js" type="text/javascript" charset="utf-8"></script>
<script src="jquery.ui.js" type="text/javascript" charset="utf-8"></script>
<script src="application.utils.js" type="text/javascript"
        charset="utf-8"></script>
<script src="application.js" type="text/javascript" charset="utf-8"></script>
<script src="models/asset.js" type="text/javascript" charset="utf-8"></script>
<script src="models/activity.js" type="text/javascript" charset="utf-8">
        </script>
```

```
<script src="states/loading.js" type="text/javascript" charset="utf-8">
        </script>
<script src="states/search.js" type="text/javascript" charset="utf-8">
        </script>
<!-- ... -->
```

Здесь речь идет не только о практичности, которая гарантируется конкретной системой управления зависимостями, но и об аспектах производительности. Ваш браузер должен сделать HTTP-запрос для каждого из этих JavaScript-файлов, и, хотя он может быть сделан в асинхронном режиме, такое количество соединений обходится недешево. У каждого соединения имеются издержки на HTTP-заголовки, вроде cookie-файлов, и нужно инициировать еще одно квитирование TCP. Ситуация усугубляется, если ваше приложение обслуживается с использованием SSL.

CommonJS

По мере того как JavaScript перемещался на серверную сторону, было высказано несколько предложений по управлению зависимостями. Движки SpiderMonkey и Rhino предлагают функцию `load()`, но у них нет никакой конкретной схемы для организации пространства имен. В среде Node.js имеется функция `require()`, предназначенная для загрузки дополнительных исходных файлов, а также своя собственная модульная система. Но код не был взаимозаменяемым, поэтому кто знает, что случится, когда вам захочется запустить ваш код Rhino в среде Node.js?

Стало очевидно, что для обеспечения функциональной совместимости нужен стандарт, который мог бы соблюдаться всеми реализациями JavaScript, позволяя нам использовать библиотеки во всех средах окружения. Именно для этого Кевин Денгур (Kevin Dangoor) запустил инициативу CommonJS. Он начал с *сообщения в блоге*, в котором ратовал за общий стандарт для интерпретаторов JavaScript и за то, чтобы разработчики скооперировались и написали соответствующую спецификацию:

> JavaScript нуждается в стандартном способе включения модулей и в том, чтобы эти модули существовали в отдельных пространствах имен. Существуют простые способы организации пространства имен, но нет стандартного программного способа загрузки модуля (однократной загрузки!).
>
> [Это] — не техническая проблема. Это предмет обсуждения людей, собравшихся вместе и принимающих решение сделать шаг вперед и приступить к совместному созданию чего-то более существенного и значимого.

Был составлен *перечень заинтересованных*, и родился CommonJS. Он быстро приобрел стартовый импульс с поддержкой от основных игроков. Теперь он де факто стал стандартом формата модулей для JavaScript с наращиваемым набором стандартов, включая интерфейсы ввода-вывода, сокет-потоки (Socket streams) и блочные тесты.

Объявление модуля

Модуль CommonJS объявляется довольно просто. Организация пространства имен осуществляется непосредственно в модуле; модули помещаются отдельно в разные файлы и выставляют переменные для публичного использования путем их добавления к определяемому в интерпретаторе объекту `exports`:

```
// maths.js
exports.per = function(value, total) {
  return( (value / total) * 100 );
};
// application.js
var Maths = require("./maths");
assertEqual( Maths.per(50, 100), 50 );
```

Для использования любых функций, определенных в модуле, нужно просто затребовать файл с помощью функции `require()`, сохраняя результат в локальной переменной. В показанном выше примере любые функции, экспортированные maths.js, доступны в переменной `Maths`. Ключевой особенностью является то, что модули имеют свое собственное пространство имен и будут работать на всех CommonJS-совместимых интерпретаторах JavaScript, таких как Narwhal и Node.js.

Модули и браузер

Но как это относится к JavaScript-разработке на стороне клиента? Многие разработчики видели последствия использования модулей на стороне клиента, а именно, что стандарт в его текущем состоянии требует, чтобы модули CommonJS загружались одновременно. Это требование вполне подходит для JavaScript, работающего на стороне сервера, но может стать весьма проблематичным в браузерах, поскольку оно блокирует пользовательский интерфейс и требует интерпретации сценариев, основанной на вычислении (eval-based compilation), чего всегда следует избегать. Команда CommonJS разработала спецификацию формата переноса модуля — *module transport format*, — обращенную к этой проблеме. Этот формат переноса заключает CommonJS-модули в функцию обратного вызова, чтобы допустить асинхронную загрузку на клиентские машины.

Возьмем показанный выше пример модуля. Мы можем заключить его в формат переноса, чтобы допустить асинхронную загрузку и сделать его привлекательным для браузера:

```
// maths.js
require.define("maths", function(require, exports){
  exports.per = function(value, total) {
    return( (value / total) * 100 );
  };
});
```

```
// application.js
require.define("application", function(require, exports){
  var per = require("./maths").per;
  assertEqual( per(50, 100), 50 );
}), ["./maths"]); // Перечисление зависимостей (maths.js)
```

Затем наши модули могут быть востребованы библиотекой загрузки модулей и выполнены в браузере. Все это действительно имеет большое значение. Мы не только разбили наш код на отдельные модульные компоненты, являющиеся ключом к разработке хороших приложений, но мы также добились управления зависимостями, изоляции области видимости и организации пространства имен. Разумеется, вышеупомянутые модули могут запускаться на браузерах, серверах, в обычных приложениях и в любой другой CommonJS-совместимой среде. Иными словами, теперь появилась возможность использовать один и тот же код как на серверной, так и на клиентской машине!

Загрузчики модулей

Для использования CommonJS-модулей на стороне клиента нужно воспользоваться библиотекой загрузки модулей. Существует множество вариантов, у каждого из которых имеются свои сильные и слабые стороны. Я рассмотрю наиболее популярные из них, а вы сможете выбрать, какой из них наиболее полно отвечает вашим потребностям.

Формат CommonJS-модуля пока еще подвергается постоянным изменениям, и на рассмотрении находятся разные предложения. Получается, что нет никакого официально одобренного транспортного формата, что, к сожалению, усложняет сложившуюся ситуацию. В этой неопределенной ситуации возникли две основные реализации модуля: Transport C и Transport D. При использовании какого-нибудь из изоляционных инструментов, упомянутых в предыдущем разделе, вы должны обеспечить генерацию изолированных модулей в формате, поддерживаемом вашим загрузчиком.

К счастью, многие загрузчики модулей поставляются с совместимыми инструментами изоляции или определяют поддерживаемые ими инструменты в своей документации.

Yabble

Yabble является превосходным и облегченным загрузчиком модулей. Его можно настроить либо на запрос модулей с использованием XHR, либо на использование `script`-тегов. Преимущество извлечения модулей с помощью XHR заключается в том, что их не нужно изолировать, т. е. заключать в транспортный формат. А недостатком является необходимость выполнения модулей с помощью функции `eval()`, что существенно затрудняет отладку. Кроме этого существуют кросс-доменные проблемы, особенно если вы используете CDN.

В идеале вариант XHR должен использоваться только для быстрой и черновой разработки, но ни в коем случае не при создании коммерческого продукта:

```
<script src="https://github.com/jbrantly/yabble/raw/master/lib/yabble.js">
</script>
<script>
  require.setModuleRoot("javascripts");

  // В модулях, заключенных в транспортный формат
  // мы можем использовать script-теги
  require.useScriptTags();

  require.ensure(["application"], function(require) {
    // Приложение загружено
  });
</script>
```

Показанный выше пример извлечет наш заключенный в транспортный протокол модуль приложения, а затем, перед запуском модуля, загрузит его зависимости. Мы можем загрузить модули, используя функцию `require()`:

```
<script>
  require.ensure(["application", "utils"], function(require) {
    var utils = require("utils");
    assertEqual( utils.per( 50, 200 ), 25 );
  });
</script>
```

Хотя `utils` запрашивается дважды, один раз встроенной функцией `require.ensure()` и второй раз — модулем приложения, наш сценарий достаточно «сообразителен», чтобы извлечь модуль только один раз. Перечень всех необходимых модулю зависимостей нужно обязательно указать в транспортной оболочке.

RequireJS

Хорошей альтернативой Yabble может послужить RequireJS, один из наиболее популярных загрузчиков. У RequireJS немного другой механизм получения загружаемых модулей, он придерживается формата асинхронного определения модулей — Asynchronous Module Definition, или AMD. Основное отличие, которое должно вас заинтересовать, заключается в том, что API вычисляет зависимости более настойчиво. На практике RequireJS полностью совместим с CommonJS-модулями, требуя только лишь другую изолированную транспортировку.

Для загрузки файлов JavaScript передайте их путевые имена функции `require()`, указав функцию обратного вызова, которая будет вызвана, когда будут загружены все зависимости:

```
<script>
  require(["lib/application", "lib/utils"], function(application, utils) {
    // Загружены!
  });
</script>
```

Как видно в показанном выше примере, приложение и его модули `utils` передаются функции обратного вызова в качестве аргументов и их не нужно извлекать с помощью функции `require()`.

Но можно запрашивать не только модули, RequireJS также поддерживает в качестве зависимостей обычные JavaScript-библиотеки, в особенности jQuery и Dojo. Другие библиотеки также будут работать, но они не будут правильно переданы в качестве аргументов требуемой функции обратного вызова. Но от любой библиотеки, имеющей зависимости, требуется использование формата модуля:

```
require(["lib/jquery.js"], function($) {
  // jQuery загружена
  $("#el").show();
});
```

Пути, предоставленные функции `require()`, относятся к текущему файлу или модулю, если только они начинаются с символа `/`. Чтобы помочь с оптимизацией, RequireJS предлагает вам помещать ваш исходный загрузчик сценария в отдельный файл. Библиотека даже предоставляет для этого краткую запись: атрибут `data-main`:

```
<script data-main="lib/application" src="lib/require.js"></script>
```

Установка атрибута `data-main` предписывает RequireJS считать `script`-тег неким подобием вызова функции `require()` и загружать то, что указано в значении атрибута. В данном случае будет загружен сценарий lib/application.js, который, в свою очередь, загрузит все остальное приложение:

```
// Внутри lib/application.js
require(["jquery", "models/asset", "models/user"], function($, Asset,
User) {
    //...
});
```

Итак, мы рассмотрели запрос модулей, а как насчет их определений? Как уже ранее утверждалось, RequireJS использует для модулей немного другой синтаксис. Вместо использования `require.define()`, нужно использовать обыкновенную функцию `define()`. Поскольку модули находятся в отдельных файлах, они не нуждаются в явном обозначении. Сначала идут зависимости в виде массива строк, а затем следует функция обратного вызова, содержащая фактический модуль. Как и в RequireJS-функции `require()`, зависимости передаются в виде аргументов функции обратного вызова:

```
define(["underscore", "./utils"], function(_, Utils) {
  return({
    size: 10
  })
});
```

По умолчанию переменная `exports` отсутствует. Для выставления переменных за пределы модуля просто верните данные из функции. Выгода от RequireJS-модулей

состоит в том, что они уже изолированы, поэтому вам не приходится волноваться насчет транспортных форматов для браузера. Но следует предупредить, что этот API несовместим с модулями CommonJS, т. е. вы не можете совместно использовать модули Node.js и браузера. Но не все еще потеряно: в RequireJS имеется уровень, совместимый с CommonJS-модулями, нужно просто заключить имеющиеся у вас модули в функцию `define()`:

```
define(function(require, exports) {
  var mod = require("./relative/name");
  exports.value = "exposed";
});
```

Аргументы, передаваемые функциям обратного вызова, должны быть точно такими же, как показано выше, т. е. `require` и `exports`. Ваши модули могут затем продолжать использовать эти переменные как обычно, без каких-либо изменений.

Изолирование модулей

На данный момент у нас уже есть управление зависимостями и способ организации адресного пространства, но по-прежнему не решена исходная проблема: все эти HTTP-запросы. Любые модули, от которых мы зависим, должны загружаться с удаленной машины, и даже при том, что все это происходит в асинхронном режиме, все же приходится иметь дело с большими издержками производительности, замедляя запуск вашего приложения.

Мы также изолируем наши модули, помещая их в транспортный формат, который, будучи необходимым для асинхронной загрузки, слишком многословен. Давайте убьем сразу двух зайцев, воспользовавшись действием на стороне сервера для объединения модулей в одном файле. Это означает, что браузеру придется извлекать только один ресурс для загрузки всех модулей, что будет намного эффективнее. Доступные средства компоновки достаточно «сообразительны», они не только связывают модули произвольным образом, но и проводят их статический анализ для рекурсивного решения их зависимостей. Они также берут на себя помещение модулей в транспортный формат, экономя время на набор кода.

Вдобавок к объединению многие компоновщики модулей также поддерживают миниатюризацию, т. е. дальнейшее сокращение требуемого размера. Фактически некоторые средства, такие как rack-modulr и Transporter, интегрируются с вашим веб-сервером, справляясь с обработкой модулей в автоматическом режиме при первом же их востребовании.

Вот, например, простой Rack-сервер модулей CommonJS, использующий rack-modulr:

```
require "rack/modulr"
use Rack::Modulr, :source => "lib", :hosted_at => "/lib"
run Rack::Directory.new("public")
```

Сервер можно запустить с помощью команды `rackup`. Любые содержащиеся внутри папки lib CommonJS-модули теперь автоматически объединены со всеми свои-

ми зависимостями и помещены в транспортную функцию обратного вызова. Затем наш загрузчик сценариев может запросить модули по мере необходимости, загружая их в страницу:

```
>> curl "http://localhost:9292/lib/application.js"
     require.define("maths"....
```

Если вы не работаете с Ruby, есть множество других доступных для выбора вариантов. FlyScript является средством изоляции CommonJS-модулей, написанным на PHP, Transporter является одним из средств, предназначенных для JSGI-серверов, а Stitch интегрируется с Node.js-серверами.

Альтернативы модулям

Вы можете принять решение не следовать маршрутом использования модулей, возможно, потому что у вас уже есть большой объем готового кода и библиотек для его поддержки, которые не могут быть легко сконвертированы. К счастью, есть ряд вполне достойных альтернатив, например Sprockets. Это средство добавляет поддержку синхронной функции `require()` для вашего JavaScript. Комментарии, начинающиеся с символов `//=`, работают как директивы для препроцессора Sprockets. Например, директива `//= require` предписывает Sprockets найти в его пути загрузки библиотеку, извлечь ее и включить ее в код:

```
//= require <jquery>
//= require "./states"
```

В показанном выше примере jquery.js является путем загрузки, а states.js требуется относительно текущего файла. Sprockets достаточно «сообразительное» средство, чтобы включить библиотеку только один раз, независимо от того, сколько раз она будет востребована. Как и все средства изоляции CommonJS-модулей, Sprockets поддерживает кэширование и миниатюризацию. В процессе разработки ваш сервер может анализировать и объединять файлы по мере потребности в ходе загрузки страницы. Когда сайт используется, файлы JavaScript могут быть предварительно объединены и обслужены в статическом режиме, повышая производительность.

Хотя Sprockets является средством командной строки, существует ряд отличных компоновок с Rack и Rails, например rack-sprockets. Есть даже несколько PHP-реализаций. Недостаток Sprockets и, конечно, всех этих средств изоляции модулей состоит в том, что все ваши файлы JavaScript нуждаются в предварительной обработке, либо сервером, либо средством командной строки.

LABjs

LABjs — одно из самых простых из имеющихся решений по управлению зависимостями. Ему не нужно привлекать что-либо на стороне сервера или использовать CommonJS-модули. Загрузка ваших сценариев с помощью LABjs сокращает блокировку ресурса во время загрузки страницы, что является простым и эффективным

способом оптимизации производительности вашего сайта. По умолчанию LABjs загрузит и выполнит сценарии в параллельном режиме с максимально возможной скоростью. Но вы можете легко указать порядок выполнения, если у каких-нибудь сценариев есть зависимости:

```
<script>
  $LAB
  .script('/js/json2.js')
  .script('/js/jquery.js').wait()
  .script('/js/jquery-ui.js')
  .script('/js/vapor.js');
</script>
```

В показанном выше примере все сценарии загружаются параллельно, но LABjs обеспечивает выполнение jquery.js до выполнения jquery-ui.js и vapor.js. API невероятно прост и лаконичен, но если вы хотите научиться более сложным вещам, таким как поддержка встроенных сценариев, обратитесь к документации.

FUBC

При использовании всех этих загрузчиков сценариев нужно не упустить из виду еще одну вещь: во время загрузки страницы пользователи могут увидеть демонстрационный, ни на что не реагирующий контент (flash of unbehaviored content, FUBC) – т. е. бегло просмотреть примитивную страницу до того, как будет выполнен любой код JavaScript. С этим не будет проблемы, если вы не зависите от JavaScript при задании стиля или при обработке исходной страницы. Но если вы без этого не обходитесь, решите вопрос путем установки некоторых исходных стилей в CSS, возможно, скрывая ряд элементов или отображая краткую заставку на время загрузки.

7 Работа с файлами

Традиционно доступ к файлам и работа с ними считались чем-то из области локальных приложений, а при работе с веб-технологиями все это ограничивалось функциональными возможностями, предоставляемыми дополнительными модулями, подобными Adobe Flash. Но все изменилось с приходом HTML5, который существенно расширил область действий разработчиков при работе с файлами, продолжая стирать границы между локальным и веб-приложениями. Работая с современными браузерами, пользователи могут перетаскивать файлы на страницу, вставлять структурированные данные и видеть работающие в режиме реального времени индикаторы выполнения, отображающие передачу файла на сервер, осуществляемую в фоновом режиме.

Поддержка браузерами

Поддержка новых файловых API, имеющихся в HTML5, не является всеобщей, но вполне достаточное количество браузеров имеют их реализации, поэтому есть смысл потратить время на интеграцию этих интерфейсов.

- Firefox >= 3.6
- Safari >= 6.0
- Chrome >= 7.0
- IE: не поддерживает
- Opera >= 11.1

Поскольку IE их пока не поддерживает, вам придется использовать передовые расширения, дающие пользователям возможность традиционного файлового ввода для передачи файлов, а также позволяющие применять более совершенный прием перетаскивания файлов. Определить наличие поддержки нетрудно, нужно просто проверить наличие соответствующих объектов:

```
if (window.File && window.FileReader && window.FileList) {
  // API поддерживаются
}
```

Получение информации о файлах

Главная мера безопасности, заложенная в работу с файлами в HTML5, заключается в том, что доступ может быть получен только к файлам, выбранным пользователями. Файл можно выбрать путем перетаскивания его в браузер, выбрав его в файловом вводе или вставив его в веб-приложение. Хотя при показе файловой системы в JavaScript проделывается определенная работа, доступ всегда организуется в «песочнице». Понятно, что предоставление JavaScript возможности чтения и записи произвольных файлов на вашей системе было бы с точки зрения безопасности очень большим упущением.

Файлы представляются в HTML5 посредством объектов `File`, имеющих три атрибута.

`name`

Имя файла в виде строки, доступной только для чтения

`size`

Размер файла в виде целого числа, доступного только для чтения

`type`

MIME-тип файла в виде строки, доступной только для чтения, или пустая строка (`""`), если тип не может быть определен

Из соображений безопасности информация о пути к файлу никогда не раскрывается.

Несколько файлов показываются как объект `FileList`, который, по сути, может рассматриваться как массив объектов `File`.

Ввод файлов

Ввод файлов, речь о котором шла с самого зарождения веб-технологий, является традиционным способом предоставления пользователям возможности передавать файлы. HTML5 усовершенствовал его, избавив от ряда недостатков. Одним из давних опасений разработчиков было разрешение передачи сразу нескольких файлов. В прежние времена разработчикам приходилось прибегать к множественным вводам файлов или полагаться на дополнительные модули вроде Adobe Flash. HTML5 справляется с этим с помощью атрибута `multiple`. Указывая `multiple` применительно к вводу файла, вы сигнализируете браузеру, что пользователю будет разрешено выбрать несколько файлов. Устаревшие браузеры, не поддерживающие HTML5, этот атрибут проигнорируют:

```
<input type="file" multiple>
```

Но пользовательский интерфейс несовершенен; чтобы выбрать несколько файлов, пользователям нужно удерживать нажатой клавишу `Shift`. Вполне возможно, что вам придется это разъяснить пользователям путем показа соответствующего сообщения. В сети Facebook выяснили, что 85% пользователей, выкладывающих фо-

тографии, делают это *поштучно*. После добавления подсказки, объясняющей, как нужно выбирать сразу несколько фотографий для передачи на сервер (см. рис. 7.1), показатели упали с 85 до 40%. Еще одной проблемой для разработчиков было отсутствие информации о том, какие файлы были выбраны. Зачастую полезно проверять приемлемость выбранных файлов, гарантируя тем самым конкретный тип или не превышение конкретного размера. Теперь HTML5 делает это возможным путем предоставления доступа к вводу выбранных файлов, используя атрибут `files`.

Доступный только для чтения атрибут `files` возвращает `FileList`, элементы которого можно последовательно перебрать, выполняя проверку приемлемости, с последующим информированием пользователя о результате:

```
var input = $("input[type=file]");
input.change(function(){
  var files = this.files;
  for (var i=0; i < files.length; i++)
    assert( files[i].type.match(/image.*/) )
});
```

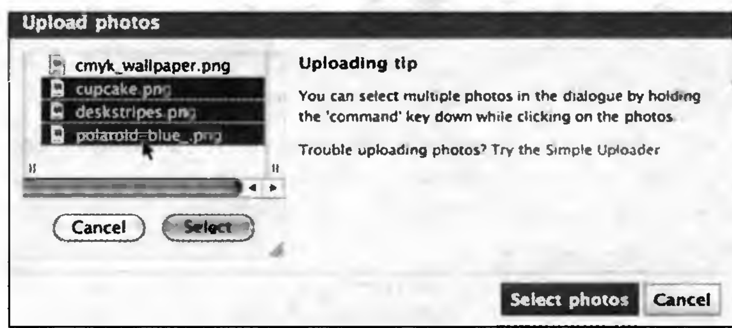

Рис. 7.1. Выкладывание нескольких файлов на Facebook

Но наличие доступа к выбранным файлам не ограничивает ваши действия проверкой приемлемости. Например, вы можете прочитать содержимое файла, отображая предварительное представление выкладываемого файла. Или вместо блокирования пользовательского интерфейса во время передачи файла, вы можете выложить его в фоновом режиме, используя Ajax и показывая индикатор выполнения. Все это и кое-что еще рассматривается в последующих разделах.

Перетаскивание

Поддержка перетаскивания была изначально «придумана» и реализована компанией Microsoft в далеком 1999 году для Internet Explorer 5.0, и IE с тех пор ее поддерживает. В спецификации HTML5 просто задокументировано то, что там уже было, и теперь в Safari, Firefox и Chrome добавлена поддержка, имитирующая реализацию Microsoft. Но, мягко говоря, спецификация сильно запутана и требует

выворачиваться наизнанку для удовлетворения своих зачастую бессмысленных требований.

Есть не менее семи событий, связанных с перетаскиванием: `dragstart`, `drag`, `dragover`, `dragenter`, `dragleave`, `drop` и `dragend`. В следующих разделах я подробно остановлюсь на каждом из них.

Даже если ваш браузер не поддерживает файловые API, имеющиеся в HTML5, вполне вероятно, что вы все же можете использовать API перетаскивания. На данный момент к браузерам предъявляются следующие требования:

- Firefox >= 3.5
- Safari >= 3.2
- Chrome >= 7.0
- IE >= 6.0
- Opera: не поддерживает

Захват и перетаскивание

Перетаскивание осуществляется довольно просто. Чтобы сделать элемент перетаскиваемым, нужно присвоить его атрибуту `draggable` значение `true`.

```
<div id="dragme" draggable="true">Перетащи меня!</div>
```

Теперь нам нужно связать перетаскиваемый элемент с какими-нибудь данными. Мы можем сделать, это прислушиваясь к событию `dragstart` и вызывая обрабатывающую событие функцию `setData()`:

```
var element = $("#dragme");
element.bind("dragstart", function(event){
  // Мы не хотим использовать абстракцию jQuery
  event = event.originalEvent;
  event.dataTransfer.effectAllowed = "move";
  event.dataTransfer.setData("text/plain", $(this).text());
  event.dataTransfer.setData("text/html", $(this).html());
  event.dataTransfer.setDragImage("/images/drag.png", -10, -10);
});
```

jQuery предоставляет абстракцию события, не содержащую нужного нам объекта `dataTransfer`. Для нашего удобства у абстрагированного события есть атрибут `originalEvent`, который мы можем использовать для доступа к API перетаскивания.

Как показано выше, у события есть объект `dataTransfer`, имеющий различные нужные нам функции перетаскивания. Функции `setData()` передаются сведения о типе — mimetype и строковые данные. В данном случае для события `drag` установлены некие данные типа `text` и `text/html`. Когда элементы освобождаются и инициируется событие `drop`, мы можем прочесть эти данные. Таким же образом, если элемент перетащен за пределы окна браузера, другие приложения могут обработать освобожденные данные в соответствии с теми типами данных, которые ими поддерживаются.

При перетаскивании текста следует использовать тип `text/plain`. Его всегда рекомендуется устанавливать в качестве альтернативного для приложений или целей перетаскивания, не поддерживающих ни одного другого формата. Перетаскиваемые ссылки должны иметь два формата: `text/plain` и `text/uri-list`. Для перетаскивания нескольких ссылок нужно присоединять каждую ссылку с использованием новой строки:

```
// Перетаскиваемые ссылки
event.dataTransfer.setData("text/uri-list", "http://example.com");
event.dataTransfer.setData("text/plain", "http://example.com");

// Несколько ссылок разделены символом новой строки
event.dataTransfer.setData("text/uri-list",
                           "http://example.com\nhttp://google.com");
event.dataTransfer.setData("text/plain",
                           "http://example.com\nhttp://google.com");
```

Дополнительная функция `setDragImage()` управляет тем, что будет отображено под указателем мыши в процессе перетаскивания. Ей передается источник изображения и координаты `x/y` позиции изображения относительно курсора. Если аргументы не переданы, вы получаете просто призрачный клон перетаскиваемого элемента. Альтернативой функции `setDragImage()` является функция `addElement(element, x, y)`, которая использует заданный элемент для обновления отклика перетаскивания. Иными словами, вы можете предоставить свой собственный элемент, отображаемый в процессе операций перетаскивания.

Вы можете также позволить пользователям перетаскивать файлы за пределы окна браузера, установив тип `DownloadURL`. Вы можете указать в URL местонахождение файла, который браузер впоследствии перешлет. В Gmail это используется для создания эффектного приема, позволяющего пользователям перетаскивать почтовые прикрепления непосредственно из браузера на Рабочий стол.

Плохо только, что на данный момент все это поддерживается только браузером Chrome и является его незадокументированным свойством. Но это не мешает его использовать, и есть надежда, что в будущем поддержку этого свойства добавят и в других браузерах. Значением формата `DownloadURL` служит список информации о файле: mime-тип, имя и местонахождение, где элементы списка отделены друг от друга двоеточием (`:`).

```
$("#preview").bind("dragstart", function(e){
  e.originalEvent.dataTransfer.setData("DownloadURL", [
    "application/octet-stream",    // MIME-тип
    "File.exe",                    // Имя файла
    "http://example.com/file.png"  // Местонахождение файла
  ].join(":"));
});
```

Полную версию примера API перетаскивания, имеющегося в HTML5, можно увидеть в assets/ch07/drag.html.

Освобождение после перетаскивания

API перетаскивания позволяет прислушиваться к событиям освобождения (drop) перетаскиваемых файлов или других элементов. Именно здесь мы начинаем наблюдать некоторые странности API перетаскивания. Для полного инициирования события drop вам нужно отменить все, что делается по умолчанию для двух событий: dragover и dragenter. Вот как, к примеру, отменяются эти два события:

```
var element = $("#dropzone");
element.bind("dragenter", function(e){
  // Отмена события
  e.stopPropagation();
  e.preventDefault();
});
element.bind("dragover", function(e){
  // Установка указателя мыши
  e.originalEvent.dataTransfer.dropEffect = "copy";
  // Отмена события
  e.stopPropagation();
  e.preventDefault();
});
```

Можно также установить dropEffect, т. е. вид указателя мыши при событии dragover, как показано выше. Прислушиваясь к событиям dragenter и dragleave и переключая установку классов для элементов под указателем, вы можете дать пользователям визуальную индикацию приемлемости той или иной области для освобождения перетаскиваемых файлов.

Но только отменив события dragenter и dragover, мы можем начать прислушиваться к событиям drop. Событие drop будет инициироваться, когда перетаскиваемый элемент или файл освобождается над целевым элементом. Объект dataTransfer события drop содержит свойство files, возвращающее список FileList, содержащий все перетащенные файлы:

```
element.bind("drop", function(event){
  // Отмена перенаправления
  event.stopPropagation();
  event.preventDefault();
  event = event.originalEvent;
  // Доступ к перетащенным файлам
  var files = event.dataTransfer.files;
  for (var i=0; i < files.length; i++)
    alert("Dropped " + files[i].name);
});
```

Доступ к данным, не являющимся файлами, можно получить с помощью функции dataTransfer.getData(), передав ей поддерживаемый формат. Если этот формат недоступен, функция просто вернет значение undefined (неопределенный).

```
var text = event.dataTransfer.getData("Text");
```

Объект `dataTransfer` имеет свойство `types`, предназначенное только для чтения, которое возвращает список `DOMStringList` (являющийся массивом) mime-форматов, установленных для события `dragstart`. Кроме того, если были перетащены какие-нибудь файлы, одним из типов будет строка `"Files"`.

```
var dt = event.dataTransfer
for (var i=0; i < dt.types.length; i++)
  console.log( dt.types[i], dt.getData(dt.types[i]) );
```

Полный пример, относящийся к событию `drop`, можно найти в **assets/ch07/drop.html**.

Отмена действия по умолчанию при перетаскивании

По умолчанию перетаскивание файла на веб-страницу заставляет браузер перейти к отображению этого файла. Мы хотим воспрепятствовать такому поведению, поскольку не хотим, чтобы пользователи уходили из нашего приложения, если они попали мимо области, над которой нужно было освободить перетаскиваемый элемент. Сделать это совсем нетрудно, нужно просто отменить относящееся к телу документа (`body`) событие `dragover`.

```
$("body").bind("dragover", function(e){
  e.stopPropagation();
  e.preventDefault();
  return false;
});
```

Копирование и вставка

Вдобавок к интеграции с Рабочим столом в области перетаскивания некоторые браузеры обеспечивают поддержку копирования и вставки. API не подвергался стандартизации и не является частью спецификации HTML5, поэтому вам нужно будет определить, как именно угодить различным браузерам.

И опять здесь проявилось своеобразие IE, который, как ни странно, был пионером в этом деле, начиная с версии IE 5.0. WebKit взял API компании Microsoft и немного его усовершенствовал, поставив в один ряд с API перетаскивания. Оба интерфейса практически идентичны, за исключением разных объектов: вместо `dataTransfer` используется `clipboardData`.

В Firefox поддержка пока отсутствует, и хотя в нем есть собственный API для обращения к буферу обмена, он слишком неуклюжий, если не сказать хуже. WebKit (Safari и Chrome) имеет достаточно хорошую поддержку, и я допускаю, что W3C в конечном итоге превратит его в стандарт, применяемый к API для работы с буфером обмена. Поддержка имеется в следующих браузерах:

- Safari >= 6.0
- Chrome (только вставка)
- Firefox: не поддерживает
- IE >= 5.0 (другой API)

Копирование

Есть два события, связанных с копированием, и два события, связанных с вырезкой фрагмента:

- beforecopy
- copy
- beforecut
- cut

Судя по их именам, beforecopy и beforecut инициируются перед любыми операциями с буфером обмена, позволяя вам отключать их в случае необходимости. Когда пользователь копирует какой-нибудь выделенный текст, инициируется событие copy, предоставляя вам объект clipboardData, который может использоваться для установки в буфер обмена пользовательских данных. Аналогично объекту dataTransfer, у объекта clipboardData имеется функция setData(), которой передается mime-формат и строковое значение. Если вы планируете вызывать эту функцию, нужно отменить исходное событие copy, предотвратив действия по умолчанию.

Вместо события IE устанавливает объект clipboardData для окна. Нужно проводить проверку на присутствие объекта в отношении события и, если он там отсутствует, на его присутствие в отношении окна.

Firefox инициализирует событие copy, но не дает вам доступа к объекту Data, принадлежащему буферу обмена. Chrome даст вам объект, но проигнорирует любые устанавливаемые для него данные.

```
$("textarea").bind("copy", function(event){
  event.stopPropagation();
  event.preventDefault();

  var cd = event.originalEvent.clipboardData;

  // Для IE
  if ( !cd ) cd = window.clipboardData;

  // Для Firefox
  if ( !cd ) return;
  cd.setData("text/plain", $(this).text());
});
```

Учитывая скорость обновления браузеров, вполне вероятно, что поддержка этих свойств вскоре станет стандартом. Если вы хотите добавить поддержку копирования-вставки к вашему приложению, нужно присмотреться к текущей ситуации, в которой вы находитесь.

Вставка после копирования

Со вставкой связаны два события: beforepaste и paste. Событие paste инициируется, когда пользователь уже отпустил кнопку для вставки, но перед тем как будут

вставлены какие-нибудь данные. И тут мы снова сталкиваемся с разной реализацией в разных браузерах. Chrome инициирует событие, даже если фокус не получен ни одним из элементов. А IE и Safari требуют наличия элемента, имеющего фокус.

API очень похож на API события drop. У события есть свойство clipboardData, которое дает вам доступ к вставленным данным с помощью функции getData(), которой передается mime-формат. К сожалению, сколько я ни пробовал, свойство types всегда имеет значение null, поэтому вы не можете увидеть, какие mime-типы доступны в отношении данных буфера обмена. Пока событие не будет отменено, процесс вставки будет выполняться в обычном режиме, и данные будут вставлены в элемент, имеющий фокус:

```
$("textarea").bind("paste", function(event){
  event.stopPropagation();
  event.preventDefault();
  event = event.originalEvent;
  var cd = event.clipboardData;
  // Для IE
  if ( !cd ) cd = window.clipboardData;
  // Для Firefox
  if ( !cd ) return;
  $("#result").text(cd.getData("text/plain"));
  // Поддержка события вставки файла  в Safari
  var files = cd.files;
});
```

Начальные версии WebKit предоставляют доступ к свойству files объекта clipboardData, позволяя поддерживать вставку файлов в ваше приложение. Я ожидаю, что и другие браузеры последуют примеру, как только будет утвержден стандарт.

Итак, есть ли какая-нибудь возможность заставить все это работать на кроссбраузерной основе? Конечно, есть, и существует сразу несколько приемов. К примеру, Cappuccino вообще обходит семейство событий oncopy, и просто перехватывает ввод с клавиатуры. При обнаружении клавиатурной комбинации Command/Ctrl + v эта библиотека переводит фокус на скрытое поле ввода, которое заполняется вставляемыми данными. Этот прием работает на любом браузере, но только для вставок, инициированных с клавиатуры, а не из меню.

Чтение файлов

Как будет получена ссылка File, можно будет создать экземпляр объекта FileReader для считывания содержимого файла в память. Файлы считываются в асинхронном режиме — экземпляру FileReader предоставляется функция обратного вызова, которая вызывается, когда файл будет готов.

Для чтения данных файла объект FileReader предоставляет вам четыре функции. Какую из них использовать, зависит от формата данных, который нужно вернуть.

`readAsBinaryString(Blob|File)`

Возвращает данные файла или блоба в виде двоичной последовательности. Каждый байт представлен целым числом в диапазоне от 0 до 255.

`readAsDataURL(Blob|File)`

Возвращает данные файла или блоба, закодированные в виде URL данных. К примеру, эта функция может использоваться в качестве значения атрибута `src`, указывающего на изображение.

`readAsText(Blob|File, encoding='UTF-8')`

Возвращает данные файла или блоба в виде текстовой строки. По умолчанию строка декодируется как UTF-8.

`readAsArrayBuffer(Blob|File)`

Возвращает данные файла или блоба в виде объекта `ArrayBuffer`. Реализация в большинстве браузеров отсутствует.

Экземпляры `FileReader` содержат ряд событий, инициируемых при вызове одной из вышеперечисленных функций. Самыми главными из них, которые должны вас заинтересовать, являются:

`onerror`

Вызывается при возникновении ошибки

`onprogress`

Вызывается периодически по мере чтения данных

`onload`

Вызывается при доступности данных

Для использования `FileReader` нужно создать экземпляр, добавить события и воспользоваться одной из функций чтения. Событие `onload` содержит свойство `result`, устанавливающее считываемые данные в соответствующем формате:

```
var reader = new FileReader();
reader.onload = function(e) {
  var data = e.target.result;
};
reader.readAsDataURL(file);
```

Например, можно воспользоваться переменной `data`, как в примере, показанном выше, в качестве источника изображения, отображающего свернутое в значок изображение указанного файла:

```
var preview = $("img#preview")
// Проверка, относится ли файл к типу image и не будет ли
// предварительный просмотр
// вызывать проблемы у браузера из-за слишком большого размера
```

```
if (file.type.match(/image.*/) &&
      file.size < 50000000) {
  var reader = new FileReader();
  reader.onload = function(e) {
    var data    = e.target.result;
    preview.attr("src", data);
  };
  reader.readAsDataURL(file);
}
```

Блобы и части

Иногда предпочтительнее считывать в память часть файла, а не весь файл целиком. Файловые API, имеющиеся в HTML5, поддерживают в этом плане удобную функцию `slice()`. В качестве первого аргумента ей передается стартовый байт, а в качестве второго — байтовое смещение (или длина части файла). Эта функция возвращает объект `Blob`, в отношении которого можно применять методы, поддерживающие объект `File`, такие как `FileReader`. Например, мы можем буферизовать файл, осуществив следующее чтение:

```
var bufferSize = 1024;
var pos = 0;

var onload = function(e){
  console.log("Прочитано: ", e.target.result);
};

var onerror = function(e){
  console.log("Ошибка!", e);
};

while (pos < file.size) {
  var blob = file.slice(pos, bufferSize);

  var reader = new FileReader();
  reader.onload  = onload;
  reader.onerror = onerror;
  reader.readAsText(blob);
  pos += bufferSize;
}
```

Из этого примера можно увидеть, что экземпляр `FileReader` можно использовать только один раз, после чего нужно создавать новый экземпляр.

Полную версию примера можно найти в assets/ch07/slices.html. Нужно иметь в виду, что файл не может быть прочитан, если «песочница» является локальной. Иными словами, если slices.html считывается с диска, а не с хоста, чтение не состоится и будет выдано событие `onerror`.

Собственные кнопки просмотра

Открытие диалогового окна для просмотра файла программным способом стало уже устоявшейся практикой. Иными словами, по щелчку на стилизованных кнопках **Просмотр** (Browse) или **Вложение** (Attachment) тут же открывается диалоговое окно, при этом пользователь избавляется от необходимости заниматься традиционным вводом файла. Но из соображений безопасности все не так легко, как кажется на первый взгляд. С вводом файла не связана функция просмотра, и, за исключением Firefox, вы не можете просто инициировать свое собственное событие щелчка при вводе файла.

Рассматриваемое решение может показаться неким программистским трюком, но оно вполне справляется с задачей. Когда пользователь проводит указатель мыши над кнопкой просмотра, наложите на эту же область прозрачный ввод файла, имеющий ту же позицию и размеры, что и у кнопки. Этот прозрачный ввод файла будет перехватывать любое событие щелчка, открывая диалоговое окно просмотра.

Внутри папки assets/ch07 вы найдете файл jquery.browse.js, в котором содержится дополнительный модуль jQuery, который занимается выполнением именно этой задачи. Для создания собственной кнопки просмотра нужно вызвать в отношении экземпляра `jQuery` функцию `browseElement()`. Эта функция вернет файловый ввод, который можно добавить к слушателю события `change`, определяя момент выбора пользователем каких-либо файлов.

```
var input = $("#attach").browseElement();
input.change(function(){
  var files = $(this).attr("files");
});
```

Что может быть проще при полной кроссбраузерной поддержке!

Передача файлов

Возможность передачи файлов была частью спецификации XMLHttpRequest Level 2. Передача файлов долгое время была головной болью для пользователей. После просмотра и выбора файла происходила перезагрузка страницы, и им приходилось долгое время ждать, пока не завершится передача файла, не имея при этом ни индикатора выполнения, ни какой-то ответной реакции, т. е. никаких удобств использования. К счастью для нас, в XHR 2 эта проблема была решена. Эта технология позволяет передавать файлы в фоновом режиме и даже дает нам события хода выполнения, позволяя предоставить пользователю индикатор выполнения, работающий в реальном времени. Все это имеет неплохую поддержку во всех основных браузерах:

- Safari >= 5.0
- Firefox >= 4.0
- Chrome >= 7.0

- IE: не поддерживает
- Opera: не поддерживает

Передача файлов может быть осуществлена посредством существующего XMLHttpRequest API с использованием функции `send()` или, альтернативно, с использованием экземпляра `FormData`. Экземпляр объекта `FormData` просто представляет содержимое формы в легком и удобном интерфейсе. Вы можете создать объект `FormData` с нуля или путем передачи ему существующего элемента `form` при создании экземпляра объекта:

```
var formData = new FormData($("form")[0]);

// Данные формы можно добавить в виде строк
formData.append("stringKey", "stringData");

// И даже добавить объекты File
formData.append("fileKey", file);
```

После завершения создания экземпляра `FormData` его можно передать методом POST на ваш сервер, используя XMLHttpRequest. Если для Ajax-запросов используется jQuery, вам нужно будет установить для настройки `processData` значение `false`, чтобы код jQuery не пытался перевести предоставленные данные в последовательный формат. Заголовок `Content-Type` устанавливать не нужно, поскольку браузер установит его автоматически в значение `multipart/form-data`, наряду с установкой границы многокомпонентного содержимого:

```
jQuery.ajax({
  data: formData,
  processData: false,
  url: "http://example.com",
  type: "POST"
})
```

Альтернативой использованию `FormData` является передача файла непосредственно в функцию `send()`XHR-объекта:

```
var req = new XMLHttpRequest();
req.open("POST", "http://example.com", true);
req.send(file);
```

Или, используя Ajax API, принадлежащий jQuery, вы можете переслать файлы следующим образом:

```
$.ajax({
  url: "http://example.com",
  type: "POST",
  success: function(){ /* ... */ },
  processData: false,
  data: file
});
```

Стоит заметить, что эта передача немного отличается от традиционной передачи с типом содержимого `multipart/formdata`. Обычно информация о файле, такая как имя, включается в передачу. Но в данном случае этого не происходит — передаются только данные файла. Для передачи информации о файле можно установить собственные заголовки, например `X-File-Name`. Наши сервера могут прочитать эти заголовки и правильно обработать файл:

```
$.ajax({
  url: "http://example.com",
  type: "POST",
  success: function(){ /* ... */ },
  processData: false,
  contentType: "multipart/form-data",
  beforeSend: function(xhr, settings){
    xhr.setRequestHeader("Cache-Control", "no-cache");
    xhr.setRequestHeader("X-File-Name", file.fileName);
    xhr.setRequestHeader("X-File-Size", file.fileSize);
  },
  data: file
});
```

К сожалению, многие серверы будут испытывать проблемы с получением переданных им файлов, поскольку чистые данные являются менее известным форматом, чем `multipart` или закодированные в URL параметры формы. При использовании данного метода вам может понадобиться проводить разбор запроса своими силами. По этой причине я являюсь сторонником использования объектов `FormData` и отправки передаваемых файлов в последовательном формате, в виде запроса `multipart/formdata`. В assets/ch07 вы найдете файл jquery.upload.js, дополнительный модуль jQuery, абстрагирующий передачу файлов в простой интерфейс `$.upload(url, file)`.

Индикатор выполнения на основе Ajax

Спецификация XHR Level 2 добавляет поддержку для событий хода выполнения progress, как для запросов на загрузку, так и для запросов на передачу. Это дает возможность создания индикатора передачи файла, работающего в реальном времени, предоставляя пользователям представление о времени, оставшемся до завершения передачи файла.

Для прислушивания к событию хода выполнения при запросе на загрузку добавьте слушателя непосредственно к XHR-экземпляру:

```
var req = new XMLHttpRequest();
req.addEventListener("progress", updateProgress, false);
req.addEventListener("load", transferComplete, false);
req.open();
```

Что касается прислушивания к событию progress при передаче файла, то слушателя нужно добавлять к свойству upload XHR-экземпляра:

```
var req = new XMLHttpRequest();
req.upload.addEventListener("progress", updateProgress, false);
req.upload.addEventListener("load", transferComplete, false);
req.open();
```

Событие load будет инициировано, как только запрос на передачу будет завершен, но перед тем как сервер выдаст ответ. Мы можем добавить его к jQuery, поскольку XHR-объект и установки переданы функции обратного вызова beforeSend. Полная версия примера, включая самостоятельно созданные заголовки, имеет следующий вид:

```
$.ajax({
  url: "http://example.com",
  type: "POST",
  success: function(){ /* ... */ },
  processData: false,
  dataType: "multipart/form-data",
  beforeSend: function(xhr, settings){
    var upload = xhr.upload;
    if (settings.progress)
      upload.addEventListener("progress", settings.progress, false);
    if (settings.load)
      upload.addEventListener("load", settings.load, false);
    var fd = new FormData;
    for (var key in settings.data)
      fd.append(key, settings.data[key]);
    settings.data = fd;
  },
  data: file
});
```

В событии progress содержится позиция состояния передачи (т. е. количество переданных байтов) и общее количество (размер запроса на передачу в байтах). Эти два свойства можно использовать для вычисления хода выполнения передачи в процентах:

```
var progress = function(event){
  var percentage = Math.round((event.position / event.total) * 100);
  // Установка индикатора выполнения
}
```

В действительности у события есть метка времени, поэтому если вы записываете время начала передачи, можно создать элементарный показатель расчетного времени окончания — estimated time of completion (ETA):

```
var startStamp = new Date();
var progress = function(e){
  var lapsed = startStamp - e.timeStamp;
  var eta    = lapsed * e.total / e.position - lapsed;
};
```

Но этот расчет вряд ли будет точен при небольших (и более быстрых) передачах. По моему мнению, ETA стоит показывать, только если передача займет примерно более четырех минут. Обычно достаточно процентного индикатора выполнения, поскольку он дает пользователю четкую визуальную картину того, сколько еще продлится передача.

Сценарий передачи файлов с использованием перетаскивания и jQuery

Итак, давайте применим на практике все знания и создадим сценарий передачи файлов с использованием перетаскивания. Нам понадобятся несколько библиотек: jquery.js в качестве основы, jquery.ui.js для индикатора выполнения, jquery.drop.js для абстрагирования API перетаскивания и jquery.upload.js для передачи по технологии Ajax. Вся логика будет находиться внутри функции `jQuery.ready()`, поэтому она должна быть запущена по готовности DOM-модели:

```
//= require <jquery>
//= require <jquery.ui>
//= require <jquery.drop>
//= require <jquery.upload>

jQuery.ready(function($){
  /* ... */
});
```

Создание области освобождения перетаскиваемого элемента

Нам нужно, чтобы пользователи могли перетаскивать файлы в элемент `#drop`, поэтому давайте превратим его в область освобождения. Нам нужно привязаться к событию `drop`, отменить его и извлечь имена всех освобожденных файлов, которые затем передаются функции `uploadFile()`:

```
var view = $("#drop");
view.dropArea();

view.bind("drop", function(e){
  e.stopPropagation();
  e.preventDefault();

  var files = e.originalEvent.dataTransfer.files;
  for ( var i = 0; i < files.length; i++)
    uploadFile(files[i]);

  return false;
});
```

Передача файла

А теперь перейдем к функции `uploadFile()`, где и происходят все чудеса. Для отправки Ajax-запроса на передачу на сервер мы собираемся воспользоваться функцией `$.upload()` в jquery.upload.js. Мы будем прислушиваться к событиям `progress` запроса и обновлять индикатор выполнения jQuery UI. Как только передача будет завершена, мы уведомим об этом пользователя и удалим элемент:

```
var uploadFile = function(file){
  var element = $("<div />");
  element.text(file.fileName);

  var bar = $("<div />");
  element.append(bar);
  $("#progress").append(element);

  var onProgress = function(e){
    var per = Math.round((e.position / e.total) * 100);
    bar.progressbar({value: per});
  };

  var onSuccess = function(){
    element.text("Передача завершена");
    element.delay(1000).fade();
  };

  $.upload("/uploads", file, {upload: {progress: onProgress},
          success: onSuccess});
};
```

Все предельно просто! Полную версию примера можно найти в assets/ch07/dragdropupload.html.

8 Веб-технологии реального времени

Почему так важны веб-технологии реального времени? Мы живем в мире реального времени, и вполне естественно, что веб-технологии движутся в этом направлении. Пользователи настоятельно требуют связи, данных и поиска в реальном времени. Наши ожидания, связанные со скоростью доставки в наш адрес информации по Интернету, изменились — задержка новостей на несколько минут уже неприемлема. Основные компании, такие как Google, Facebook и Twitter, очень быстро поняли это и стали предлагать в своих службах функциональные возможности работы в реальном времени. Это нарастающая тенденция, склонная только лишь к ускорению.

История работы в режиме реального времени

Традиционно веб-технологии выстраивались вокруг модели запрос-ответ HTTP-протокола: клиент запрашивал веб-страницу, сервер ее доставлял, и больше ничего не происходило, пока клиент не запрашивал другую страницу. Затем появилась технология Ajax, которая придала веб-страницам динамизма — запросы к серверу теперь могли посылаться в фоновом режиме. Но если сервер имел для клиента дополнительные данные, способа оповестить его об этом после загрузки страницы не существовало, возможность принудительного преподнесения клиенту оперативной информации отсутствовала.

Было изобретено немало решений. Самым основным из них был опрос: постоянные обращения к серверу за получением новой информации. Это давало пользователям ощущение работы в реальном времени. На практике такая технология оборачивалась задержками и снижением производительности, поскольку серверам приходилось обрабатывать огромное число подключений в секунду, каждое из которых сопровождалось как TCP-квитированием, так и издержками работы HTTP-протокола. Хотя опрос до сих пор применяется, он весьма далек от идеала.

Затем были изобретены более совершенные транспортные технологии под общим термином Comet. Эти технологии состояли из информационных кадров (iframes), имеющих постоянный характер, протоколов xhr-multipart, htmlfile и длительного опроса (long polling). При использовании длительного опроса клиент открывает

XMLHttpRequest-подключение (XHR) к серверу, которое никогда не закрывается, удерживая клиента на связи. Когда у сервера появляются новые данные, он, соответственно, прерывает подключение, которое затем закрывается. Затем весь процесс повторяется, позволяя серверу выдать новые данные.

Технология Comet не была приведена к стандарту, и поэтому у нее была проблема совместимости с браузерами. Кроме того, существовали и проблемы производительности. Каждое подключение к серверу включало полный набор HTTP-заголовков, и при необходимости малого времени задержки это становилось настоящей проблемой. Но это не сбрасывало Comet со счетов, поскольку данная технология была безальтернативной.

Для внедрения информации со стороны сервера также использовались и дополнительные модули для браузеров, такие как Adobe Flash и Java. Они допускали простые TCP-сокет соединения с серверами, которые могли использоваться для принудительной отправки клиентам оперативной информации. Возражения были связаны с тем, что не было никакой гарантии установки соответствующих дополнительных модулей, и технология часто страдала от проблем совместной работы с брандмауэрами, особенно в корпоративных сетях.

Теперь появилось альтернативное решение в виде части спецификации HTML5. Но должно пройти некоторое время, пока все браузеры, и особенно Internet Explorer, не выйдут на уровень текущих разработок. А до тех пор Comet будет оставаться полезным инструментом в арсенале любого разработчика программ на стороне клиента.

WebSockets

WebSockets является частью спецификации HTML5, предоставляющей двунаправленные полнодуплексные сокеты поверх TCP-протокола. Это означает, что сервера могут принудительно отправлять данные клиенту, не заставляя разработчиков прибегать к длительному опросу или к использованию дополнительных модулей для браузера, что является весьма серьезным усовершенствованием. Но поддержка этой технологии реализована далеко не на всех браузерах, а протокол все еще дорабатывается с целью решения проблем безопасности. И тем не менее это не должно вас отпугивать — скоро все проблемы начального периода будут решены, и спецификация приобретет окончательный вид. Тем временем браузеры, не поддерживающие WebSockets, могут отступить к использованию устаревших методов, например Comet или длительный опрос (polling).

Технология WebSockets обладает значительными преимуществами над предыдущими системами принудительной передачи данных со стороны сервера, поскольку она имеет полнодуплексный характер, не осуществляется поверх HTTP-протокола и, будучи открытой, продолжает свое присутствие. Серьезным недостатком Comet были издержки протокола HTTP, где каждый запрос также имел полный набор HTTP-заголовков. Затем следует упомянуть издержки

многочисленного, никому не нужного TCP-квитирования, имеющего значение на верхних уровнях запросов.

При использовании WebSockets, как только клиент и сервер обменялись квитанциями, сообщения могут ходить в обоих направлениях без издержек на HTTP-заголовки. Это существенно снижает трафик, повышая тем самым производительность. Поскольку мы имеем дело с открытым подключением, сервера получают реальную возможность отправлять клиентам обновления, как только новые данные станут доступны (опрос уже не нужен). Кроме того, соединение является дуплексным, поэтому клиенты также могут отправлять сообщения на сервер и также не иметь при этом издержек, связанных с особенностями HTTP-протокола.

Вот что сказал о WebSockets ведущий разработчик спецификации HTML5 со стороны Google Ян Хиксон (Ian Hickson):

> *Сокращение килобайтов данных до 2 байтов... и сокращение времени ожидания с 150 мс до 50 мс никак нельзя посчитать незначительным. Фактически только этих двух факторов уже вполне достаточно, чтобы технология WebSockets представляла для Google весьма серьезный интерес.*

Итак, посмотрим на поддержку WebSocket в браузерах:

- Chrome >= 4
- Safari >= 5
- iOS >= 4.2
- Firefox >= 4*
- Opera >= 11*

Хотя в Firefox и Opera имеется реализация WebSocket, она в настоящее время отключена из-за недавних опасений, связанных с безопасностью. Но все они на момент выхода этой книги в печать должны быть уже развеяны. А тем временем вы вполне можете обойтись и более старыми технологиями вроде Comet и Adobe Flash. В IE поддержка данной технологии в повестке дня пока не стоит, и вряд ли она будет добавлена в этот браузер до выхода IE9.

Обнаружение поддержки WebSockets осуществляется довольно просто:

```
var supported = ("WebSocket" in window);
if (supported) alert("Технология WebSockets поддерживается");
```

С точки зрения браузера API WebSocket вполне понятен и логичен. Вы создаете экземпляр нового сокета, используя класс `WebSocket`, передавая сокету конечный адрес сервера, в данном случае ws://example.com:

```
var socket = new WebSocket("ws://example.com");
```

Затем к сокету можно добавить ряд слушателей событий:

```
// Подключение состоялось
socket.onopen = function(){ /* ... */ }
```

```
// У подключения имеются новые данные
socket.onmessage = function(data){ /* ... */ }

// Подключение закрыто
socket.onclose = function(){ /* ... */ }
```

когда сервер отправляет данные, инициируется событие `onmessage`. Клиенты, в свою очередь, могут вызвать функцию `send()` для передачи данных обратно на сервер. Несомненно, мы можем вызвать эту функцию только после того, как сокет подключен и состоялось событие onopen:

```
socket.onmessage = function(msg){
  console.log("New data - ", msg);
};

socket.onopen = function(){
  socket.send("Эй, всем кто там — привет").
};
```

При отправке и получении сообщений поддерживаются только строки. Но несложно преобразовать объект в последовательную форму JSON, получив строку сообщения, а затем восстановить объект в прежнем виде, создавая свой собственный протокол:

```
var rpc = {
  test: function(arg1, arg2) { /* ... */ }
};

socket.onmessage = function(data){

  // Анализ JSON
  var msg = JSON.parse(data);

  // Вызов функции RPC
  rpc[msg.method].apply(rpc, msg.args);
};
```

В показанном выше коде мы создали сценарий удаленного вызова процедуры (RPC). Наш сервер может отправлять простой объект в формате JSON, подобный следующему, для вызова функций на стороне клиента:

```
{"method": "test", "args": [1, 2]}
```

Обратите внимание, что мы ограничили вызов объектом `rpc`. Это играет важную роль с точки зрения безопасности, поскольку мы не хотим разрешать выполнение произвольного кода JavaScript.

Для завершения подключения нужно просто вызвать функцию `close()`:

```
var socket = new WebSocket("ws://localhost:8000/server");
```

Вы, наверное, обратили внимание, что при создании экземпляра `WebSocket` использовалась схема WebSocket, `ws://`, а не `http://`. Технология WebSockets также позволяет использовать закодированные подключения с помощью протокола TLS, используя схему `wss://`. По умолчанию в WebSockets будет использоваться порт 80 для незакодированных подключений и порт 443 для закодированных подключений. Эту установку можно отменить, предоставив в URL порт, указанный пользователем. Следует помнить, что для клиентов доступны не все порты, а брандмауэры могут блокировать большинство необычных портов.

На данной стадии вы можете подумать: «Возможно, мне не удастся использовать это в конечном продукте, поскольку стандарт еще не установился, а поддержка в IE отсутствует». Опасения, конечно, не напрасны, но, к счастью, есть решение данной проблемы. Можно воспользоваться Web-socket-js, реализацией WebSocket, осуществленной компанией Adobe Flash. Эту библиотеку можно использовать для предоставления устаревшим браузерам WebSocket-доработки к Flash, дополнительного модуля, доступного практически повсеместно. Он в точности отображает API WebSocket, поэтому, когда вопрос с внедрением WebSockets улучшится, вам нужно будет только лишь удалить библиотеку, ничего не меняя в коде сценария.

Хотя на стороне клиента используется довольно простой API, на серверной стороне ситуация сложнее. Протокол WebSocket прошел через несколько несовместимых шагов: проекты (drafts) 75 и 76. Серверам нужно брать в расчет оба проекта, определяя тип квитирования, используемый клиентом.

WebSockets работает с использованием первоначального «обновляющего» HTTP-запроса к вашему серверу. Если у вашего сервера есть поддержка WebSocket, он выполнит WebSocket-квитирование и соединение будет установлено. В качестве включения в обновляющий запрос используется информация об исходном домене (откуда поступил запрос). Клиенты могут устанавливать WebSocket-подключения к любому домену, т. е. к серверу, который решает, какие клиенты могут подключиться, зачастую используя для этого список допустимых доменов.

Согласно замыслу, технология WebSockets была разработана для нормальной работы с брандмауэрами и прокси-серверами, используя для начального подключения популярные порты и HTTP-заголовки. Но в диком мире Интернета не все и не всегда получается так просто. Некоторые прокси-сервера изменяют заголовки обновления WebSockets, приводя их в негодность. Другие прокси-сервера не допускают долговременных подключений, и через некоторое время их прерывают. Фактически в самом последнем обновлении предварительного протокола (версия 76) произошло непреднамеренное нарушение совместимости с реверсными прокси-серверами и шлюзами. Чтобы дать вашей технологии WebSockets наивысшие шансы на успех, нужно предпринять ряд действий.

- Используйте безопасные WebSocket-подключения (wss). Прокси-серверы не станут заниматься закодированными подключениями, и добавится преимущество защищенности данных от подслушивания.
- Используйте перед своими WebSocket-серверами балансировщик TCP-загруженности, а не HTTP-балансировщик. Рассматривайте возможность при-

менения HTTP-балансировщика только в случае активного рекламирования его поддержки WebSocket.

- Не стоит предполагать, что если браузер имеет поддержку WebSocket, он будет работать. Вместо этого прерывайте подключения по истечении контрольного времени, если они не устанавливаются быстро, и элегантно переходите на другие, более примитивные средства транспортировки данных, такие как Comet или опрос.

Итак, какими вариантами настроек сервера мы располагаем? К счастью, имеется множество реализаций на таких языках, как Ruby, Python и Java. Убедитесь в том, что любая из этих реализаций поддерживает, по крайней мере, проект 76 протокола, поскольку он наиболее распространен на клиентских машинах.

- Node.js:
  - node-Websocket-server;
  - Socket.IO.
- Ruby:
  - EventMachine;
  - Cramp;
  - Sunshowers.
- Python:
  - Twisted;
  - Apache module.
- PHP:
  - php-Websocket.
- Java:
  - Jetty.
- Google Go:
  - Native.

Node.js и Socket.IO

Node.js является самым новым средством, доказавшим свою состоятельность, но при этом и самым впечатляющим. Node.js представляет собой исходный JavaScript-сервер, построенный на основе Google-движка V8 JS. По сути, он невероятно расторопен и хорошо подходит для служб, имеющих большое количество подключенных клиентов, например для WebSocket-сервера.

Socket.IO является библиотекой Node.js для WebSockets. Но, что интересно, она выходит далеко за рамки этой задачи. Вот что говорится в рекламе на ее сайте:

> Socket.IO предназначена для создания приложений реального времени в каждом браузере и мобильном устройстве, стирая границы между разными транспортными механизмами.

Если WebSockets поддерживается, Socket.IO будет стараться использовать эту технологию, но при необходимости будет возвращаться к использованию других транспортных механизмов. Сама библиотека предлагает совместимость со многими браузерами, и список поддерживаемых транспортных механизмов весьма обширен:

- WebSocket;
- Adobe Flash Socket;
- ActiveX HTMLFile (IE);
- XHR с multipart-кодированием;
- XHR с длительным опросом;
- JSONP-опрос (для междоменного обмена данными).

Библиотека Socket.IO обладает превосходной поддержкой браузеров. С реализацией выходных данных сервера могут быть общеизвестные проблемы, но команда разработчиков Socket.IO прошла за вас этот трудный путь, гарантируя совместимость с большинством браузеров. В настоящее время библиотека работает со следующими браузерами:

- Safari >= 4
- Chrome >= 5
- IE >= 6
- iOS
- Firefox >= 3
- Opera >= 10.61

Хотя серверная сторона Socket.IO была сначала написана для Node.js, сейчас она имеет реализации и на других языках, таких как Ruby (Rack), Python (Tornado), Java и Google Go.

Беглый взгляд на API продемонстрирует вам его простоту и открытость. Клиентская часть API очень похожа на API, имеющийся в WebSocket:

```
var socket = new io.Socket();

socket.on("connect", function(){
  socket.send('Привет!');
});

socket.on("message", function(data){
  alert(data);
});

socket.on("disconnect", function(){});
```

Закулисно Socket.IO вычислит самый подходящий транспортный механизм. Как написано в файле readme этой библиотеки, Socket.IO «отличается тем, что предназначена для создания приложений реального времени, работающих в любой среде».

Если вы ищете более высокоуровневое средство, чем Socket.IO, тогда вас может заинтересовать Juggernaut, являющаяся надстройкой над этой библиотекой. Juggernaut имеет канальный интерфейс: клиенты могут подписаться на каналы, а сервер может публиковать информацию в этих каналах, т. е. используется схема PubSub (подписки-публикации). Библиотека может управлять масштабированием, публиковать информацию для конкретных клиентов, использовать протокол защиты транспортного уровня TLS и делать многое другое.

Если вам нужны решения с использованием хостов, можете остановиться на библиотеке Pusher. Она позволит вам оставить позади все трудности управления вашим собственным сервером, чтобы сконцентрироваться на самом интересном: на разработке веб-приложения. Для клиентов все осуществляется так же просто, как включение JavaScript-файла в страницу и подписка на канал. Когда дело касается публикации сообщений, все сводится к отправке HTTP-запроса к имеющемуся REST API.

Архитектура реального времени

Теоретически с возможностью отправки данных клиентам дело обстоит весьма неплохо, но как все это интегрируется с JavaScript-приложением? Если ваше приложение смоделировано правильно, то все предельно просто. Мы собираемся пройти все этапы, необходимые для создания приложения реального времени, в точности следуя схеме PubSub. Первое, что нужно понять, — как процесс обновления добирается до клиентов.

Архитектура реального времени управляется событиями. Обычно события случаются в результате взаимодействия с пользователем: пользователь изменяет запись, а события распространяются по всей системе, пока данные не будут выданы подключенным клиентам, обновляя имеющуюся у них информацию. Замышляя перевод приложения в режим работы реального времени, нужно продумать следующее:

- какие модели должны работать в режиме реального времени?
- каких пользователей нужно уведомить об изменении экземпляров таких моделей?

Возможно, при изменении модели вам понадобится отправить уведомления всем подключенным клиентам. Это будет как раз тот случай действия по предоставлению главной странице информации в реальном масштабе времени, например когда каждый клиент видел одну и ту же информацию. Но более типичным будет случай наличия ресурса, связанного с конкретной группой пользователей. Этих пользователей нужно оповестить об изменении ресурса.

Рассмотрим пример сценария дискуссионной группы.

1. Пользователь публикует в группе новое сообщение.
2. На сервер отправляется Ajax-запрос, и создается запись `Chat`.

3. В отношении модели `Chat` вызываются сохраняющие функции обратного вызова, инициируя выполнение нашего метода по мере обновления информации у клиентов.
4. Мы ищем всех пользователей, связанных с группой дискуссионных записей, т. е. тех, кого нужно оповестить.
5. Обновление, конкретизирующее случившееся (создание записи в `Chat`), отправляется соответствующим пользователям.

Подробности процесса специфичны для выбранной вами машины базы данных. Но если вы используете Rails, то неплохим примером будет Holla. Когда создаются записи `Message`, `JuggernautObserver` обновляет информацию для соответствующих клиентов.

И тут возникает следующий вопрос: как мы можем отправить уведомления конкретным пользователям? Отличным способом выполнения данной задачи будет использование схемы PubSub: клиенты подписываются на конкретные каналы, а сервера публикуют информацию в этих каналах. Пользователь просто подписывается на уникальный канал, содержащий идентификатор — возможно, что это ID пользовательской базы данных, — затем сервер просто должен поместить публикацию в этом уникальном канале, чтобы отправить оповещения конкретному пользователю.

Например, конкретный пользователь может подписаться на следующий канал:

```
/observer/0765F0ED-96E6-476D-B82D-8EBDA33F4EC4
```

Где случайный набор цифр является уникальным идентификатором для текущего, вошедшего пользователя. Для отправки уведомления этому конкретному пользователю серверу просто нужно опубликовать данные в этом же канале.

У вас может появиться вопрос, как схема PubSub работает с такими транспортными механизмами, как WebSockets и Comet. К счастью, существует множество готовых решений, таких как уже упоминавшиеся Juggernaut и Pusher. PubSub является общей абстракцией поверх WebSockets, и его API должны быть очень похожи на любую службу или библиотеку, которую вы в конечном счете выбираете.

Как только уведомление будет отправлено клиентам, вы увидите настоящую красоту MVC-архитектуры. Вернемся к нашему примеру дискуссионной группы. Отправленное нами в адрес клиентов уведомление может иметь следующий вид:

```
{
  "klass":  "Chat",
  "type":   "create",
  "id":     "3",
  "record": {"body": "New chat"}
}
```

Оно содержит модель, подвергшуюся изменению, тип изменения и любые соответствующие свойства. Используя это уведомление, наш клиент может локально создать новую запись `Chat`. Поскольку клиентские модели привязаны к UI,

интерфейс автоматически будет обновлен для отображения нового сообщения дискуссионной группы.

Самое замечательное, что ничто из вышеперечисленного не имеет отношения к модели `Chat`. Если нам нужно создать другую модель реального времени, для этого понадобится всего лишь еще один наблюдатель на стороне сервера, чтобы обеспечить обновления клиентов при ее изменении. Теперь наша машина базы данных и модели на стороне клиента связаны вместе. Любые изменения, вносимые в модели на машине базы данных, автоматически распространятся на все соответствующие машины клиентов, обновляя их пользовательские интерфейсы. При использовании данной архитектуры приложение обретает свойства приложения реального времени. Любые действия, предпринятые пользователем, моментально транслируются другим пользователям.

Ощущаемая скорость

Скорость является очень важной составляющей пользовательского интерфейса, поскольку она вносит существенную разницу в пользовательское восприятие (user experience, UX) и может оказать непосредственное влияние на доход, получаемый от приложения. Нижеперечисленные компании постоянно изучают ее влияние:

Amazon

100 мс дополнительного времени загрузки приводят к падению продаж в 1% (источник: Грэг Линден (Greg Linden), Amazon).

Google

500 мс дополнительного времени загрузки приводят к 20% уменьшению количества поисков (источник: Маррисса Майер (Marrissa Mayer), Google).

Yahoo!

400 мс дополнительного времени загрузки приводят к 5–9% увеличению количества людей, щелкающих на кнопке возврата к предыдущей странице еще до того, как страница будет загружена (источник: Николь Салливан (Nicole Sullivan), Yahoo!).

Ощущаемая скорость не менее важна, чем действительная скорость, потому что именно ее склонны замечать пользователи. Поэтому ключевой момент заключается в том, чтобы пользователи думали, что приложение работает быстро, даже если реально это не так. Возможность сделать именно так является одним из преимуществ приложений JavaScript — пользовательский интерфейс не блокируется, даже если выполняемый в фоновом режиме запрос занимает много времени.

Давайте снова вернемся к сценарию дискуссионной группы. Пользователь отправляет новое сообщение, инициируя Ajax-запрос к серверу. Мы можем ждать до тех пор, пока сообщение не совершит путешествие к серверу и обратно к клиентам, и только потом добавить его к журналу дискуссии. Но это может быть связано с задержкой в пару секунд между временем отправки пользователем нового

сообщения и его появлением в дискуссионном журнале. Приложение покажется медленно работающим, что, несомненно, испортит пользовательское восприятие.

Почему бы вместо этого не создать новое сообщение локально, тем самым немедленно добавив его к дискуссионному журналу? С точки зрения пользователя, создается впечатление немедленной отправки сообщения. Пользователи не хотят знать (или брать в расчет), что сообщение еще не было доставлено другим клиентам дискуссионной группы. Они просто будут рады ощущению быстроты и моментальной реакции.

Кроме взаимодействий с пользователем, одной из самых медленных частей веб-приложений является загрузка новых данных. Важно осуществлять разумную предварительную загрузку, чтобы попытаться предсказать, что понадобится пользователю до того, как он действительно запросит эту информацию. Затем следует кэшировать данные в памяти — если пользователю они впоследствии понадобятся, вам не придется снова запрашивать их у сервера. С самого начала своей работы приложение должно осуществлять предварительную загрузку часто востребуемых данных. Пользователи скорее простят более медленную начальную загрузку, чем текущую.

При взаимодействии пользователей с вашим приложением им всегда нужно давать обратную реакцию на их действия — обычно это некий визуальный индикатор. На деловом жаргоне это называется *управлением ожиданиями*, обеспечением информирования клиентов о состоянии проекта и о ETA (расчетном времени завершения). То же самое относится к пользовательскому восприятию (UX) — пользователи будут более терпимы, если им дать индикатор происходящего. Пока пользователи ожидают новые данные, покажите им сообщение или какой-нибудь вращающийся индикатор. Если файл передается на сервер, покажите индикатор выполнения и предполагаемую продолжительность передачи. Все это дает ощущение скорости, улучшая пользовательское восприятие.

9 Тестирование и отладка

Все тесты в той или иной степени проводятся разработчиками в процессе программирования. Формой тестирования является даже простой запуск кода вручную. Но здесь мы собираемся рассмотреть автоматизированное тестирование в JavaScript, т. е. написание определенных утверждений, запускаемых автоматически. Автоматизированное тестирование не приведет к избавлению от ошибок в вашем коде, но станет показателем для эффективного сокращения количества дефектов и для предотвращения попадания в основу кода прежних ошибок. Существует множество великолепных ресурсов, обосновывающих и объясняющих различные типы тестирования. Поэтому, чтобы не заниматься пересказом, в данной главе внимание будет сконцентрировано на особенностях тестирования в JavaScript, в отличие от тестирования в других языках.

Тестирование в JavaScript по-настоящему еще не вошло в культуру программирования, поэтому многие JavaScript-разработчики не пишут для своего кода никаких тестов. Я полагаю, главная причина в том, что автоматизированное тестирование JavaScript — не такое простое дело, и определить его масштабы невозможно. Возьмем, к примеру, jQuery. В этой библиотеке имеются сотни блочных тестов и около 10 различных тестовых комплектов для имитации различных сред окружения, в которых ожидается работа. Каждый имеющийся в комплекте тест должен быть запущен один раз. Теперь посмотрим на браузеры, поддерживаемые jQuery:

- Safari: 3.2, 4, 5 и самые последние выпуски
- Chrome: 8, 9, 10, 11
- Internet Explorer: 6, 7, 8, 9
- Firefox: 2, 3, 3.5, 3.6 и самые последние выпуски
- Opera: 9.6, 10, 11

Итак, мы имеем дело с примерно двадцатью версиями пяти браузеров, и каждый комплект должен быть запущен в каждой версии браузера. Вы можете увидеть, как количество тестов, которые нужно запускать, растет в геометрической прогрессии, и я еще даже не добрался до платформ! Масштабы этой работы не поддаются оценке.

Очевидно, что jQuery — особый случай, пример, разработанный для высвечивания существующей проблемы. Вполне возможно, что вам не понадобится поддержка

и половины тех браузеров, в которых работает jQuery, и вам вряд ли придется иметь так много комплектов. Но вам все же придется выбрать те браузеры, которые будут поддерживаться вашим приложением, а затем выбрать тесты для этих браузеров.

Перед тем как продолжить тему, стоит обратить внимание на браузерное окружение, поскольку именно оно в конечном счете диктует ограничения, навязываемые веб-разработчикам. Это окружение изменяется так быстро, что этот анализ на момент чтения данных строк, скорее всего, устареет. Но основные тенденции должны остаться прежними.

Показатели использования браузеров зависят от того, как их измерять. Они существенно отличаются для разных стран и континентов и обычно зависят от уровня интернет-грамотности населения. Вот, к примеру, результаты, представленные на сайте Statcounter.com для Европы начала 2011 года:

- Safari: 4%
- Chrome: 15%
- IE: 36%
- Firefox: 37%
- Opera: 2%

Основная тенденция заключается в падении популярности IE и увеличении использования Firefox и Chrome. Устаревшие браузеры, наподобие IE6, стали достоянием истории, занимая доли процента. Если разработка ведется не для корпоративной или правительственной клиентуры и предназначена в том числе и для консервативных пользователей, следует побеспокоиться и о поддержке этих древних браузеров.

Как говорится, «ложь бывает трех видов: обычная ложь, гнусная ложь и статистика». Это вдвойне верно для статистики использования браузеров. К примеру, судя по статистике моего блога, IE используется примерно в 5% случаев, что значительно ниже национального среднего показателя. Иными словами, наблюдаемый трафик находится под огромным влиянием конкретной аудитории. Если ваш сайт ориентирован на технически подкованных, легких на подъем людей, будет получен высокий процент пользователей Firefox и Chrome, а вот сайты более общей направленности будут посещаться теми, кто лучше вписывается в картину средних национальных показателей. При выборе браузеров, которые необходимо поддерживать, нужно учесть специфику вашей аудитории, а не общую картину проникновения браузеров. Но я, как правило, тестирую следующие браузеры:

- IE 8, 9
- Firefox 3.6
- Safari 5
- Chrome 11

При отсутствии статистики для ваших реально существующих служб, показывающей, какими браузерами пользуется ваша аудитория, придется воспользоваться догадками, основанными на знании реального положения дел у вашей целевой аудитории. После определения круга поддерживаемых браузеров следующим шагом должно стать написание автоматизированных тестов и обеспечение их прохождения на каждом поддерживаемом браузере.

Блочное тестирование

Ручное тестирование похоже на комплексное тестирование, придающее уверенность в работоспособности приложения на высоком уровне. Блочное тестирование относится к более низкому уровню, гарантирующему ожидаемое выполнение конкретных фрагментов кода в фоновом режиме. Блочное тестирование больше похоже на выявление кроссбраузерных проблем, но оно позволяет решить их довольно быстро, поскольку испытанию подвергаются сравнительно небольшие фрагменты кода.

Другим преимуществом блочных тестов является то, что они прокладывают путь к автоматизации. Более подробно этот вопрос будет рассмотрен в данной главе чуть позже, но блочные тесты позволяют установить постоянный интеграционный сервер, запускающий тесты приложения при каждой передаче кода. Тестирование осуществляется намного быстрее, чем прогон всего приложения вручную, и гарантирует, что изменения, внесенные в один кодовый фрагмент, не будут иметь каких-либо побочных эффектов в каком-нибудь другом месте приложения.

Для осуществления блочного тестирования существует множество библиотек JavaScript, у каждой из которых есть свои сильные и слабые стороны. Мы собираемся рассмотреть наиболее популярные библиотеки такого рода, но основные принципы должны применяться к любой выбранной вами библиотеке.

Утверждения

Утверждения являются основой тестирования, они определяют, какие из тестов пройдены или провалены. Утверждения являются формулировками, показывающими ожидаемый результат выполнения вашего кода. Если утверждение дает ложный результат, тест проваливается и вы знаете, что что-то пошло не так.

Например, здесь показана простая функция `assert()`, которая будет использоваться для примеров в данной книге:

```
var asset = function(value, msg) {
  if ( !value )
    throw(msg || (value + " не является истиной"));
};
```

Этой функции передается значение и необязательное сообщение. Если значение не вычисляется в истину (`true`), утверждение считается несостоятельным:

```
// Несостоятельные утверждения
assert( false );
assert( "" );
assert( 0 );
```

В процессе булевой проверки JavaScript автоматически осуществляет преобразование типов `undefined`, `0` и `null` в `false`. Иными словами, `assert` работает и для `null`-проверки:

```
// Если инструкция дает результат null, утверждение считается ложным
assert( User.first() );
```

Преобразование типов будет оказывать существенное влияние на ваше тестирование, поэтому стоит проверить некоторые странные и удивительные манеры поведения JavaScript, демонстрируемые этим языком при *конвертации типов*.

Библиотеки утверждений не ограничиваются только проверкой истинности. Многие библиотеки включают полные массивы обнаружителей совпадений, от сравнения примитивных объектов и до проверки, какое из двух чисел больше. По крайней мере, все они включают функцию `assertEqual()`, позволяющую сравнивать два значения:

```
var assertEqual = function(val1, val2, msg) {
  if (val1 !== val2)
    throw(msg || (val1 + " не равно " + val2));
};

// Проходное утверждение
assertEqual("one", "one");
```

Все библиотеки тестирования, которые мы собираемся рассматривать, содержат набор утверждений наряду с немного отличающимися друг от друга API для их определения.

QUnit

QUnit является одной из наиболее популярных и постоянно поддерживаемых библиотек, первоначально разработанной для тестирования jQuery. Итак, как же установить среду тестирования QUnit? Сначала нужно провести локальную установку *файлов проекта*, а затем создать статическую страницу запуска теста:

```
<!DOCTYPE html>
<html>
<head>
<title>QUnit Test Suite</title>
```

```
<link rel="stylesheet" href="qunit/qunit.css" type="text/css"
media="screen">
<script type="text/javascript" src="qunit/qunit.js"></script>
<!-- include tests here... -->
</head>
<body>
<h1 id="qunit-header">QUnit Test Suite</h1>
<h2 id="qunit-banner"></h2>
<div id="qunit-testrunner-toolbar"></div>
<h2 id="qunit-userAgent"></h2>
<ol id="qunit-tests"></ol>
<div id="qunit-fixture">test markup</div>
</body>
</html>
```

Для создания утверждений их нужно поместить в контрольный пример. Давайте, к примеру, создадим тестовый комплект для того объектно-реляционного отображения (ORM), который был создан в главе 3:

```
test("load()", function(){
  var Asset = Model.setup();

  var a = Asset.init();
  a.load({
    local: true,
    name: "test.pdf"
  });

ok(a.local, "Load sets properties");
  equals(a.name, "test.pdf", "load() sets properties (2)");
  var b = Asset.init({
    name: "test2.pdf"
  });
  equals(b.name, "test2.pdf", "Calls load() on instantiation");
});
```

Новая тестируемая ситуация определяется путем вызова функции `test()` и передачи ей имени теста и его функции обратного вызова (где и происходит волшебство). Внутри функции обратного вызова мы имеем различные утверждения: функция `ok()` утверждает, что ее первый аргумент вычисляется в `true`, а функция `equals()` сравнивает свои два аргумента. Всем утверждениям передается последний строковый аргумент, являющийся именем утверждения, который упрощает для нас определение, какое утверждение было пройдено, а какое нет.

Давайте добавим этот тест к странице и обновим ее, получив изображение, показанное на рис. 9.1.

Довольно мощный результат! С первого взгляда можно увидеть, какие тесты были пройдены, а какие нет, для этого достаточно только лишь обновить страницу.

Теперь можно приступить к тестированию каждого браузера, поддерживаемого нашим приложением, убеждаясь в прохождении блочного теста на каждом из них.

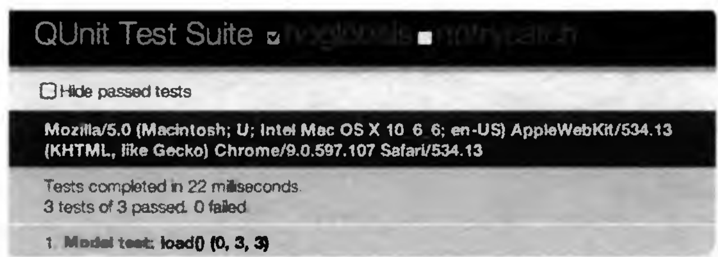

Рис. 9.1. Результаты тестирования с помощью QUnit

Тесты обособлены с помощью функции `module()`, которой передается имя и дополнительные аргументы. Давайте доработаем первый пример, передав дополнительный аргумент `setup` функции `module()`, содержащий функцию обратного вызова, которая будет выполнена для каждого теста, запускаемого в модуле. В данном случае всем нашим тестам понадобится модель `Asset`, поэтому мы создадим ее в `setup`:

```
module("Model test", {
  setup: function(){
    this.Asset = Model.setup();
  }
});
test("load()", function(){
  var a = this.Asset.init();
  a.load({
    local: true,
    name: "test.pdf"
  });
  ok(a.local, "Load sets properties");
  equals(a.name, "test.pdf", "load() sets properties (2)");
  var b = this.Asset.init({
    name: "test2.pdf"
  });
  equals(b.name, "test2.pdf", "Calls load() on instantiation");
});
```

Теперь тест стал немного понятнее, и его можно будет использовать при добавлении новых тестов. Функции `module()` также передается дополнительный аргумент `teardown` (демонтаж), являющийся функцией обратного вызова, которая будет выполнена после каждого теста в модуле. Давайте добавим к нашему комплекту еще один тест:

```
test("attributes()", function(){
```

```
  this.Asset.attributes = ["name"];
  var a = this.Asset.init();
  a.name = "test.pdf";
  a.id   = 1;
  equals(a.attributes(), {
    name: "test.pdf";
    id: 1
  });
});
```

Если его опробовать, вы увидите, что он не будет пройден, как на странице, показанной на рис. 9.2. Это потому что в функции `equals()` используется оператор сравнения `==`, который будет давать отрицательный результат для объектов и массивов. Вместо этого нам нужно воспользоваться функцией `same()`, осуществляющей глубокое сравнение, и наш комплект тестов будет снова пройден:

```
test("attributes()", function(){
  this.Asset.attributes = ["name"];
  var a = this.Asset.init();
  a.name = "test.pdf";
  a.id   = 1;
  same(a.attributes(), {
    name: "test.pdf",
    id: 1
  });
});
```

QUnit включает пару других типов утверждений: `notEqual()` и `raises()`. Полную версию примеров их применения можно найти в **assets/ch09/qunit/model.test.js** или в документации по QUnit.

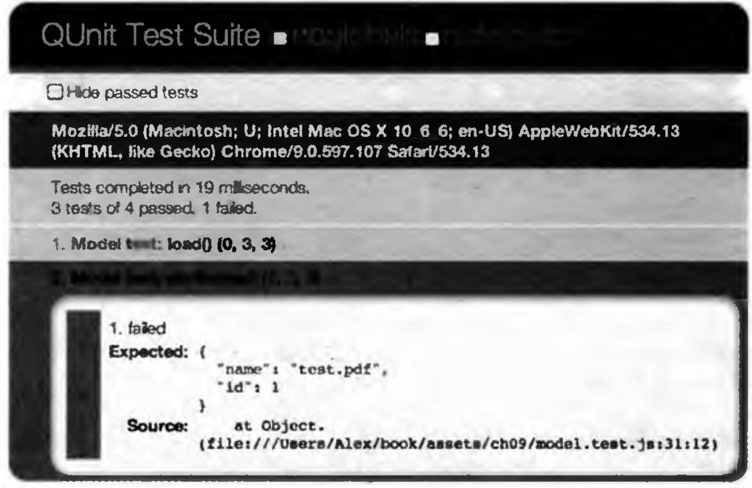

Рис. 9.2. Проваленные тесты в QUnit

Jasmine

Еще одной популярной библиотекой тестирования (и моим личным предпочтением) является Jasmine. Вместо блочного тестирования у Jasmine есть спецификации, описывающие поведение определенных объектов внутри вашего приложения. Практически они подобны блочным тестам, но только с другой терминологией.

Преимущество Jasmine в том, что она не требует никаких других библиотек, и даже не требует DOM. Значит, ее можно запускать в любой среде окружения JavaScript, например на серверной стороне вместе с Node.js.

Как и при использовании QUnit, нам нужно установить статическую HTML-страницу, которая будет загружать все спецификации, запускать их и отображать результат:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Jasmine Test Runner</title>
  <link rel="stylesheet" type="text/css" href="lib/jasmine.css">
  <script type="text/javascript" src="lib/jasmine.js"></script>
  <script type="text/javascript" src="lib/jasmine-html.js"></script>
  <!-- сюда нужно включить исходные файлы... -->
  <!-- сюда нужно включить исходные файлы спецификаций... -->
</head>
<body>
  <script type="text/javascript">
    jasmine.getEnv().addReporter(new jasmine.TrivialReporter());
    jasmine.getEnv().execute();
  </script>
</body>
</html>
```

Давайте посмотрим, как пишутся спецификации Jasmine. Протестируем кое-что еще из библиотеки ORM из главы 3:

```
describe("Model", function(){
  var Asset;
  beforeEach(function(){
    Asset = Model.setup();
    Asset.attributes = ["name"];
  });
  it("can create records", function(){
    var asset = Asset.create({name: "test.pdf"});
    expect(Asset.first()).toEqual(asset);
  });
  it("can update records", function(){
    var asset = Asset.create({name: "test.pdf"});
    expect(Asset.first().name).toEqual("test.pdf");
    asset.name = "wem.pdf";
```

```
    asset.save();
    expect(Asset.first().name).toEqual("wem.pdf");
  });
  it("can destroy records", function(){
    var asset = Asset.create({name: "test.pdf"});
    expect(Asset.first()).toEqual(asset);
    asset.destroy();
    expect(Asset.first()).toBeFalsy();
  });
});
```

Спецификации сгруппированы в наборы с помощью функции `describe()`, которой передается имя спецификации и безымянная функция. В показанном выше примере в качестве установочной утилиты, запускаемой перед каждой функцией, используется функция `beforeEach()`. Jasmine также включает функцию демонтажа `afterEach()`, которая вызывается после запуска каждой спецификации. Внутри функции `beforeEach()` мы определяем переменную `Asset`, поэтому она является локальной по отношению к набору, и к ней можно обращаться внутри каждой спецификации.

Спецификация начинается с функции `it()`, которой передаются имя спецификации и безымянная функция, содержащая утверждения. Утверждения создаются путем передачи соответствующих значений функции `expect()`, с последующим вызовом обнаружителя совпадения, например, одного из следующих:

`expect(x).toEqual(y)`

Сравнивает объекты или примитивы `x` и `y` и обеспечивает прохождение теста при их равенстве.

`expect(x).toBe(y)`

Сравнивает объекты или примитивы `x` и `y`, и обеспечивает прохождение теста, если они являются одним и тем же объектом.

`expect(x).toMatch(pattern)`

Сравнивает `x` со строкой или с шаблоном регулярного выражения и обеспечивает прохождение теста, если они соответствуют друг другу.

`expect(x).toBeNull()`

Обеспечивает прохождение теста, если `x` имеет значение `null`.

`expect(x).toBeTruthy()`

Обеспечивает прохождение теста, если `x` вычисляется в `true`.

`expect(x).toBeFalsy()`

Обеспечивает прохождение теста, если `x` вычисляется в `false`.

`expect(x).toContain(y)`

Обеспечивает прохождение теста, если массив или строка `x` содержит `y`.

`expect(fn).toThrow(e)`

Обеспечивает прохождение теста, если функция `fn` выдает при выполнении исключение `e`.

Jasmine включает в себя множество других обнаружителей совпадений и даже позволяет вам создавать свои собственные обнаружители. На рис. 9.3 показан обзор работающего средства Jasmine Test Runner, использующего показанные выше спецификации.

Jasmine 1.0.2 revision 1298837858 Show passed skipped
3 specs, 0 failures in 0.021s Finished at Fri Mar 11 2011 07:51:58 GMT+1300 (NZDT) run all
Model run
can create records run
can update records run
can destroy records run

Рис. 9.3. Результаты теста, проведенного с помощью Jasmine

Драйверы

Благодаря использованию библиотеки тестирования у нас появилась какая-то степень автоматизации, но еще не решена проблема запуска ваших тестов в среде множества других браузеров. Нельзя признать идеальной с точки зрения производительности ситуацию, когда перед передачей кода в эксплуатацию разработчикам приходится заново запускать все тесты на пяти различных браузерах.

Для решения именно этой проблемы и были разработаны драйверы. Они представляют собой программы, выполняемые в фоновом режиме и интегрированные с различными браузерами. Эти программы запускают ваши JavaScript тесты в автоматическом режиме и уведомляют вас о провалах.

Установка драйверов на машине каждого разработчика может стать слишком затратным делом, поэтому у большинства компаний имеется один постоянный интеграционный сервер, который будет использовать перехват переданного кода для автоматического запуска всех JavaScript-тестов, чтобы можно было убедиться в их успешном прохождении.

Watir (произносится как «уотэ») — это написанная на языке Ruby библиотека драйверов, интегрированная с Chrome, Firefox, Safari и Internet Explorer (в зависимости от используемой платформы). После установки вы можете дать Watir ряд Ruby-команд по управлению браузером, щелчкам на ссылках и заполнению форм, как будто это делается человеком. Вы можете запускать несколько контрольных примеров и утверждений, настроенных на ожидаемую работу:

```
# FireWatir управляет Firefox
require "firewatir"
browser = Watir::Browser.new
browser.goto("http://bit.ly/watir-example")
browser.text_field(:name => "entry.0.single").set "Watir"
browser.button(:name => "logon").click
```

Из-за ограничений возможности установки того или иного браузера на той или иной операционной системе, если тестирование происходит с использованием Internet Explorer, ваш постоянный интеграционный сервер должен будет запустить версию Windows. Аналогично для тестирования в Safari вам также понадобится сервер, запущенный под управлением Mac OS X.

Еще одним весьма популярным инструментом управления браузерами является Selenium. Эта библиотека предоставляет доменный язык сценариев (domain scripting language, DSL) для написания тестов на нескольких языках программирования, таких как C#, Java, Groovy, Perl, PHP, Python и Ruby. Selenium может запускаться в локальном режиме. Обычно эта библиотека запускается в фоновом режиме на постоянном интеграционном сервере, не обращая на себя внимания при передаче нового кода, но уведомляя вас о провале тестов. Сила Selenium состоит в поддержке нескольких языков, а также в Selenium IDE, дополнительном модуле Firefox, который записывает и воспроизводит действия внутри браузера, существенно упрощая разработку тестов.

На рис. 9.4 показано использование средства Selenium IDE для записи щелчка на ссылке, заполнения формы и ее отправки после заполнения. После записи сеанса его можно воспроизвести, используя зеленую кнопку. Это средство будет имитировать записанные действия, осуществляя навигацию и заполняя тестовую форму.

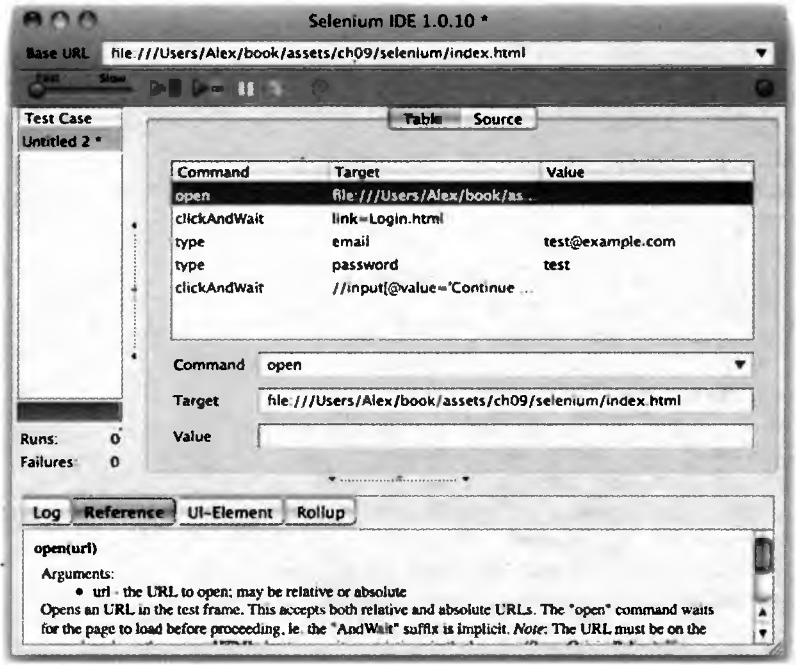

Рис. 9.4. Запись инструкций с помощью Selenium

Затем, как показано на рис. 9.5, мы можем экспортировать записанный контрольный пример (Test Case) во множество разных форматов.

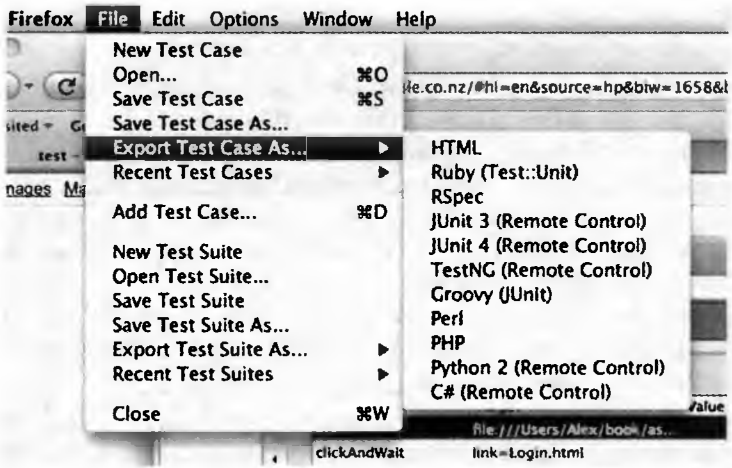

Рис. 9.5. Экспорт контрольных примеров Selenium в различные форматы

Например, вот как выглядит экспортированный контрольный пример в виде `Test::Unit` на языке Ruby. Как видите, среда Selenium IDE последовательно сгенерировала все соответствующие методы управления, существенно сократив объем работы, необходимый для тестирования страницы:

```
class SeleniumTest < Test::Unit::TestCase
  def setup
    @selenium = Selenium::Client::Driver.new \
      :host => "localhost",
      :port => 4444,
      :browser => "*chrome",
      :url => "http://example.com/index.html",
      :timeout_in_second => 60
    @selenium.start_new_browser_session
  end
  def test_selenium
    @selenium.open "http://example.com/index.html"
    @selenium.click "link=Login.html"
    @selenium.wait_for_page_to_load "30000"
    @selenium.type "email", "test@example.com"
    @selenium.type "password", "test"
    @selenium.click "//input[@value='Continue →']"
    @selenium.wait_for_page_to_load "30000"
  end
end
```

Теперь можно создать утверждение относительно объекта `@selenium`, например о содержании в нем определенного текстового фрагмента:

```
def test_selenium
  # ...
  assert @selenium.is_text_present("elvis")
end
```

Дополнительную информацию о Selenium можно найти на веб-сайте selemiumhq.org, просмотрев введение screencast.

Автономное тестирование

При разработке таких серверных JavaScript-реализаций, как Node.js и Rhino, появляется возможность запуска ваших тестов за пределами браузера в автономном окружении с использованием командной строки. Такие тесты выгодно отличаются скоростью выполнения и простотой настройки, поскольку здесь уже нет никакого многообразия браузеров и постоянной интегрированной среды. Очевидным недостатком является запуск тестов вне реальной рабочей среды.

Возможно, это не такая уж большая проблема, как кажется на первый взгляд. Разобравшись, вы поймете, что основной объем созданного кода JavaScript относится к логике приложения и не зависит от используемых браузеров. Кроме того, такие библиотеки, как jQuery, сгладят несовместимость браузеров, когда дело коснется управления DOM и событиями. Для приложений меньшего масштаба вам вполне должно хватить подготовительной среды развертывания и ряда высокоуровневых кроссбраузерных интеграционных тестов (выполняемых в ручном или автоматическом режиме).

Библиотека Envjs изначально была разработана Джоном Рейсигом (John Resig), создателем JavaScript-среды jQuery. Она предлагает реализацию браузерных и DOM API поверх Rhino, реализации JavaScript на языке Java, используемой в движке Mozilla. Библиотеку env.js можно использовать вместе с Rhino для запуска тестов в командной строке.

Zombie

Zombie.js является автономной библиотекой, разработанной с целью получения преимуществ от невероятной производительности и асинхронной природы Node.js. Скорость является ключевым фактором: чем меньше время ожидания завершения тестов, тем больше времени вы получаете для создания новых свойств и устранения недостатков.

Приложения, использующие большой объем кода JavaScript на стороне клиента, тратят основную часть своего времени на загрузку, анализ и вычисление этого JavaScript. Разработанный Google JavaScript-движок V8 помогает вашим тестам выполняться быстрее.

Хотя комплект ваших тестов и код JavaScript на стороне клиента запускаются на одном и том же движке, Zombie использует другое свойство V8 — контексты, которые их разделяют, не позволяя смешиваться одним и тем же глобальным переменным и состояниям. Это похоже на способ, с помощью которого Chrome открывает каждую вкладку в ее собственном процессе.

Еще одним преимуществом контекстов является возможность запуска нескольких тестов в параллельном режиме — каждый использует свой собственный объект `Browser`. Один тест может проверять содержимое DOM, в то время как другой тест ждет ответа после запроса страницы, сокращая время завершения всего комплекта тестов. Вам понадобится использовать асинхронную среду тестирования, например такую великолепную, как Vows.js, и обратить внимание на то, что некоторые тесты должны быть запущены в параллельном режиме, а другие должны быть запущены последовательно.

Zombie.js предоставляет объект `Browser`, который работает во многом как реальный веб-браузер: он сохраняет состояние между страницами (cookie-файлы, история, веб-хранилище) и предоставляет доступ к текущему окну (и через него к загруженному документу). Дополнительно библиотека снабжает вас методами для работы с текущим окном, действуя подобно пользователю (посещая страницы, заполняя формы, щелкая на ссылках и т. д.) и контролируя содержимое окна (используя XPath или CSS селекторы).

Пример заполнения имени пользователя и пароля, отправки формы, а затем тестирования содержимого элемента заголовка:

```
// Заполнение полей email, password и отправка формы.
browser.
  fill("email", "zombie@underworld.dead").
  fill("password", "eat-the-living").
  pressButton("Sign Me Up!", function(err, browser) {
   // Форма отправлена, новая страница загружена.
   assert.equal(browser.text("title"), "Welcome to Brains Depot");
  });
```

Этот пример является неполным. Вполне очевидно, что перед взаимодействием с ним нужно затребовать библиотеку Zombie.js, создать новый объект `Browser` и загрузить страницу. Кроме этого нужно позаботиться об аргументе `err`.

Подобно веб-браузеру, Zombie.js имеет асинхронную природу: ваш код не блокируется в ожидании загрузки страницы, инициализации события или срабатывания таймера. В противовес этому вы можете зарегистрировать слушателей таких событий, как `loaded` и `error`, или передать функцию обратного вызова.

По соглашению, когда Zombie передается функция обратного вызова, она будет использовать ее одним из двух способов. Если действие было успешным, она передаст `null` и некоторое другое значение, чаще всего это будет ссылка на объект `Browser`. Если действие закончилось неудачей, она передаст ссылку на объект `Error`. Поэтому для определения успешности завершения запроса нужно прове-

рить первый аргумент и посмотреть, нет ли чего-нибудь интересного в остальных аргументах.

Это соглашение распространяется на Node.js и многие библиотеки, написанные для нее, включая вышеупомянутую среду тестирования Vows.js. Эта среда также использует функции обратного вызова, вызов которых ожидается с одним аргументом, имеющим значение `error` или `null`. Если этот аргумент имеет значение `null`, то второй аргумент передается прямо контрольному примеру.

Вот, например, контрольный случай, использующий Zombie.js и Vows.js. Он касается посещения веб-страницы и ищет элементы с классом `brains` (ожидая, что не найдет ни одного):

```
var zombie  = require("zombie");
vows.describe("Zombie lunch").addBatch({
  "visiting home page": {
    topic: function() {
      var browser = new zombie.Browser;
      browser.cookies("localhost").update("session_id=5678");
      browser.visit("http://localhost:3003/", this.callback);
    },
    "should find no brains": function(browser) {
      assert.isEmpty(browser.css(".brains"));
    }
  }
});
```

С библиотекой Zombie.js можно делать многое. Например, вы можете сохранять состояние браузера (cookie-файлы, историю, веб-хранилище и т. д.) после запуска одного теста и использовать это состояние для запуска других тестов (для запуска каждого теста из состояния «новая сессия и вошедший в систему пользователь»).

Можно также инициировать события DOM — например, для имитации щелчка кнопкой мыши или ответа на запросы подтверждения и на предупреждения. Вы можете просмотреть историю запросов и ответов, подобную вкладке **Resources** в программе Web Inspector из WebKit. Хотя Zombie запускается в Node.js, она может создавать HTTP-запросы к любому веб-серверу, поэтому ее, конечно же, можно использовать для тестирования ваших Ruby- или Python-приложений.

Ichabod

Библиотека с оригинальным названием Ichabod является еще одной альтернативой для автономного запуска тестов и очень хорошим решением, если важны простота и скорость.

Преимущество Ichabod в том, что вместо имитации DOM и синтаксического анализатора в ней используется WebKit — браузерный движок, лежащий в основе Safari и Chrome. Но Ichabod работает только под управлением OS X, поскольку для нее требуется MacRuby и OS X WebView API.

Установка не составляет особого труда. Сначала нужно установить MacRuby, либо с сайта проекта, либо с rvm. Затем нужно установить gem-пакет Ichabod:

```
$ macgem install ichabod
```

Ichabod в настоящее время поддерживает запущенные тесты Jasmine или QUnit, хотя вскоре будут поддерживаться и дополнительные библиотеки. Нужно просто передать конечное положение теста исполняемому файлу ichabod:

```
$ ichabod --jasmine http://путь/к/jasmine/specs.html
$ ichabod --qunit http://путь/к/qunit/tests.html
```

Тесты необязательно должны быть на хост-узлах, вы можете также передать локальный путь:

```
$ ichabod --jasmine ./tests/index.html
    ...
    Finished in 0.393 seconds
    1 test, 5 assertions, 0 failures
```

Ichabod загрузит все ваши тесты и запустит их в версии WebKit, не имеющей графического интерфейса, прямо из командной строки.

Распределенное тестирование

Одним из решений кроссбраузерного тестирования является привлечение для решения этой задачи внешних выделенных кластеров браузеров. Именно такой подход используется в TestSwarm:

> *Основная цель TestSwarm — взять на себя сложный и затратный по времени процесс запуска тестовых комплексов JavaScript на нескольких браузерах и существенно их упростить. Эта цель достигается путем предоставления всех инструментов, необходимых для создания непрерывного интеграционного технологического процесса для вашего проекта на основе JavaScript.*

Вместо использования дополнительных модулей и расширений для интеграции с браузерами на низком уровне в TestSwarm используется обратный подход. Браузеры открывают конечный адрес TestSwarm и автоматически выполняют переданные им тесты. Они могут находиться на любой машине или операционной системе — для добавления нового браузера нужно всего лишь перейти на имеющийся внутри нее конечный адрес TestSwarm.

Этот простой подход избавляет от множества усилий и трудностей запуска постоянного интеграционного сервера. Все сложности сводятся к гарантии подключения к рою (swarm) подходящего количества браузеров. Разумеется, для привлечения сторонних ресурсов необходимо, как показано на рис. 9.6, наличие активного сообщества, успешно реализующего, наверное, самый реалистичный испытательный стенд, на который можно было бы положиться.

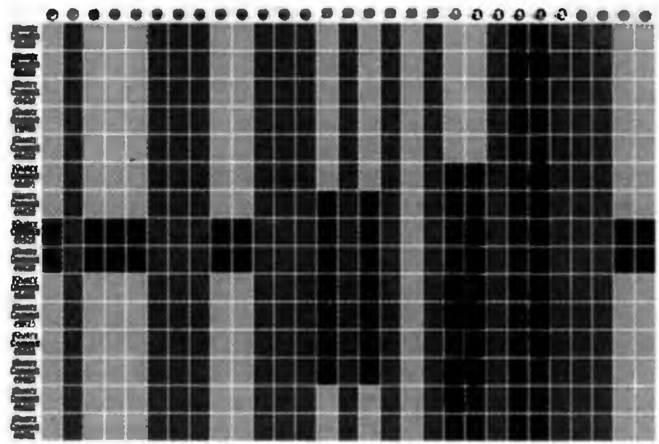

Рис. 9.6. TestSwarm возвращает результаты браузерам в виде табличной структуры тестовых комплексов

Кроме этого можно воспользоваться услугами таких компаний, как Sauce Labs, позволяющих управлять всеми этими браузерами и запускать их в облаке. Для этого нужно просто выложить любые тесты Selenium, а их службы сделают все остальное, запуская тесты на разных браузерах и платформах и удостоверяясь в том, что они проходят.

Предоставление поддержки

Сколько бы тестов вы ни создавали для своего приложения, в нем все равно всегда будут какие-нибудь недостатки. Лучшее, что можно сделать в данной ситуации, — смириться с данным фактом и научить пользователей справляться с недочетами и ошибками. Предоставьте пользователям легкий способ отправки сообщений об ошибках и наладьте систему учета, чтобы получить возможность управлять запросами на поддержку.

Инспекторы

JavaScript-разработка и отладка от оставшихся в прошлом вызовов функции `alert()` прошла уже довольно долгий путь. Сейчас большинство основных браузеров включают такие мощные компоненты, как инспекторы и отладчики, которые упрощают и существенно улучшают веб-разработку. Далее мы собираемся рассмотреть два основных инспектора, но основные понятия должны распространяться на любые выбранные вами для дальнейшего использования инспекторы.

Веб-инспектор

Веб-инспектор доступен при работе с браузерами Safari и Google Chrome. Интерфейс инспектора в этих двух браузерах немного отличается, но функциональные возможности очень близки.

В Safari его нужно включить, установив, как показано на рис. 9.7, флажок **Show Develop menu in menu bar** в **Advanced preferences**.

Рис. 9.7. Включение инспектора в Safari

В Chrome инструментарий разработчика находится на панели инструментов, которую можно использовать для включения инспектора. Кроме этого в обоих браузерах можно щелкнуть правой кнопкой мыши на элементе и выбрать пункт меню «Просмотр кода элемента».

Веб-инспектор, показанный на рис. 9.8, является очень полезным инструментом, позволяющим изучать HTML-элементы, редактировать стили, заниматься отладкой JavaScript и делать многое другое. Если этого еще не произошло, то использование инспектора должно стать частью вашей повседневной JavaScript-разработки.

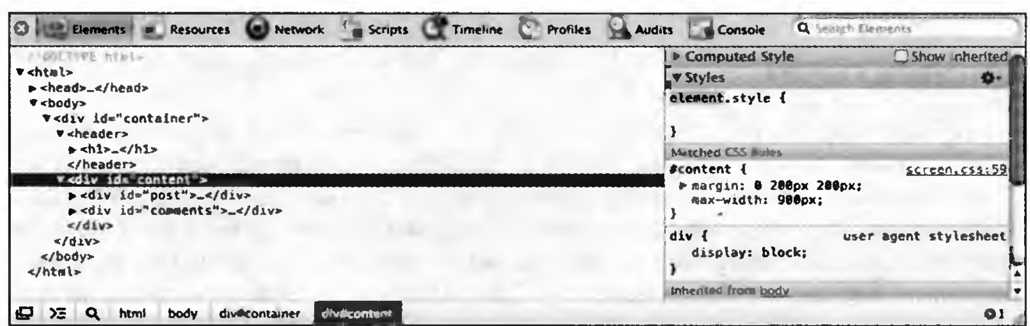

Рис. 9.8. Веб-инспектор Safari позволяет изучать DOM

Мы собираемся подробно рассмотреть большинство его свойств, а пока приведем основной обзор компонентов веб-инспектора.

Elements

Изучение HTML-элементов, редактирование стилей

Resources

Страница ресурсов и средств

Network

HTTP-запросы

Scripts

JavaScript-файлы и отладка

Timeline

Подробное представление визуализации объектов браузера

Audits

Аудитор кода и памяти

Console

Выполнение JavaScript и просмотр протоколирования

Firebug

Изначально в Firefox нет JavaScript-инспектора, но у него есть превосходное дополнение для выполнения этой задачи: Firebug (см. рис. 9.9.)

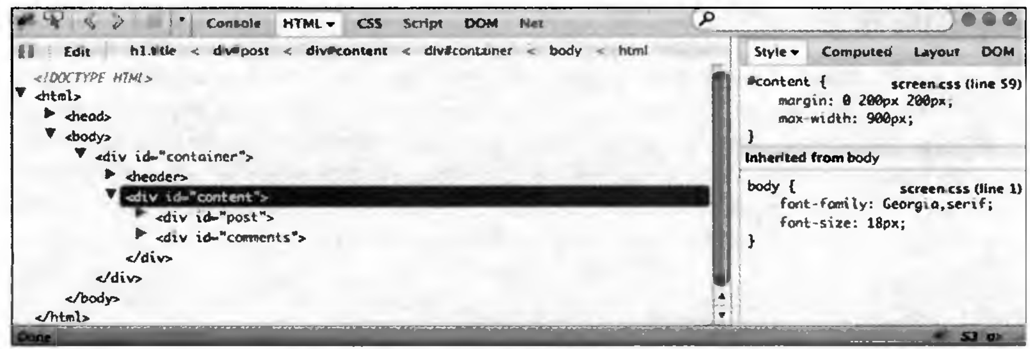

Рис. 9.9. Изучение DOM и CSS с помощью Firebug

Хотя различные компоненты Firebug имеют немного иные названия по сравнению с их аналогами в веб-инспекторе, вы увидите, что их функциональные возможности очень похожи.

Console

Выполнение JavaScript и просмотр протоколирования

HTML

Изучение элементов, редактирование стилей

CSS

Просмотр и редактирование CSS страницы

Script

JavaScript-файлы и отладчик

DOM

Изучение глобальных переменных

Net

HTTP-запросы

Команда Firebug разработала Firefox-независимую версию, Firebug Lite. Это средство имеет множество свойств, позаимствованных у Firebug, а также такой же внешний вид и ощущения от использования, и оно совместимо со всеми основными браузерами. Особенно полезен Firebug Lite для отладки в Internet Explorer (он, скорее всего, превосходит встроенные средства IE). Firebug Lite не требует никакой установки, и он может добавляться к веб-странице с помощью простого script-тега:

```
<script type="text/javascript" src="https://getfirebug.com/firebug-lite.js">
</script>
```

Кроме этого вы можете установить это средство, воспользовавшись вкладкой с веб-сайта Firebug Lite.

Консоль

Консоль позволяет без особого труда выполнять JavaScript и изучать содержимое глобальных переменных, имеющихся на странице. Одним из основных преимуществ консоли заключается в том, что вы можете протоколировать свою работу непосредственно в самой консоли, используя функцию `console.log()`. Вызов осуществляется в асинхронном режиме, может принимать несколько аргументов и проверять эти аргументы, не конвертируя их в строки:

```
console.log("test");
console.log(1, 2, {3: "three"});
```

Существуют и дополнительные виды протоколирования. Для повышения уровня протоколирования, выявляющего первые признаки неправильной работы, можно использовать функции `console.warn()` и `console.error()`:

```
console.warn("последнее предупреждение ");
console.error("что-то сломалось!");
try {
  // Инициирование какого-нибудь действия
} catch(e) {
  console.error("Ошибка приложения!", e);
}
```

Также можно установить пространство имен вызова журнала, используя прокси-функцию:

```
var App = {trace: true};
App.log = function(){
  if (!this.trace) return;
  if (typeof console == "undefined") return;
  var slice = Array.prototype.slice;
  var args  = slice.call(arguments, 0);
  args.unshift("(App)");
  console.log.apply(console, args);
};
```

Функция `App.log()` ставит строку `"App"` впереди своих аргументов, а затем передает вызов в функцию `console.log()`.

Нужно иметь в виду, что при использовании консоли для протоколирования переменная `console` может быть не определена. В браузерах, не имеющих поддержки консоли, таких как Internet Explorer или Firefox без Firebug, объект `console` не будет определен, что станет причиной возникновения ошибки при попытке его использования. Это хороший повод для использования при протоколировании внутри вашего приложения такой прокси-функции как `App.log()`.

Используя функцию `console.trace()`, можно вывести на консоль текущую трассировку стека.

Это особенно пригодится, если вы пытаетесь разобраться в том, как была вызвана функция, поскольку трассировка ведется в направлении, обратном по отношению к ходу выполнения программы:

```
// Журнал трассировки стека
console.trace();
```

Ошибки приложения будут также появляться в консоли, и, пока JIT-компилятор браузера не будет оптимизирован на вызов функции, в консоли будет показана полная трассировка стека.

Вспомогательные функции консоли

С целью экономии времени на набор текста консоль также включает ряд сокращенных команд и вспомогательных функций. Например, переменные от `$0` до `$4` содержат текущий и предыдущие три выбранных узла в веб-инспекторе или в Firebug. И поверьте, вам это очень пригодится, если потребуется получить доступ к элементам и поработать с ними:

```
// $0 содержит текущий выбранный элемент
$0.style.color = "green";
// Или, с использованием jQuery
jQuery($0).css({background: "black"});
```

Функция `$()` возвращает элемент с указанным ID. По сути, она является сокращенным вариантом вызова функции `document.getElementById()`. Такие библиотеки,

как jQuery, Prototype или им подобные, у которых уже используется $, отменят эту установку:

```
$("user").addEventListener("click", function(){ /* ... */});
```

Функция $$() возвращает массив элементов, которые соответствуют заданному CSS-селектору. Она является аналогом функции document.querySelectorAll(). Опять же, если используется библиотека Prototype или MooTools, эта установка будет отменена:

```
// Выбор элементов со значением атрибута class, равным .users
var users = $$(".users");
users.forEach(function(){ /* ... */ });
```

Функция $x() возвращает массив элементов, соответствующий заданному выражению XPath:

```
// Выбор всех форм
var checkboxes = $x("/html/body//form");
```

Функция clear() очищает консоль:

```
clear();
```

Функция dir() выводит интерактивный листинг всех свойств объекта:

```
dir({one: 1});
```

Функции inspect() передается в качестве аргумента элемент, база данных или область хранения, и она автоматически осуществляет переход на соответствующую панель для отображения важной информации:

```
inspect($("user"));
```

Функция keys() возвращает имена всех свойств объекта:

```
// Возвращает ["two"]
keys({two: 2});
```

Функция values() возвращает массив, содержащий значения всех свойств объекта, т. е. она является обратной по отношению к функции keys():

```
// Возвращает [2]
values({two: 2});
```

Использование отладчика

Отладчик JavaScript — это одна из сокровенных тайн JavaScript-разработки. Он является полнофункциональным средством отладки, позволяющим расставлять контрольные точки, просматривать выражения, исследовать значения переменных и в точности выяснять, что происходит в сценарии.

Поставить контрольную точку несложно, для этого нужно просто добавить инструкцию **debugger** в какое-нибудь место кода сценария, где нужно остановить его выполнение с помощью отладчика:

```
var test = function(){
  // ...
  debugger
};
```

Можно также перейти в инспекторе на панель **Scripts**, выбрать соответствующий сценарий и щелкнуть мышью на номере той строки, в которой нужно установить контрольную точку. Пример показан на рис. 9.10.

Последний подход будет, наверное, предпочтительней, поскольку не хочется рисковать тем, что в коде конечного продукта может остаться инструкция **debugger**. Когда при выполнении кода JavaScript попадется контрольная точка, возникнет пауза, позволяющая изучить текущее окружение. Эта ситуация показана на рис. 9.11.

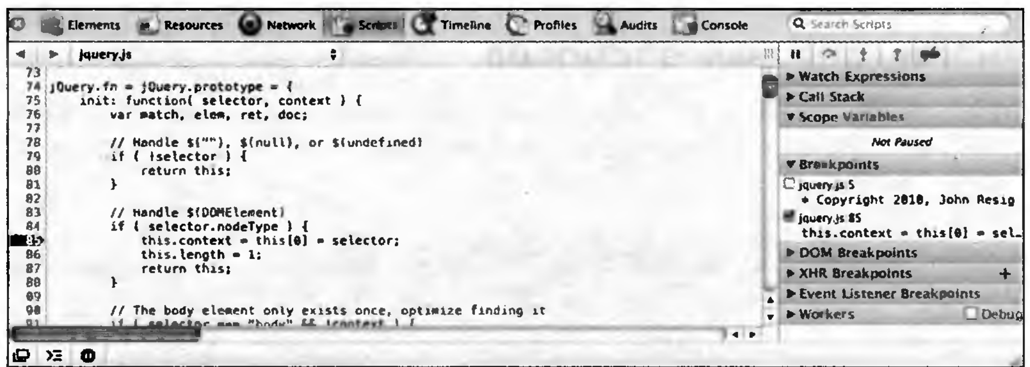

Рис. 9.10. Установка контрольной точки в веб-инспекторе браузера Safari

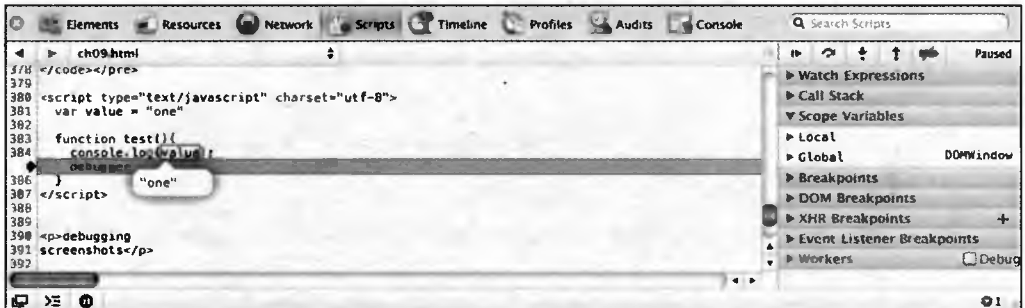

Рис. 9.11. Отладка в контрольной точке веб-инспектора Safari

В правой части имеющейся в инспекторе панели `Scripts` вы можете увидеть полную картину стека вызовов (call stack), локальные и глобальные переменные и другую важную информацию. Чтобы увидеть текущее значение любой переменной, вы можете провести над ней указатель мыши. Консоль даже осуществляет переход к среде окружения контрольной точки, позволяя работать с переменными и вызывать другие функции.

Вы можете продолжить выполнение кода, перейти к вызову следующей функции или обойти этот вызов и управлять текущим стеком, используя расположенную справа панель отладчика. Значки на панели отладчика зависят от конкретно применяемого браузера, но вы можете определить функцию каждого из них, проводя над ними указатель мыши, что вызовет появление желтой подсказки.

Важно запомнить, что контрольные точки остаются и между перезагрузками страницы. Если контрольную точку нужно удалить, следует просто снять отметку в номере ее строки или снять флажок в списке контрольных точек. Отладчик JavaScript является замечательной альтернативой `console.log()`, поскольку он помогает выяснить, что происходит внутри вашего приложения.

Анализ сетевых запросов

Как показано на рис. 9.12, сетевой раздел инспектора показывает все HTTP-запросы, сделанные на странице, время, затраченное на эти запросы, и время их завершения.

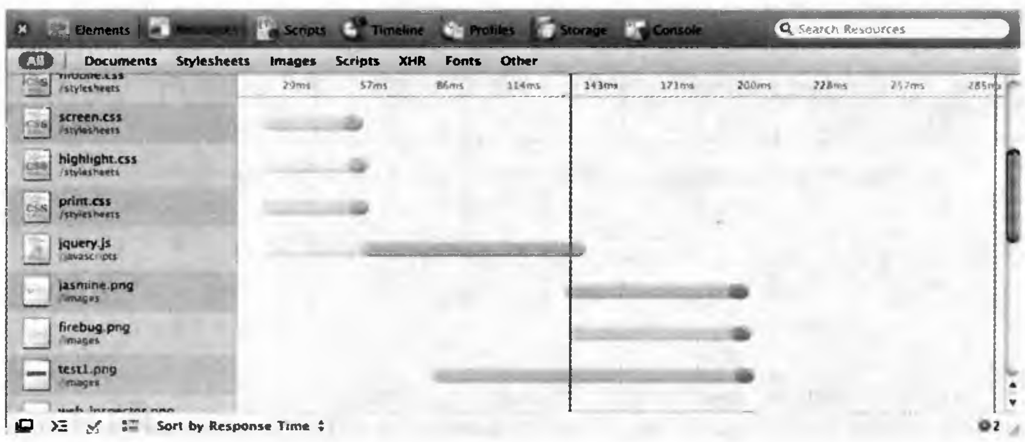

Рис. 9.12. Анализ сетевых запросов с использованием веб-инспектора

Вы можете увидеть исходное время ожидания запроса, которое показано слегка прозрачным цветом. Затем, когда начинают поступать данные, цвет шкалы времени становится более темным. В приведенном выше примере размер файла jQuery значительно больше размера файла таблиц стилей, поэтому, хотя исходное время запроса почти одинаковое, сценарий загружается дольше таблиц.

Если в отношении ваших сценариев не используются настройки async или defer (см. главу 10), вы сможете заметить, что файлы JavaScript загружены последовательно, а не параллельно. Сценарии запрашиваются только после того, как был полностью загружен и выполнен предыдущий запрошенный сценарий. Все остальные ресурсы загружаются в параллельном режиме.

Линии в сетевой линейке времени показывают статус загрузки страницы. Синяя линия появляется при инициировании события документа DOMContentLoaded, иными словами, когда готова DOM-модель. Красная линия появляется, как только будет инициировано событие окна load, когда будут полностью загружены все изображения на странице и загрузка страницы завершится.

В сетевом разделе также показываются полные заголовки запроса и ответа для каждого запроса, что особенно полезно, если нужно удостовериться, что кэширование было применено правильно (рис. 9.13).

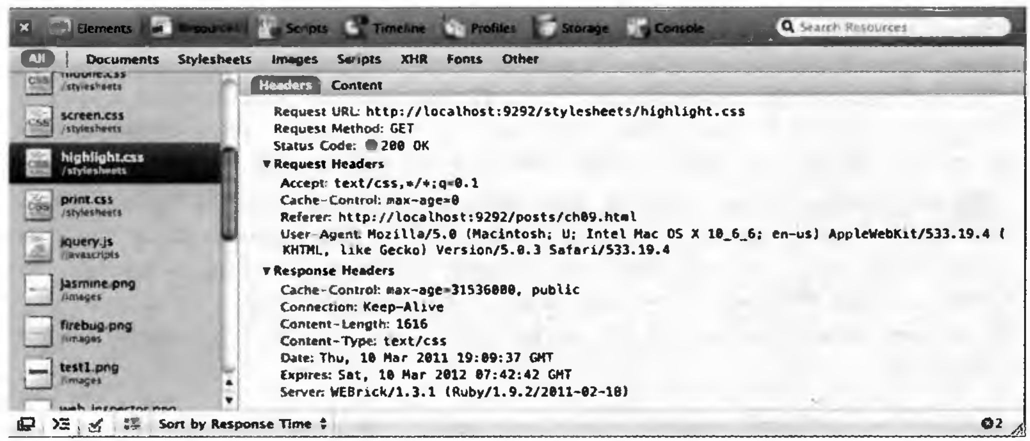

Рис. 9.13. Просмотр всестороннего анализа сетевых запросов, например заголовков запросов и ответов

Профилирование и хронометраж

При создании больших JavaScript-приложений нужно обращать внимание на производительность, особенно при наличии мобильных клиентов. Утилиты профилирования и хронометража, которые могут помочь добиться слаженной работы приложения, есть и в веб-инспекторе, и в Firebug.

Профилирование кода осуществляется довольно просто, для этого нужно просто окружить любой профилируемый код функциями console.profile() и console.profileEnd():

```
console.profile();
// ...
console.profileEnd();
```

Как только будет вызвана функция `profileEnd()`, будет создан профиль, в котором перечисляются все функции (и время, затраченное на каждую из них), которые были вызваны между двумя инструкциями (рис. 9.14).

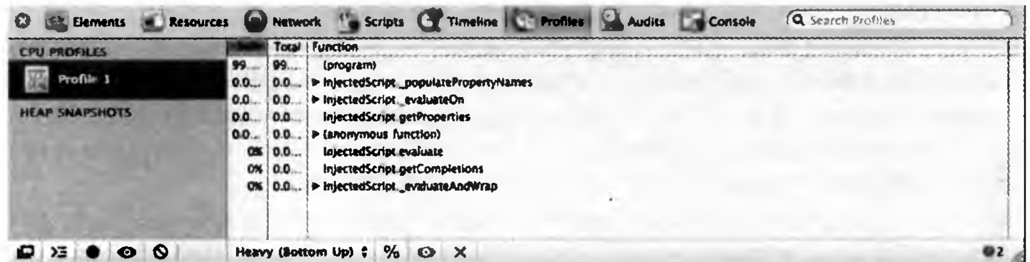

Рис. 9.14. Профилирование показателей выполнения функций с помощью веб-инспектора

Вместо этого можно воспользоваться свойством `record` (запись) принадлежащего инспектору создателя профилей, который эквивалентен заключению кода в инструкции профилирования консоли. Путем просмотра вызванных функций и определения тех из них, которые требовали больше времени на свое завершение, вы можете обнаружить узкие места в своем коде и провести оптимизацию.

Профилировщик также позволяет вам снять моментальную копию текущей динамической памяти страницы, показанную на рис. 9.15. Она покажет вам количество объектов, которым выделена память, и объем памяти, занятой под страницу. Это отличный способ выявления утечек памяти, поскольку вы видите непреднамеренно сохраняемые объекты, которые впоследствии не могут быть удалены сборщиком мусора.

Constructor	Count	Size	± Count	± Size
▸ (code)	3327	1.03MB	0	0B
▸ String	13300	410.46KB	0	0B
▸ (closure)	2677	404.12KB	0	0B
▸ Object	1547	168.62KB	0	0B
▸ Array	295	14.53KB	0	0B
▸ RegExp	352	12.38KB	0	0B
▸ (global property)	1517	11.85KB	0	0B
▸ DOMWindow	6	6.23KB	0	0B
▸ Error	18	504B	0	0B
▸ HTMLElement	15	420B	0	0B

Рис. 9.15. Наблюдение общей панорамы динамической памяти с помощью веб-инспектора

Консоль также позволяет замерять продолжительность выполнения того или иного фрагмента кода. Для этого используется API, похожий на используемый в профилировщике, — код просто нужно заключить в вызовы функций `console`.

`time(имя)` и `console.time End(имя)`. Если вам не удастся поместить все в одну строку, точный хронометраж кода внутри консоли JavaScript не получится, вместо этого придется добавлять инструкции хронометрирования непосредственно в ваши сценарии:

```
console.time("timeName");
// ...
console.timeEnd("timeName");
```

При вызове функции `timeEnd()` время в миллисекундах, затраченное на выполнение кода между двумя инструкциями хронометрирования, отправляется в журнал консоли. Используя API хронометрирования консоли, вы можете потенциально вставить средства сравнения эффективности в тесты своего приложения, чтобы убедиться, что дополнительный код не слишком отрицательно отразился на имевшейся производительности выполнения приложения.

10 Развертывание

Правильное развертывание вашего веб-приложения играет не менее важную роль, чем его разработка. Не имеет смысла создавать очередной Facebook, если он загружается слишком медленно. Пользователь ждет от вашего сайта максимальной надежности и скорости работы, с хорошими показателями безотказной работы.

Развертывание JavaScript и HTML кажется совсем несложным, ведь это же, в конце концов, статические средства, но на самом деле тут есть над чем поработать. Зачастую этой части создания веб-приложения уделяется недостаточно внимания.

К счастью, существуют испытанные и проверенные технологии, применимые ко всем JavaScript-приложениям и несомненно подходящие для любого вида статических средств. Если вы последуете изложенным ниже рекомендациям, то у вас вполне может получиться производство проворных веб-приложений.

Производительность

Один из наиболее простых способов повышения производительности является и наиболее очевидным: это минимизация количества HTTP-запросов. Каждый HTTP-запрос содержит большой объем заголовков, а также создает издержки на выполнение протокола TCP. Сведение отдельных подключений к абсолютному минимуму гарантирует более быструю загрузку страниц для пользователей. Разумеется, это распространяется и на объем тех данных, которые должен передать сервер. Поддержание небольшого объема файлов страницы и ее ресурсов сократит любое время сетевой работы, которая является настоящим узким местом любого веб-приложения.

Связывание сценариев в единый сценарий и объединение CSS в единую страницу стилей сократит количество HTTP-подключений, необходимых для визуализации страницы. Это можно сделать во время развертывания или во время выполнения приложения. В последнем случае нужно позаботиться о кэшировании любых сгенерированных файлов при производстве веб-содержимого.

Используйте CSS-спрайты для объединения изображений в одно полное изображение. Затем для отображения соответствующих изображений на своей стра-

нице используйте CSS-свойства `background-image` и `background-position`. Вам останется лишь определить границы координат фона.

Минимизации количества HTTP-запросов также способствует уклонение от перенаправлений. Кажется, что это случается довольно редко, но чаще всего перенаправление происходит из-за пропуска замыкающего слеша (/) в URL, который должен содержать слэш. Например, переход на http://facebook.com перенаправит вас на http://facebook.com/. При использовании Apache этот недостаток можно устранить, используя модуль Alias или mod_rewrite.

Важно также понимать, как ваш браузер загружает ресурсы. Для ускорения визуализации страницы современные браузеры загружают требуемые ресурсы в параллельном режиме. Но визуализация страницы не может начаться, пока не закончится загрузка всех таблиц стилей и сценариев. Некоторые браузеры идут еще дальше, блокируя все остальные загрузки, пока не будут обработаны все JavaScript-файлы.

Большинство сценариев нуждается в доступе к DOM-модели и добавляет такие компоненты, как обработчики событий, выполняя все это уже после загрузки страницы. Иными словами, браузер совершенно напрасно сдерживает визуализацию страницы до тех пор, пока не завершится вся загрузка, снижая тем самым производительность. Эту проблему можно решить, установив для сценариев атрибут задержки `defer`, чтобы сообщить браузеру о том, что сценарию не нужно работать с DOM, пока страница не загрузилась:

```
<script src="foo.js" type="text/javascript" charset="utf-8" defer></script>
```

Сценарии с атрибутом `defer`, имеющим значение `"defer"`, будут загружаться параллельно с другими ресурсами и не будут препятствовать визуализации страницы. В HTML5 также был представлен новый режим загрузки и выполнения сценария, названный `async`. При установке атрибута `async` сценарий будет выполнен при первой же возможности после завершения своей загрузки. Это означает наличие возможности (и вероятности) выполнения `async`-сценариев не в том порядке, в котором они появляются на странице, оставляя возможность возникновения ошибок зависимости. Но если сценарий не имеет никаких зависимостей, `async` будет весьма полезным средством. К примеру, Google Analytics пользуется им по умолчанию:

```
<script src="http://www.google-analytics.com/ga.js" async></script>
```

Кэширование

Если бы не кэширование, Интернет впал бы в коллапс из-за объема сетевого трафика. При кэшировании на локальной машине сохраняются самые последние запрошенные ресурсы, чтобы последующие запросы могли обслуживаться прямо с диска, без повторной загрузки. Важно указать браузерам на необходимость кэширования. Некоторые браузеры, такие как Chrome, по умолчанию будут принимать свои собственные решения, но на них не нужно полагаться.

Для статических ресурсов сделайте кэш «никогда» не устаревающим, путем добавления заголовка `Expires`, указывающего на далекое будущее. Тем самым будет гарантировано, что браузер загрузит ресурс только один раз, и затем эта установка должна быть сделана для всех статических компонентов, включая сценарии, таблицы стилей и изображения.

```
Expires: Thu, 20 March 2015 00:00:00 GMT
```

Дату истечения срока нужно установить относительно текущей даты на весьма отдаленное будущее. Показанный выше пример сообщает браузеру, что кэш не устареет до 20 марта 2015 года. Если используется веб-сервер Apache, вы можете установить относительную дату истечения срока, воспользовавшись `ExpiresDefault`:

```
ExpiresDefault "access plus 5 years"
```

А если нужно, чтобы срок годности ресурса истек до этого времени? Здесь пригодится технология добавления времени модификации файла (или `mtime`) в качестве параметра запроса для URL-адресов, ссылающихся на этот файл. К примеру, Rails делает это по умолчанию. Затем, когда файл будет модифицирован, URL ресурса изменится, очищая кэш.

```
<link rel="stylesheet" href="master.css?1296085785" type="text/css">
```

В протоколе HTTP 1.1. был представлен новый класс заголовков — `Cache-Control` — для предоставления разработчикам усовершенствованного кэширования и для обращения к срокам годности `Expires`. В заголовке `Cache-Control` можно передавать несколько настроек, разделенных запятыми:

```
Cache-Control: max-age=3600, must-revalidate
```

Полный список настроек можно посмотреть в спецификациях. А те из них, которые, скорее всего, будут востребованы в настоящее время, перечислены ниже.

`max-age`

Указывает максимальное количество времени в секундах, в течение которого данный ресурс будет считаться свежим. В отличие от параметра `Expires`, имеющего абсолютное значение, эта директива задается относительно времени отправки запроса.

`public`

Помечает ресурсы пригодными для кэширования. По умолчанию, если ресурсы обслуживаются через SSL или если используется HTTP-аутентификация, кэширование выключается.

`no-store`

Полностью выключает кэширование, что иногда требуется исключительно для динамического контента.

`must-revalidate`

Сообщает кэш-блокам, что они должны прислушиваться к любой информации, передаваемой им относительно свежести ресурсов. При определенных условиях HTTP позволяет кэш-блокам обслуживать несвежие ресурсы в соответствии

с их собственными правилами. Указывая данный заголовок, вы сообщаете кэш-блоку, что от него требуется четко следовать вашим правилам.

Кэшированию может помочь добавление к поданному ресурсу заголовка `Last-Modified`. С последующими запросами ресурса браузеры могут указать заголовок `If-Modified-Since`, содержащий отметку времени. Если ресурс со времени последнего запроса не был модифицирован, сервер может просто вернуть статус 304 (не модифицирован). Браузер по-прежнему должен сделать запрос, но сервер не должен включать ресурс в свой ответ, экономя сетевое время и трафик:

```
# Запрос
GET /example.gif HTTP/1.1
Host:www.example.com
If-Modified-Since:Thu, 29 Apr 2010 12:09:05 GMT

# Ответ
HTTP/1.1 200 OK
Date: Thu, 20 March 2009 00:00:00 GMT
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-revalidate
Expires: Fri, 30 Oct 1998 14:19:41 GMT
Last-Modified: Mon, 17 March 2009 00:00:00 GMT
Content-Length: 1040
Content-Type: text/html
```

У заголовка `Last-Modified` существует альтернатива: `ETags`. Сравнение `ETags` похоже на сравнение хэшей двух файлов — если `ETags` разные, значит, кэш устарел и должен быть повторно утвержден. Это работает точно так же, как заголовок `Last-Modified`. Сервер присоединит `ETag` к ресурсу с помощью заголовка `ETag`, а клиент проверит заголовки `ETag` с заголовком `If-None-Match`:

```
# Запрос
GET /example.gif HTTP/1.1
Host:www.example.com
If-Modified-Since:Thu, 29 Apr 2010 12:09:05 GMT
If-None-Match:"48ef9-14a1-4855efe32ba40"

# Ответ
HTTP/1.1 304 Not Modified
```

`ETags` обычно создаются с атрибутами, специфичными для сервера, т. е. два разных сервера для одного и того же ресурса дадут разные `ETags`. По мере того как кластеры получают все большее распространение, это становится настоящей проблемой. Лично я советую вам остановить свой выбор на заголовке `Last-Modified` и полностью отключить `ETags`.

Минификация

Минификация JavaScript сокращает количество ненужных в сценариях символов, не меняя их функциональности. Сокращаются пробельные символы, символы

новых строк и комментарии. Самые лучшие минификаторы способны интерпретировать JavaScript. Благодаря этому они могут совершенно безопасно сокращать имена переменных и функций, сокращая количество символов. Чем меньше размер файла, тем лучше, поскольку по сети передается меньше данных.

Но минифицирован может быть не только код JavaScript. Обработке можно также подвергнуть таблицы стилей и HTML. Таблицы стилей особенно склонны к наличию лишних пробельных символов. Минификация относится к тем действиям, которые лучше выполнять в процессе развертывания, поскольку минифицированный код не требует отладки. Если в процессе эксплуатации выявится ошибка, попробуйте воспроизвести ее сначала в среде разработки — вы поймете, что отладку будет вести намного проще.

Минификация также имеет дополнительное преимущество, существенно затрудняя изучение вашего кода. Конечно, серьезно заинтересованный в этом специалист, наверное, сможет его реконструировать, но такой код станет барьером для случайного наблюдателя.

Существует множество библиотек, минимизирующих код, но я советую выбрать вам одну из тех, у которых есть движок JavaScript, способный интерпретировать код. Мне больше всего нравится YUI Compressor, поскольку эта библиотека хорошо обслуживается и поддерживается. Библиотека создана специалистом Yahoo! Жульеном Лекомте (Julien Lecomte) с целью еще большего сжатия JavaScript-файлов, чем при использовании JSMin, путем применения разумной оптимизации исходного кода. Предположим, что у нас есть следующая функция:

```
function per(value, total) {
  return( (value / total) * 100 );
}
```

Кроме удаления пробельных символов YUI Compressor укоротит имена локальных переменных, экономя еще больше символов:

```
function per(b,a){return((b/a)*100)};
```

Поскольку YUI Compressor проводит анализ кода JavaScript, он обычно заменяет имена переменных, не совершая ошибок в коде. Но так бывает не всегда, иногда компрессор не может распознать ваш код, поэтому он оставляет его без изменений. Наиболее частой причиной этого является использование инструкций `eval()` или `with()`. Если компрессор определит, что вы используете любую из этих инструкций, он не осуществляет замену имен переменных. Кроме того, как `eval()`, так и `with()` могут вызвать проблемы производительности — у JIT-компилятора браузера возникают те же трудности, что и у компрессора. Я советую избегать применения обеих этих инструкций.

Проще всего использовать YUI Compressor путем загрузки исполняемых файлов, требующих Java, и выполнения этих файлов в командной строке:

```
java -jar yuicompressor-x.y.z.jar foo.js | foo.min.js
```

Но вы можете сделать это программным способом при развертывании. Если вы используете такую библиотеку, как Sprockets или Less, то она сделает это за вас. Или же существует несколько библиотек интерфейсов к YUI Compressor, таких как gem-пакет Сэма Стефенсона (Sam Stephenson) Ruby-YUI-compressor или библиотека Jammit.

Сжатие с помощью Gzip

Gzip является наиболее популярным и поддерживаемым методом, применяемым при веб-разработках. Он был разработан в рамках GNU-проекта, и его поддержка была добавлена в HTTP/1.1. Веб-клиент показывает, что он поддерживает сжатие путем отправки вместе с запросом заголовка `Accept-Encoding`:

```
Accept-Encoding: gzip, deflate
```

Если веб-сервер видит этот заголовок и поддерживает любой из перечисленных типов сжатия, он может сжать свой ответ и показать это с помощью заголовка `Content-Encoding`:

```
Content-Encoding: gzip
```

Затем браузеры могут соответствующим образом декодировать ответ. Разумеется, сжатые данные могут сократить сетевое время, но истинная степень сокращения часто недооценивается. Gzip обычно уменьшает размер ответа на 70%, а это существенное сокращение.

Серверы обычно должны быть настроены на те типы файлов, которые должны быть сжаты с помощью gzip. Нужно взять за правило сжимать с помощью gzip любой текстовый ответ, например HTML, JSON, JavaScript и таблицы стилей. Если файлы уже сжаты, как изображения и PDF-файлы, они не должны обрабатываться с помощью gzip, поскольку повторное сжатие не уменьшит их размер.

Настройка gzip зависит от вашего сервера, но если вы используете Apache 2.x или более поздней версии, вам нужен модуль mod_deflate. Для других веб-серверов нужно обратиться к их документации.

Использование CDN

Сеть доставки контента, или CDN (content delivery network), может обслуживать статический контент от вашего имени, сокращая время его загрузки. Близость пользователя к веб-серверу сети может повлиять на время загрузки.

CDN-сети могут развернуть ваш контент на нескольких географически разнесенных серверах, поэтому когда пользователь запрашивает ресурс, он может быть обслужен ближайшим сервером (в идеале, находящимся в той же стране). Компания Yahoo! обнаружила, что CDN-сети могут улучшить время отклика в адрес конечного пользователя на 20% и более.

В зависимости от того, сколько вы можете позволить себе потратить, можно воспользоваться услугами множества компаний, предлагающих CDN-сети, среди

которых Akamai Technologies, Limelight Networks, EdgeCast и Level 3 Communications. Компания Amazon Web Services недавно выпустила вполне доступное средство под названием Cloud Front, которое тесно привязывается к ее службе S3 и может стать неплохим вариантом для компаний-новичков.

Google предлагает бесплатную CDN-сеть и загрузку архитектуры для многих популярных JavaScript-библиотек с открытым кодом, включая jQuery и jQueryUI. Одним из преимуществ использования CDN компании Google является то, что ею пользуются и многие другие сайты, повышая вероятность наличия в кэше пользовательского браузера запрашиваемых JavaScript-файлов.

Проверьте список доступных библиотек. Если, скажем, вам нужно включить библиотеку jQuery, вы можете либо воспользоваться библиотекой JavaScript-загрузчика компании Google, либо, что проще, старым добрым script-тегом:

```
<!-- минимизированная версия библиотеки jQuery -->
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min.js">
</script>
<!-- минимизированная версия библиотеки jQueryUI -->
<script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.8.6/jquery-ui.min.js">
</script>
```

Можно заметить, что в приведенном выше примере не указан протокол, а вместо этого используются символы //. Это малоизвестный трюк, обеспечивающий извлечение файла сценария с использованием того же самого протокола, который использовался для страницы-хозяина. Иными словами, если ваша страница загружалась в безопасном режиме с использованием HTTPS, файл сценария будет также загружен по этому протоколу, исключая любые предупреждения о безопасности. Можно использовать относительный URL-адрес без схемы, который совместим с RTF-спецификацией. Более важным обстоятельством является повсеместная поддержка: URL-адреса с относительным протоколом работают даже в Internet Explorer 3.0.

Аудиторы

Есть весьма полезные инструментальные средства, дающие быстрое представление о производительности вашего сайта. YSlow является расширением Firebug, а этот модуль, в свою очередь, является расширением Firefox. Чтобы воспользоваться этим средством, нужно загрузить все три компонента. После установки его можно использовать для аудита веб-страниц. Расширение запустит серию проверок, включая кэширование, минификацию, сжатие с помощью gzip и использование CDN-сетей. В зависимости от достигнутого результата, сайту будет выставлена определенная оценка, а затем будет дан совет по повышению производительности.

У Google Chrome и у Safari также имеются аудиторы, но они встроены непосредственно в браузеры. Как показано на рис. 10.1, в Chrome нужно просто перейти

в раздел Audits веб-инспектора и щелкнуть на кнопке Run (Запуск). Это отличный способ посмотреть, что можно улучшить на вашем сайте, чтобы повысить его производительность.

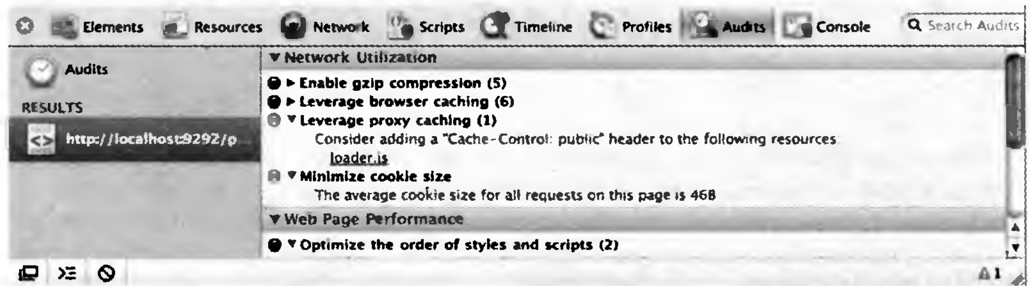

Рис. 10.1. Аудит производительности веб-страницы с помощью веб-инспектора

Ресурсы

Как Yahoo!, так и Google прилагают большие усилия к анализу веб-производительности. Особый интерес уделяется увеличению скорости визуализации сайтов, как для своих собственных служб, так и в восприятии своих клиентов при просмотре ими более обширной всемирной сети. В настоящее время Google с помощью своего алгоритма Pagerank берет в расчет и скорость, помогая определять ранг сайтов в поисковых запросах. У обеих компаний имеются замечательные ресурсы, помогающие повысить производительность. Их можно найти на сайтах разработчиков Google и Yahoo!.

11 Библиотека Spine

Spine является небольшой по объему библиотекой для разработки JavaScript-приложений, в которой используется множество понятий, рассмотренных в данной книге, — например MVC, события и классы. Когда я говорю «небольшой по объему», я имею в виду размер кода библиотеки — в ней все лишь около 500 строк JavaScript, которые в минифицированном и сжатом виде занимают около 2 Кб. Но это не должно создавать у вас какое-то ложное представление. Spine позволит вам создавать полнофункциональные JavaScript-приложения, гарантируя при этом чистоту и необходимую обособленность вашего кода.

Я создал Spine при написании данной книги, поскольку не смог найти MVC-среды, работающей на стороне клиента, которая всецело отвечала моим потребностям. При создании этой библиотеки я постарался вместить в нее все лучшие приемы работы, предлагаемые в данной книге, и, разумеется, приложение Holla, ставшее в ней примером, создано с использованием Spine.

В отличие от библиотек, основанных на использовании виджетов, к числу которых можно отнести Cappuccino и SproutCore, библиотека Spine не содержит каких-либо решений, касающихся отображения данных пользователю. В ней ставка сделана на гибкость и простоту. Spine предоставляет вам основные элементы, не требуя к себе особого внимания, поэтому вы сразу можете приступать к самому приятному — разработке приложений.

Spine включает в себя библиотеку классов с поддержкой наследования — `Spine.Class`, модуль событий — `Spine.Events`, ORM — `Spine.Model` и класс контроллеров — `Spine.Controller`. Все остальное, что может понадобиться, например поддержка шаблонов или DOM-библиотека, остается за вами, поэтому вы можете воспользоваться тем, с чем вы знакомы лучше всего. Следует заметить, что Spine включает конкретную поддержку библиотек jQuery и Zepto.js, которые прекрасно ее дополняют.

В настоящее время слабой стороной Spine является отсутствие документации. Но, учитывая юность данной библиотеки, ситуация с документацией наверняка улучшится. На данный момент эта глава предоставит вам неплохое введение, а приводимые в качестве примеров приложения дадут дальнейшее объяснение.

Установка

Просто загрузите Spine из репозитория проекта и включите ее в вашу страницу. У Spine нет никаких зависимостей:

```
<script src="spine.js" type="text/javascript" charset="utf-8"></script>
```

Благодаря переменной `Spine`, сама библиотека полностью работает в своем собственном пространстве имен, поэтому она не должна конфликтовать с какими-нибудь другими переменными. Вы можете спокойно включать такие библиотеки, как jQuery, Zepto или Prototype, не испытывая при этом никаких осложнений.

Классы

Практически каждый объект в Spine инкапсулирован в класс. Но классы в Spine созданы с помощью `Object.create()` и простого прототипного наследования, рассмотренного в главе 3, что отличается от конструкций большинства других абстракций класса.

Для создания нового класса нужно вызвать функцию `Spine.Class.create([instanceProperties, classProperties])`, передав ей необязательный набор свойств экземпляра (instanceProperties) и класса (classProperties):

```
var User = Spine.Class.create({
  name: "Caroline"
});
```

В показанном выше примере у экземпляров класса `User` теперь по умолчанию будет свойство `name`. Закулисно функция `create()` создает новый объект, чей прототип установлен в `Spine.Class`, — т. е. наследуется из этого модуля. Если нужно создать последующие подклассы, следует просто вызвать `create()` в отношении их родительского класса:

```
var Friend = User.create();
```

Теперь `Friend` является подклассом класса `User`, и он унаследует все свойства этого класса:

```
assertEqual( Friend.prototype.name, "Caroline" );
```

Создание экземпляра

Поскольку вместо функции-конструктора мы используем простые прототипные объекты и наследование, мы не можем для создания экземпляров использовать ключевое слово `new`. Вместо этого в Spine используется функция `init()`:

```
var user = User.init();
assertEqual( user.name,  "Caroline" );

user.name = "Trish";
assertEqual( user.name, "Trish" );
```

Любые аргументы, переданные функции `init()`, будут отправлены в функцию инициализатора экземпляров класса `init()`:

```
var User = Spine.Class.create({
  init: function(name){
    this.name = name;
```

продолжение ⇗

```
  }
});

var user = User.init("Martina");
assertEqual( user.name, "Martina" );
```

Расширение классов

Кроме установки свойств класса и экземпляра во время создания, вы можете воспользоваться функциями `include()` и `extend()`, передавая им объектный литерал:

```
User.include({
  // Свойства экземпляра
});

User.extend({
  // Свойства класса
});
```

Функции `include()` и `extend()` прокладывают путь к модулям, являющимся повторно используемыми фрагментами кода, которые можно включать в сценарий несколько раз:

```
var ORM = {
  extended: function(){
    // вызывается при расширении
    //   this === User
  },
  find:  function(){ /* ... */ },
  first: function(){ /* ... */ }
};

User.extend( ORM );
```

При включении или расширении модуля вы можете получить функцию обратного вызова. В приведенном выше примере функция `extended` будет вызвана при вызове `User.extend()` в контексте класса `User`. Точно так же, если у модуля будет свойство `included`, соответствующая функция будет вызвана при включении модуля в класс.

Поскольку мы используем наследование на основе прототипа, любые свойства, добавляемые в классы, будут во время выполнения сценария динамически отражаться на всех подклассах:

```
var Friend = User.create();

User.include({
  email: "info@eribium.org"
});

assertEqual( Friend.init().email, "info@eribium.org" );
```

Свойства в подклассах могут подменяться без влияния на родительский класс. Но модификация таких объектов в подклассах, как массивы, будет отражаться на всем дереве наследований. Если нужно, чтобы объект был определен для класса

или экземпляра, его нужно создать при первой инициализации класса или экземпляра. Это можно сделать в функции `created()`, которую Spine будет вызывать при первой установке или инициализации класса:

```
// Нам нужно, чтобы массив records был определен для класса
var User = Spine.Class.create({
  // Вызывается при инициализации
  init: function(){
    this.attributes = {};
  }
}, {
  // Вызывается при создании класса
  created: function(){
    this.records = [];
  }
});
```

Контекст

Изменение контекста внутри программ JavaScript — явление довольно привычное, поэтому `Spine.Class` включает несколько полезных методов для управления областью видимости. Для иллюстрации этой проблемы рассмотрим следующий пример:

```
var Controller = Spine.Class.create({
  init: function(){
    // Добавление слушателя события
    $("#destroy").click(this.destroy);
  },

  destroy: function(){
    // Функция destroy вызывается с неверным контекстом,
    // поэтому любые ссылки на 'this' вызовут проблемы
    // Следующее утверждение будет несостоятельным:
    assertEqual( this, Controller.fn );
  }
});
```

В показанном выше примере при наступлении события функция `destroy()` будет вызвана с контекстом элемента `#destroy`, а не с контекстом `Controller`. Чтобы справиться с этой проблемой, можно воспользоваться представительством контекста, заставив его быть в точности таким, каким вы его определили. Для этого Spine предоставляет вам функцию `proxy()`:

```
var Controller = Spine.Class.create({
  init: function(){
    $("#destroy").click(this.proxy(this.destroy));
  },

  destroy: function(){ }
});
```

Если обнаружится, что вы постоянно используете представительство контекста функции, может потребоваться переписать ее, чтобы она всегда включала представительство. Именно для этого Spine включает в себя функцию proxyAll():

```
var Controller = Spine.Class.create({
  init: function(){
    this.proxyAll("destroy", "render")
    $("#destroy").click(this.destroy);
  },

  // Теперь функции всегда вызываются с правильным контекстом
  destroy: function(){ },
  render:  function(){ }
});
```

Функции proxyAll() передаются несколько имен функций, и при вызове этих функций она будет их переписывать, заменяя в них область видимости на текущую. Тем самым будет гарантироваться постоянное выполнение функций destroy() или render() в локальном контексте.

События

События являются ключевым фактором для Spine, и они часто используются внутри этой библиотеки. Функциональные возможности Spine в области событий содержатся в модуле Spine.Events, который может быть включен туда, где он нужен. Давайте, к примеру, добавим поддержку событий к классу библиотеки Spine:

```
var User = Spine.Class.create();
User.extend(Spine.Events);
```

Модуль Spine.Events предоставляет для обработки событий три функции:

- bind(eventName, callback)
- trigger(eventName, [*data])
- unbind(eventName, [callback])

Если используется API событий, имеющийся в jQuery, то вам все это будет знакомо. Давайте, к примеру, привяжем событие к нашему классу User и инициируем его:

```
User.bind("create", function(){ /* ... */ });
User.trigger("create");
```

Для привязки нескольких событий к одной функции обратного вызова нужно просто разделить имена событий пробелами:

```
User.bind("create update", function(){ /* ... */ });
```

Функции `trigger()` передается имя события, а необязательные аргументы она передает функциям обратного вызова, обрабатывающим события:

```
User.bind("countChange", function(count){
  // Аргумент 'count' передается функцией trigger
  assertEqual(count, 5);
});

User.trigger("countChange", 5);
```

Чаще всего вы будете использовать события Spine для связывания данных, присоединяя модели ваших приложений к их представлениям. Подробнее этот вопрос будет рассмотрен в разделе «Создание диспетчера контактов».

Модели

Если взглянуть на код библиотеки Spine, вы увидите, что основной упор в ней сделан на работу с моделями, и это правильно, поскольку модели являются центральной частью любого MVC-приложения. Модели занимаются сохранением данных вашего приложения и работой с ними, а Spine упрощает эту работу, предоставляя полное объектно-реляционное отображение (ORM).

Для создания новой модели вместо использований функции `create()`, которая уже зарезервирована, следует использовать функцию `Spine.Model.setup(name, attrs)`, передавая ей имя модели и массив с именами атрибутов:

```
// Создание новой модели Task.
var Task = Spine.Model.setup("Task", ["name", "done"]);
```

Для добавления свойств экземпляра и класса следует использовать функции `include()` и `extend()`:

```
Task.extend({
  // Возвращение всех выполненных заданий.
  done: function(){ /* ... */ }
});

Task.include({
  // Имя по умолчанию
  name: "Пусто...",
  done: false,

  toggle: function(){
    this.done = !this.done;
  }
});
```

При создании экземпляра записи нужно передать дополнительный объект, содержащий исходные свойства записи:

```
var task = Task.init({name: "Выгулять собаку "});
assertEqual( task.name, "Выгулять собаку" );
```

Установка и извлечение атрибутов — то же самое, что и установка и извлечение свойств в отношении обычного объекта. Кроме этого, функция `attributes()` возвращает объектный литерал, содержащий все атрибуты записи:

```
var task = Task.init();
task.name = "Прочитать статью";
assertEqual( task.attributes(), {name: "Прочитать статью"});
```

Сохранение новых или существующих записей сводится к простому вызову функции `save()`. При сохранении записи, если ID еще нет, он будет сгенерирован, затем запись будет локально сохраняться в памяти:

```
var task = Task.init({name: "Завершить написание книги"});
task.save();

task.id //=> "44E1DB33-2455-4728-AEA2-ECBD724B5E7B"
```

Записи могут быть извлечены с помощью имеющейся у модели функции `find()`, которой передается ID записи:

```
var task = Task.find("44E1DB33-2455-4728-AEA2-ECBD724B5E7B");
assertEqual( task.name, "Завершить написание книги");
```

Если не существует ни одной записи с заданным ID, будет выдано исключение. Можно проверить существование записи, не опасаясь выдачи исключения, воспользовавшись для этого функцией `exists()`:

```
var taskExists = Task.exists("44E1DB33-2455-4728-AEA2-ECBD724B5E7B");
assert( taskExists );
```

Можно удалить запись из локальной кэш-памяти, воспользовавшись функцией `destroy()`:

```
var task = Task.create({name: "Поблагодарить за все накладки"});
assert( task.exists() );
task.destroy();
assertEqual( task.exists(), false );
```

Извлечение записей

Извлечение записей по их ID является единственным способом их получения. Обычно имеет смысл последовательно перебрать все записи или вернуть отфильтрованный набор записей. Spine позволяет сделать это с помощью функций `all()`, `select()` и `each()`:

```
// Возвращение всех заданий (tasks)
Task.all(); //=> [Объект]
```

```
// Возвращение всех заданий с атрибутом выполнения (done), равным false
var pending = Task.select(function(task){ return !task.done });

// Инициирование функции обратного вызова для каждого задания
Task.each(function(task){ /* ... */ });
```

Кроме этого, Spine предоставляет несколько вспомогательных функций для поиска записей по атрибуту:

```
// Поиск первого задания с указанным значением атрибута
Task.findByAttribute(имя, значение);    //=> Объект

// Поиск всех заданий с указанным значением атрибута
Task.findAllByAttribute(имя, значение); //=> [Объект]
```

События моделей

К модели можно привязать события, чтобы инициировать функцию обратного вызова при изменении записей:

```
Task.bind("save", function(record){
  console.log(record.name, "сохранена!");
});
```

Если запись была обработана, она будет передана функции обратного вызова события. Вы можете привязать слушателя к модели, чтобы получить глобальные функции обратного вызова для каждой записи, или же вы можете привязать слушателя к конкретной записи:

```
Task.first().bind("save", function(){
  console.log(this.name, "сохранена!")
});
Task.first().updateAttributes({name: "Попить чай с королевой"});
```

Хотя с помощью функции `trigger()` вы, конечно же, можете создать свои собственные события, вам доступны следующие готовые события.

save

Запись была сохранена (либо после создания, либо после обновления)

update

Запись была обновлена

create

Запись была создана

destroy

Запись была удалена

change

Все вышеперечисленное, т. е. запись была создана (обновлена, уничтожена)

refresh

Все записи были забракованы и заменены

error

Проверка завершилась неудачно

Всю важность событий модели вы поймете при создании своего приложения, особенно когда дело дойдет до привязки моделей к представлению.

Проверка

Проверка осуществляется наиболее простым методом: путем замены функции `validate()`, имеющейся в экземплярах модели. Функция `validate()` вызывается при сохранении записи. Если функция `validate()` что-нибудь возвращает, проверка завершается неудачно. В противном случае сохранение продолжается беспрепятственно, оставляя запись в локальной памяти:

```
Task.include({
  validate: function(){
    if ( !this.name ) return "Нужно указать имя";
  }
});
```

Если проверка проваливается, вы можете вернуть из функции `validate()` строку, содержащую пояснение. Это сообщение нужно использовать для уведомления пользователя о неправильном развитии событий и способах исправления ситуации:

```
Task.bind("error", function(record, msg){
  // Слишком общее уведомление об ошибке
  alert("Задание не сохранено: " + msg);
});
```

Событие модели `error` будет инициировано при провале проверки. Функциям обратного вызова будут переданы неверная запись и сообщение об ошибке.

Сохранение состояния

Записи библиотеки Spine всегда остаются в памяти, но у вас есть выбор скрытого от вас хранилища: например, можно выбрать принадлежащее HTML5 локальное хранилище (Local Storage) или же выбрать Ajax.

Использование локального хранилища организуется довольно легко. Нужно просто включить JavaScript-файл **spine.model.local.js** и расширить вашу модель модулем `Spine.Model.Local`:

```
// Сохранение с помощью локального хранилища
Task.extend(Spine.Model.Local);
Task.fetch();
```

Записи не будут автоматически извлекаться из локального хранилища, поэтому вам нужно вызвать функцию `fetch()`, чтобы наполнить свою модель уже существующими данными. Обычно это делается после всех остальных действий по инициализации вашего приложения. Как только модель будет заполнена новыми данными, будет инициировано событие `refresh`:

```
Task.bind("refresh", function(){
  // Новые задания!
  renderTemplate(Task.all());
});
```

Использование сохранности состояния с помощью Ajax организуется похожим образом, для этого нужно лишь включить сценарий **spine.model.ajax.js** и расширить вашу модель модулем **Spine.Model.Ajax**:

```
// Сохранение на сервере
Task.extend(Spine.Model.Ajax);
```

По умолчанию Spine определяет имя модели и использует для генерирования URL некоторые базовые правила создания множественной формы существительного. Поэтому для показанного выше примера URL-адресом модели `Task` будет **/tasks**. Это исходное поведение можно отменить, предоставив классу свое собственное свойство `URL`:

```
// Добавление своего собственного URL
Task.extend({
  url: "/tasks"
});
// Извлечение новых заданий (tasks) из сервера
Task.fetch();
```

Как только будет вызвана функция `Task.fetch()`, библиотека Spine сделает Ajax GET-запрос по адресу **/tasks**, ожидая ответ в формате JSON, содержащий массив заданий. Если сервер возвращает успешный ответ, записи будут загружены и будет инициировано событие `refresh`.

Spine будет отправлять Ajax-запросы на сервер при каждом создании, обновлении или удалении записи, синхронизируя две записи. Библиотека ожидает, что ваш сервер структурирован с использованием RESTful, поэтому она работает без сбоев, хотя вы можете явным образом отменить это поведение, чтобы оно удовлетворяло специализированным настройкам. Spine ожидает существования следующих оконечных точек:

```
read    → GET    /коллекция
create  → POST   /коллекция
update  → PUT    /коллекция/id
destroy → DELETE /коллекция/id
```

После создания записи на стороне клиента Spine отправит HTTP POST-запрос на ваш сервер, включая JSON-представление записи. Давайте создадим задание

по имени «Купить яйца». Тогда запрос, отправляемый на сервер, будет иметь следующий вид:

```
POST /tasks HTTP/1.0
Host: localhost:3000
Origin: http://localhost:3000
Content-Length: 66
Content-Type: application/json
{"id": "44E1DB33-2455-4728-AEA2-ECBD724B5E7B", "name": "Купить яйца"}
```

Соответственно, при удалении записи на сервер будет инициирован запрос DELETE, а обновление записи инициирует запрос PUT. Для запросов PUT и DELETE внутри URL дается ссылка на ID записей:

```
PUT /tasks/44E1DB33-2455-4728-AEA2-ECBD724B5E7B HTTP/1.0
Host: localhost:3000
Origin: http://localhost:3000
Content-Length: 71
Content-Type: application/json
{"id": "44E1DB33-2455-4728-AEA2-ECBD724B5E7B", "name": "Докупить яйца"}
```

Библиотека Spine синхронизирует Ajax не так, как большинство других библиотек. Она отправляет запрос серверу после того, как запись будет сохранена на стороне клиента, поэтому клиент никогда не ждет ответа. Это означает, что ваш клиент полностью обособлен от вашего сервера, т. е. ему для работы не нужно присутствие сервера.

Обособленность сервера предполагает три основных преимущества. Во-первых, это быстрый и незаблокированный интерфейс, следовательно, пользователи никогда не ждут возможности взаимодействия с вашим приложением. Во-вторых, это упрощение вашего кода — нет необходимости рассчитывать на то, что запись может быть показана в пользовательском интерфейсе, но не может быть отредактирована из-за ожидания ответа сервера. В-третьих, это существенно упрощает добавление поддержки работы в автономном режиме, если такая работа понадобится.

А как обстоят дела с проверкой на стороне сервера? Spine предполагает, что вся необходимая проверка будет производиться на стороне клиента. Единственный случай, когда сервер должен ответить с ошибкой, связан с выдачей исключения (вызванной проблемой с вашим кодом), что может произойти только в исключительных обстоятельствах.

Когда сервер возвращает неудачный ответ, в модели должно быть инициировано событие `ajaxError`, включающее запись, объект XMLHttpRequest, Ajax-установки, и выдается ошибка:

```
Task.bind("ajaxError", function(record, xhr, settings, error){
  // Неудачный_ответ
});
```

Контроллеры

Контроллеры являются последним компонентом Spine, предоставляющим средства для связывания в единое целое всех остальных составляющих вашего приложения. Контроллеры обычно добавляют к DOM-элементам и моделям обработчики событий, предоставляют шаблоны и поддерживают синхронизацию представления и моделей. Для создания Spine-контроллера вам нужно создать подкласс класса `Spine.Controller` путем вызова функции `create()`:

```
jQuery(function(){
  window.Tasks = Spine.Controller.create({
    // Свойства контроллера
  });
});
```

Контроллеры рекомендуется загружать только после того, как будут загружены все остальные компоненты страницы, чтобы вам не приходилось сталкиваться с разными состояниями страницы. Во всех примерах с использованием Spine вы заметите, что каждый контроллер помещен в вызов `jQuery()`. Тем самым гарантируется создание контроллера только после готовности документа.

В Spine действует соглашение давать контроллерам имена во множественном числе, записанные в смешанном регистре, используя обычно множественную форму имени модели, с которой они связаны. Большинство контроллеров имеют только свойства экземпляра, поскольку они используются только после создания этого экземпляра. Создание экземпляра контроллера ничем не отличается от создания экземпляра любого другого класса и осуществляется путем вызова функции `init()`:

```
var tasks = Tasks.init();
```

У контроллеров всегда есть связанный с ними DOM-элемент, доступ к которому можно получить через свойство `el`. Можно дополнительно пропустить его через создание экземпляра, в противном случае контроллер сгенерирует устанавливаемый по умолчанию `div`-элемент:

```
var tasks = Tasks.init({el: $("#tasks")});
assertEqual( tasks.el.attr("id"), "tasks" );
```

Этот элемент может использоваться внутри контроллера для добавления шаблонов и визуализации представлений:

```
window.Tasks = Spine.Controller.create({
  init: function(){
    this.el.html("Визуализируемый текст");
  }
});

var tasks = Tasks.init();
$("body").append(tasks.el);
```

В сущности, любые аргументы, переданные вами функции `init()`, будут установлены в качестве свойств контроллера. Например:

```
var tasks = Tasks.init({item: Task.first()});
assertEqual( Task.first(), tasks.item );
```

Использование представительства

В предыдущих примерах можно было заметить, что все функции обратного вызова, относящиеся к событиям, были заключены в функцию `this.proxy()`, чтобы гарантировать запуск всех этих функций в правильном контексте. Поскольку это весьма распространенная схема, Spine предоставляет сокращенную нотацию `proxied`. В ваш контроллер нужно добавить свойство `proxied`, содержащее массив имен функций, которые всегда должны выполняться в контексте контроллера:

```
// Эквивалент использования функции proxyAll
var Tasks = Spine.Controller.create({
  proxied: ["render", "addAll"],
  render: function(){ /* ... */ },
  addAll: function(){ /* ... */ }
});
```

Теперь можно передавать такие функции обратного вызова, как `render()`, слушателям событий, не беспокоясь о контексте выполнения. Такие функции всегда будут вызываться в правильном контексте.

Элементы

Зачастую к элементам внутри вашего контроллера удобно обращаться как к локальным свойствам. Spine предоставляет для этого краткую форму записи: `elements`. Нужно просто добавить к вашему контроллеру свойство `elements`, содержащее объект соотношения селекторов и имен. В показанном ниже примере `this.input` ссылается на элемент, выбранный по `form input[type=text]`. Все выборы осуществляются в контексте не всей страницы, а элемента контроллера (`el`):

```
// Переменная экземпляра 'input'
var Tasks = Spine.Controller.create({
  elements: {
    "form input[type=text]": "input"
  },

  init: function(){
    // this.input ссылается на поле ввода формы
    console.log( this.input.val() );
  }
});
```

Но при этом следует иметь в виду, что при замене HTML-кода элемента контроллера (`el`) вам нужно вызвать функцию `refreshElements()` для обновления всех ссылок на элемент.

Делегирование событий

Имеющееся в библиотеке Spine свойство `events` предоставляет вам легкий способ добавления сразу нескольких слушателей событий. Spine пользуется преимуществом всплытия события, поэтому к элементу контроллера (`el`) добавляется только один слушатель события. Как и свойство `events`, все делегирование события ограничивается областью видимости элемента `el`.

Свойство `events` принимает форму {"*селектор имениСобытия*": "*функция_обратного_вызова*"}. Селектор можно не указывать, и если он не предоставлен, событие будет относиться непосредственно к `el`. В противном случае событие будет *делегировано*, и оно будет инициировано, если событие данного типа состоится в отношении дочернего элемента, соответствующего селектору. Это происходит в динамическом режиме, поэтому изменение содержимого элемента `el` в данном случае никакой роли не играет:

```
var Tasks = Spine.Controller.create({
  events: {
    "keydown form input[type=text]": "keydown"
  },
  keydown: function(e){ /* ... */ }
});
```

В показанном выше примере как только поле ввода, соответствующее селектору, получит событие `keydown`, будет выполнена имеющаяся в контроллере функция обратного вызова `keydown`. В данном случае не нужно волноваться насчет представительства функций обратного вызова события.

Объект события передается функции обратного вызова, что в данном примере приносит ощутимую пользу, так как мы можем сказать, какая клавиша была нажата. Кроме этого, элемент, о котором идет речь, может быть извлечен из свойства события `target`.

События контроллера

Кроме делегирования событий, Spine-контроллеры поддерживают события, которые вы создаете самостоятельно. По умолчанию контроллеры расширяются модулем `Spine.Events`, стало быть, это влечет за собой наличие всех функциональных возможностей работы с событиями, включая функции `bind()` и `trigger()`. Этим можно воспользоваться, чтобы обеспечить обособленность ваших контроллеров друг от друга или в качестве части внутренней структуры контроллеров:

```
var Sidebar = Spine.Controller.create({
  events: {
    "click [data-name]": this.click
  },

  init: function(){
    this.bind("change", this.change);
  },
```

```
  change: function(name){ /* ... */ },

  click: function(e){
    this.trigger("change", $(e.target).attr("data-name"));
  }

  // ...
});
var sidebar = Sidebar.init({el: $("#sidebar")});
sidebar.bind("change", function(name){
  console.log("Sidebar changed:", name);
})
```

В показанном выше примере другие контроллеры могут привязаться к событию изменения боковой панели `Sidebar` или даже инициировать это событие. В главе 2 мы уже выяснили, что самостоятельно создаваемые события могут быть отличным способом внутреннего структурирования приложений, даже если они никогда не используются во внешней среде.

Глобальные события

Spine позволяет привязывать и инициировать события на глобальной основе. Это форма технологии PubSub, которая позволяет контроллерам общаться, даже не зная о существовании друг друга, гарантируя при этом должную обособленность. Подобный эффект достигается наличием глобального объекта `Spine.App`, к которому кто угодно может привязаться и инициировать в отношении него событие:

```
var Sidebar = Spine.Controller.create({
  proxied: ["change"],
  init: function(){
    this.App.bind("change", this.change);
  },
  change: function(name){ /* ... */ }
});
```

Spine-контроллеры допускают использование вместо `Spine.App` более краткую форму `this.App`, экономя количество набираемых символов. В показанном выше примере можно увидеть, что контроллер `Sidebar` привязан к глобальному событию `change`. Затем это событие с передачей нужных данных может быть инициировано другими контроллерами или сценариями:

```
Spine.App.trigger("change", "messages");
```

Схема визуализации

После рассмотрения всех основных возможностей, доступных в контроллерах, давайте посмотрим на некоторые случаи их типового применения.

Схема визуализации является весьма полезным способом связывания моделей и представлений. При создании экземпляра контроллера к соответствующей модели добавляется слушатель события, запускающий функцию обратного вызова

при обновлении или изменении модели. Функция обратного вызова будет обновлять el, как правило, путем замены его содержимого передаваемым шаблоном:

```
var Tasks = Spine.Controller.create({
  init: function(){
    Task.bind("refresh change", this.proxy(this.render));
  },

  template: function(items){
    return($("#tasksTemplate").tmpl(items));
  },

  render: function(){
    this.el.html(this.template(Task.all()));
  }
});
```

Этот простой, но грубоватый метод привязки данных обновляет каждый элемент при каждом изменении отдельной записи. Все это неплохо, когда дело касается простых и небольших по объему списков, но у вас может появиться потребность в получении более гибкого управления отдельными элементами, например добавления к элементам обработчиков событий. Здесь на первый план выступает *схема элемента*.

Схема элемента

Схема элемента дает вам практически те же функциональные возможности, что и схема визуализации, но с более гибким управлением. Эта схема состоит из двух контроллеров: одного, который управляет коллекцией элементов, и второго, который работает с каждым отдельным элементом. Чтобы получить хорошее представление о том, как все это работает, давайте углубимся непосредственно в код:

```
var TasksItem = Spine.Controller.create({
  // Делегирование события click локальному обработчику
  events: {
    "click": "click"
  },

  // Обеспечение функциям правильного контекста
  proxied: ["render", "remove"],

  // Привязка событий к записи
  init: function(){
    this.item.bind("update", this.render);
    this.item.bind("destroy", this.remove);
  },

  // Визуализация элемента
  render: function(item){
    if (item) this.item = item;
```

продолжение ⇨

```
    this.el.html(this.template(this.item));
    return this;
  },

  // Использование шаблона, в данном случае посредством jQuery.tmpl.js
  template: function(items){
    return($("#tasksTemplate").tmpl(items));
  },

  // Вызывается после удаления элемента
  remove: function(){
    this.el.remove();
  },

  // У нас есть неплохое управление событиями, а также
  // легкий доступ к записи
  click: function(){ /* ... */ }
});

var Tasks = Spine.Controller.create({
  proxied: ["addAll", "addOne"],

  init: function(){
    Task.bind("refresh", this.addAll);
    Task.bind("create",  this.addOne);
  },

  addOne: function(item){
    var task = TasksItem.init({item: item});
    this.el.append(task.render().el);
  },

  addAll: function(){
    Task.each(this.addOne);
  }
});
```

В показанном выше примере на контроллер `Tasks` возлагается обязанность добавления записей при их первоначальном создании, а на `TasksItem` возлагается ответственность за события записи `update` (обновление) и `destroy` (удаление), с визуализацией записи в случае необходимости. Хотя все это намного сложнее, но зато дает нам некоторые преимущества над предыдущей схемой визуализации.

Прежде всего, она эффективнее: список не нужно перерисовывать при изменениях одного элемента. Более того, у нас теперь есть намного больше возможностей управления отдельными элементами. Мы можем помещать обработчики событий, как показано на примере функции обратного вызова события `click`, и управлять отображением на поэлементной основе.

Создание программы управления контактами

Давайте применим наши знания API библиотеки Spine на практике и создадим что-нибудь полезное, например программу управления контактами. Мы хотим дать пользователям способ чтения, создания, обновления и удаления контактов, а также их поиска.

На рис. 11.1 показан конечный результат, чтобы у вас было представление о том, что мы создаем.

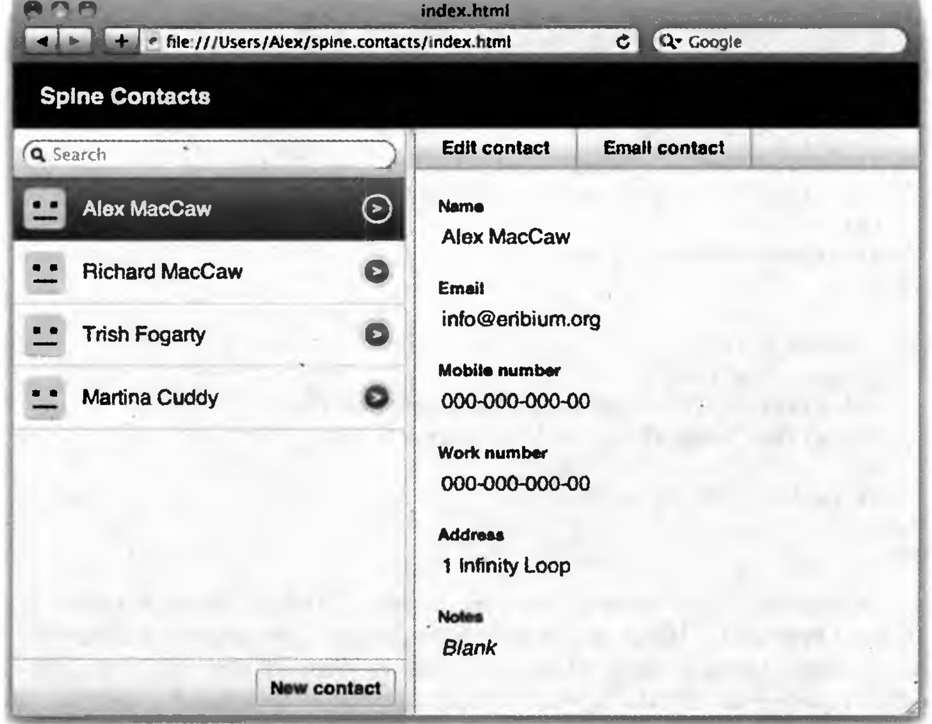

Рис. 11.1. Вывод списка контактов в Spine-приложении

Программа управления контактами — один из наборов Spine-примеров, имеющих открытый код. Вы можете последовать изложенному далее руководству или же загрузить полную версию кода из *репозитория проекта*.

На рис. 11.1 показано, что программа управления контактами имеет два основных раздела, боковую панель и панель просмотра контактов. Эти два раздела и определят наличие соответствующих контроллеров `Sidebar` и `Contacts`. Что же касается моделей, у программы она всего одна: модель `Contact`. Перед тем как рассматри-

вать каждый отдельный компонент, давайте взглянем на структуру исходной страницы:

```
<div id="sidebar">
  <ul class="items">
  </ul>
  <footer>
    <button>New contact</button>
  </footer>
</div>

<div class="vdivide"></div>

<div id="contacts">
  <div class="show">
    <ul class="options">
      <li class="optEdit">Edit contact</li>
      <li class="optEmail">Email contact</li>
    </ul>
    <div class="content"></div>
  </div>

  <div class="edit">
    <ul class="options">
      <li class="optSave default">Save contact</li>
      <li class="optDestroy">Delete contact</li>
    </ul>
    <div class="content"></div>
  </div>
</div>
```

Для каждого раздела у нас есть, соответственно, div-контейнер #sidebar и div-контейнер #contacts. Наше приложение занимается заполнением именами контактов списка .items, а также имеет текущий выбранный контакт, отображаемый в контейнере #contacts. Мы будем слушать щелчки на .optEmail и .optSave, переключая в соответствии с требованиями состояния показа и редактирования. И наконец, мы будем слушать события щелчка на .optDestroy, которое удаляет текущий контакт и выбирает другой контакт.

Модель Contact

Имея всего лишь полдюжины строк кода, модель контактов чрезвычайно проста. У нее имеются три атрибута: first_name, last_name и email. Мы также предоставим вспомогательную функцию, которая будет давать полное имя и пригодится нам в шаблонах:

```
// Создание модели
var Contact = Spine.Model.setup("Contact",
              ["first_name", "last_name", "email"]);
```

```
// Сохранение модели между перезагрузками страницы
Contact.extend(Spine.Model.Local);

// Добавление функций экземпляра
Contact.include({
  fullName: function(){
    if ( !this.first_name && !this.last_name ) return;
    return(this.first_name + " " + this.last_name);
  }
});
```

Обратите внимание, что модель расширена за счет модуля `Spine.Model.Local`. Это обеспечит сохранение записей в локальном хранилище браузера, открывая к ним доступ при следующей загрузке страницы.

Контроллер Sidebar

Теперь посмотрим на контроллер `Sidebar`, который отвечает за вывод списка контактов и за отслеживание текущего выбранного контакта. При изменении контактов контроллер `Sidebar` должен обновиться, чтобы отражать эти изменения. Кроме того, на боковой панели имеется кнопка New contact (Новый контакт), которую будет слушать контроллер, создавая по щелчку на ней новые незаполненные контакты.

А вот как выглядит полный контроллер во всей своей красе. Поначалу это может быть слишком трудным для восприятия фрагментом кода, особенно если вы не знакомы с библиотекой Spine, но он содержит множество комментариев, и при более пристальном изучении в нем вполне можно разобраться:

```
jQuery(function($){

  window.Sidebar = Spine.Controller.create({
    // Создание переменных экземпляра:
    //  this.items //=> <ul></ul>
    elements: {
      ".items": "items"
    },

    // Присоединение делегирования событий
    events: {
      "click button": "create"
    },

    // Обеспечение вызова данных функций с текущей областью видимости,
    // поскольку они используются в качестве принадлежащих событиям
    // функций обратного вызова
    proxied: ["render"],

    // Визуализация шаблона
    template: function(items){
      return($("#contactsTemplate").tmpl(items));
    },
```

продолжение ⇨

```
    init: function(){
      this.list = Spine.List.init({
        el: this.items,
        template: this.template
      });

      // Показ контакта при изменении текущего элемента списка
      this.list.bind("change", this.proxy(function(item){
        this.App.trigger("show:contact", item);
      }));

      // Изменение текущего выбранного элемента списка при изменении
      // текущего контакта, т. е. при создании нового контакта
      this.App.bind("show:contact edit:contact", this.list.change);

      // Визуализация при заполнении или изменении контактов
      Contact.bind("refresh change", this.render);
    },

    render: function(){
      var items = Contact.all();
      this.list.render(items);
    },

    // Вызывается по щелчку на кнопке создания 'Create'
    create: function(){
      var item = Contact.create();
      this.App.trigger("edit:contact", item);
    }
  });

});
```

Обратите внимание, что функция контроллера `init()` использует класс под названием `Spine.List`, который мы еще не рассматривали. `Spine.List` является сервисным контроллером, применяемым для создания списков записей. К тому же `Spine.List` будет отслеживать текущий выбранный элемент, а затем уведомлять слушателей с помощью события `change`, когда пользователь выбирает другой элемент.

При каждом изменении или обновлении контактов на экран заново выводится весь список. Это придает примеру простоту и привлекательность, но в будущем, если возникнут проблемы производительности, у нас, возможно, появится желание внести в него изменения.

Ссылка `#contactsTemplate`, имеющаяся в функции `template()`, является script-элементом, который содержит наш шаблон контактов для отдельных элементов списка:

```
<script type="text/x-jquery-tmpl" id="contactsTemplate">
  <li class="item">
```

```
    {{if fullName()}}
      <span>${fullName()}</span>
    {{else}}
      <span>No Name</span>
    {{/if}}
  </li>
</script>
```

Для создания шаблонов мы используем библиотеку jQuery.tmpl, которая должна быть вам знакома после прочтения главы 5. `Spine.List` будет использовать этот шаблон для визуализации каждого элемента, и он будет устанавливать значение `current` для class-атрибута тега `<li>`, если этот тег связан с текущим выбранным элементом.

Контроллер Contacts

Теперь наш контроллер `Sidebar` показывает список контактов, позволяя пользователям выбирать отдельные контакты. А как обстоят дела с показом текущего выбранного контакта? Здесь на первый план выходит контроллер `Contacts`:

```
jQuery(function($){

  window.Contacts = Spine.Controller.create({
    // Заполнение свойств внутреннего элемента
    elements: {
      ".show": "showEl",
      ".show .content": "showContent",
      ".edit": "editEl"
    },

    proxied: ["render", "show"],

    init: function(){
      // Показ контакта с помощью исходного представления
      this.show();

      // Визуализация представления при изменении контакта
      Contact.bind("change", this.render);

      // Привязка к глобальным событиям
      this.App.bind("show:contact", this.show);
    },

    change: function(item){
      this.current = item;
      this.render();
    },

    render: function(){
```

продолжение ⇗

```
    this.showContent.html($("#contactTemplate").tmpl(this.current));
  },

  show: function(item){
    if (item && item.model) this.change(item);

    this.showEl.show();
    this.editEl.hide();
  }
});
```

Как только в боковой панели будет выбран новый контакт, будет инициировано глобальное событие `show:contact`. Мы привязываемся к этому событию в `Contacts`, выполняя функцию `show()`, которой передается только что выбранный контакт. Затем мы заново выводим на экран `div`-контейнер `showContent`, заменяя прежнее содержимое текущей выбранной записью.

Мы сослались на шаблон `#contactTemplate`, который выведет текущий контакт контроллера `Contacts` нашим пользователям. Давайте продолжим движение и добавим этот шаблон к странице:

```
<script type="text/x-jquery-tmpl" id="contactTemplate">
  <label>
    <span>Name</span>
    ${first_name} ${last_name}
  </label>

  <label>
    <span>Email</span>
    {{if email}}
      ${email}
    {{else}}
      <div class="empty">Blank</div>
    {{/if}}
  </label>
</script>
```

Теперь у нас появились функциональные возможности для показа контактов, а как обстоят дела с их редактированием и удалением? Давайте перепишем контроллер `Contacts`, чтобы он мог осуществлять и эти действия. Основное отличие заключается в том, что мы собираемся переключаться между двумя состояниями приложения, показывая и редактируя контакт при щелчках на элементах `.optEdit` и `.optSave`. Мы также собираемся ввести в дело новый шаблон: `#editContactTemplate`. При сохранении записей мы будем считывать и редактировать поля ввода формы и обновлять атрибуты записей:

```
jQuery(function($){

  window.Contacts = Spine.Controller.create({
    // Заполнение свойств внутреннего элемента
```

```
elements: {
  ".show": "showEl",
  ".edit": "editEl",
  ".show .content": "showContent",
  ".edit .content": "editContent"
},

// Делегирование событий
events: {
  "click .optEdit": "edit",
  "click .optDestroy": "destroy",
  "click .optSave": "save"
},

proxied: ["render", "show", "edit"],

init: function(){
  this.show();
  Contact.bind("change", this.render);
  this.App.bind("show:contact", this.show);
  this.App.bind("edit:contact", this.edit);
},

change: function(item){
  this.current = item;
  this.render();
},

render: function(){
  this.showContent.html($("#contactTemplate").tmpl(this.current));
  this.editContent.html($("#editContactTemplate").tmpl(this.current));
},

show: function(item){
  if (item && item.model) this.change(item);

  this.showEl.show();
  this.editEl.hide();
},

// Вызывается по щелчку на кнопке редактирования 'edit'
edit: function(item){
  if (item && item.model) this.change(item);

  this.showEl.hide();
  this.editEl.show();
},

// Вызывается по щелчку на кнопке удаления 'delete'
```

продолжение ⇨

```
    destroy: function(){
      this.current.destroy();
    },

    // Вызывается по щелчку на кнопке сохранения 'save'
    save: function(){
      var atts = this.editEl.serializeForm();
      this.current.updateAttributes(atts);
      this.show();
    }
  });

});
```

Как уже ранее упоминалось, мы используем новый шаблон по имени `#editContactTemplate`. Нам нужно добавить его к странице, чтобы на него можно было успешно ссылаться. По сути, `#editContactTemplate` очень похож на `#contactTemplate`, за исключением того, что в нем для отображения данных записей используются элементы ввода:

```
<script type="text/x-jquery-tmpl" id="editContactTemplate">
  <label>
    <span>First name</span>
    <input type="text" name="first_name" value="${first_name}" autofocus>
  </label>

  <label>
    <span>Last name</span>
    <input type="text" name="last_name" value="${last_name}">
  </label>

  <label>
    <span>Email</span>
    <input type="text" name="email" value="${email}">
  </label>
</script>
```

Контроллер App

Итак, у нас уже есть два контроллера — `Sidebar` и `Contacts`, — которые имеют дело с выбором, отображением и редактированием записей модели `Contact`. Теперь нам осталось лишь получить контроллер `App`, который создает экземпляры любых других контроллеров, передавая им требуемые элементы страницы:

```
jQuery(function($){
  window.App = Spine.Controller.create({
    el: $("body"),

    elements: {
      "#sidebar": "sidebarEl",
```

```
        "#contacts": "contactsEl"
      },

      init: function(){
        this.sidebar = Sidebar.init({el: this.sidebarEl});
        this.contact = Contacts.init({el: this.contactsEl});
        // Извлечение контактов из локального хранилища
        Contact.fetch();
      }
    }).init();
  });
```

Обратите внимание, что мы вызываем `.init()` сразу же после создания контроллера `App`. Мы также вызываем функцию `fetch()` в отношении модели `Contact`, извлекая все контакты из локального хранилища.

Ну вот, собственно, и все! Два главных контроллера (`Sidebar` и `Contacts`), одна модель (`Contact`) и пара представлений. Чтобы увидеть конечный продукт, проверьте репозиторий исходного кода и посмотрите на рис. 11.2.

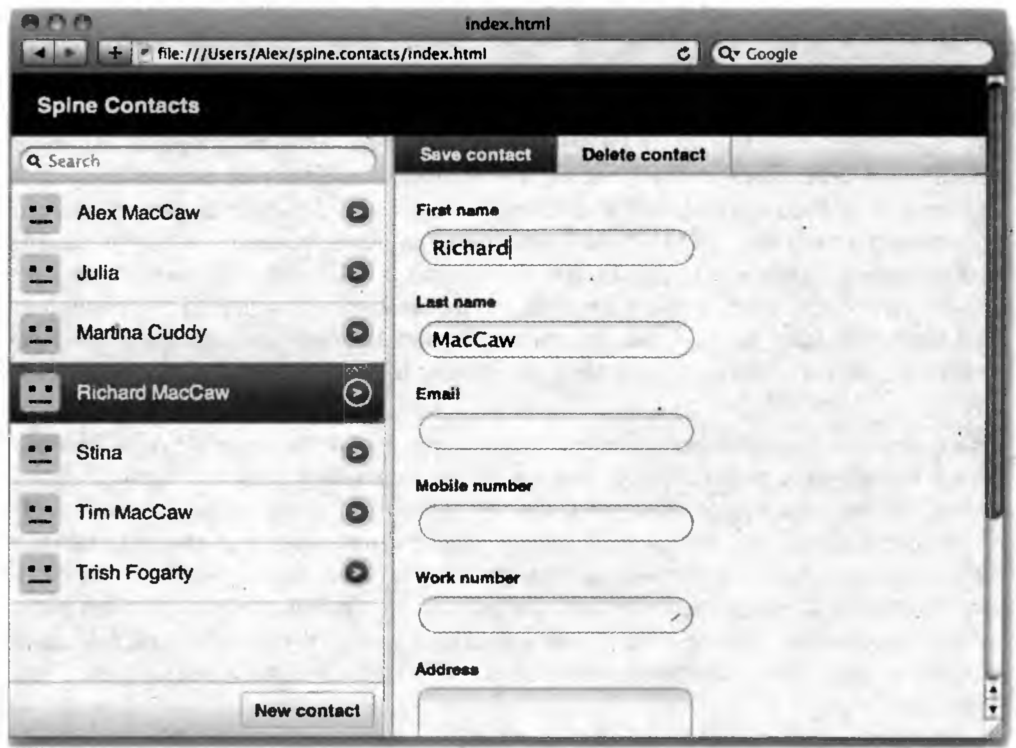

Рис. 11.2. Редактирование контактов в примере Spine-приложения

2 Библиотека Backbone

Backbone — великолепная библиотека для создания JavaScript-приложений. Вся ее прелесть заключается в простоте. При всей своей легкости библиотека удовлетворяет всем основным потребностям и является при этом весьма гибким средством. Как и во всей остальной книге, основное внимание здесь уделяется MVC, и именно эта схема положена в основу Backbone. Эта библиотека предоставляет вам модели, контроллеры и представления, являющиеся строительными блоками вашего приложения.

Чем же Backbone отличается от других сред разработки, например от SproutCore или Cappuccino? Основное отличие заключается в миниатюрной природе Backbone. SproutCore и Cappuccino предоставляют богатый набор виджетов пользовательского интерфейса и множество корневых библиотек, определяя для вас структуру вашего HTML. Оба этих средства после запаковки и gzip-сжатия насчитывают по объему сотни килобайт, и нужно прибавить еще несколько мегабайт JavaScript, CSS и изображений, загружаемых в браузер. Для сравнения, Backbone занимает всего 4 Кб, предоставляя исключительно корневые концепции моделей, событий, коллекций, представлений, контроллеров и сохраняемости.

Единственной сильной зависимостью Backbone является underscore.js, библиотека, наполненная полезными утилитами и функциями JavaScript общего назначения. Underscore предоставляет более 60 функций, работающих, кроме всего прочего, с массивами с привязкой функций, с применением шаблонов JavaScript и с широко толкуемым тестированием равенства. Вам определенно стоит ознакомиться с API, предоставляемым Underscore, особенно если вы плотно работаете с массивами. Кроме Underscore в качестве средств, помогающих Backbone с реализацией представлений, вы можете свободно использовать jQuery или Zepto.js.

Несмотря на то что Backbone имеет неплохую документацию, при первом знакомстве с ней все же могут возникнуть небольшие затруднения. Цель данной главы — исправить эту ситуацию, предоставив углубленное практическое введение в библиотеку. Первые несколько разделов представляют собой обзор компонентов

Backbone, а завершится глава практическим приложением. Если хотите увидеть Backbone в действии, можете сразу перейти к концу главы.

Модели

Начнем, пожалуй, с самого основного компонента MVC: моделей. Именно в моделях хранятся данные вашего приложения. Модели нужно воспринимать как воображаемую абстракцию необработанных данных приложения с добавлением полезных функций и событий. Backbone-модели можно создать путем вызова функции `extend()` в отношении объекта `Backbone.Model`:

```
var User = Backbone.Model.extend({
  initialize: function() {
    // ...
  }
});
```

Первый аргумент, передаваемый `extend()`, представляет собой объект, который становится свойством экземпляра модели. Второй аргумент является необязательным хэшем свойств класса. Для создания подклассов моделей, наследующих все ее родительские свойства класса и экземпляра, функцию `extend()` можно вызывать несколько раз:

```
var User = Backbone.Model.extend({
  // Свойства экземпляра
  instanceProperty: "foo"
}, {
  // Свойства класса
  classProperty: "bar"
});

assertEqual( User.instanceProperty, "foo" );
assertEqual( User.prototype.classProperty, "bar" );
```

При создании экземпляра модели вызывается принадлежащая ей функция экземпляра `initialize()`, которой передаются все аргументы, используемые при создании этого экземпляра. Закулисно Backbone-модели являются функциями-конструкторами, поэтому вы можете создать новый экземпляр путем использования ключевого слова `new`:

```
var User = Backbone.Model.extend({
  initialize: function(name) {
    this.set({name: name});
  }
});

var user = new User("Leo McGarry");
assertEqual( user.get("name"), "Leo McGarry");
```

Модели и атрибуты

Для установки и получения атрибутов экземпляра следует использовать функции `set()` и `get()`:

```
var user = new User();
user.set({name: "Donna Moss"})

assertEqual( user.get("name"), "Donna Moss" );
assertEqual( user.attributes, {name: "Donna Moss"} );
```

Функции `set(`*`attrs`*`, [`*`options`*`])` передается хэш атрибутов, применяемых к экземпляру, а функции `get(`*`attr`*`)` передается один строковый аргумент — имя атрибута, а она возвращает значение этого атрибута. Экземпляр отслеживает свои текущие атрибуты с помощью локальной хэш-структуры под названием `attributes`. Вы можете работать с ней напрямую, так же как с функциями `get()` и `set()`, обеспечивая соответствующие проверки приемлемости и инициирование событий.

Проверку приемлемости атрибутов экземпляра можно осуществить с помощью функции `validate()`. Изначально эта функция остается неопределенной, но вы можете подменить ее, добавив любую нужную вам логику проверки:

```
var User = Backbone.Model.extend({
  validate: function(atts){
    if (!atts.email || atts.email.length < 3) {
      return "Адрес электронной почты должен иметь не менее 3 символов";
    }
  }
});
```

Если модель и атрибуты соответствуют требованиям, возвращать что-нибудь из `validate()` не нужно, а если атрибуты не соответствуют требованиям, вы можете либо вернуть строку с описанием ошибки, либо экземпляр класса `Error`. Если проверка приемлемости завершается неудачно, функции `set()` и `save()` не будут продолжать свое выполнение и будет инициировано событие `error`. Вы можете привязаться к событию `error`, обеспечивая себе уведомление при неудачном завершении любой проверки:

```
var user = new User;

user.bind("error", function(model, error) {
  // Обработка ошибки
});

user.set({email: "ga"});

// Или же добавление обработчика ошибки в определенный набор
user.set({"email": "ga"}, {error: function(model, error){
  // ...
}});
```

Атрибуты по умолчанию следует указывать с помощью хэша default. При создании экземпляра модели для любых неуказанных атрибутов будут установлены их значения по умолчанию:

```
var Chat = Backbone.Model.extend({
  defaults: {
    from: "anonymous"
  }
});

assertEqual( (new Chat).get("from"), "anonymous" );
```

Коллекции

В Backbone массивы экземпляров моделей хранятся в коллекциях. Возможно, сразу будет не совсем понятно, зачем нужно отделять коллекции от моделей, но, как ни странно, это вполне устоявшаяся практика. Если, к примеру, заново создавать Twitter, у вас будут две коллекции: сторонники — Followers и наблюдаемые — Followees, и обе они заполнены экземплярами класса User. Хотя обе коллекции заполнены одной и той же моделью, в каждой из них содержится массив различных экземпляров модели User, в результате они являются отдельными коллекциями.

Как и в случае с моделями, коллекцию можно создать путем расширения Backbone.Collection:

```
var Users = Backbone.Collection.extend({
  model: User
});
```

В показанном выше примере вы можете увидеть, что мы переписали свойство model, чтобы указать, какую именно модель мы хотим связать с коллекцией — в данном случае это модель User. Хотя это и не является абсолютным требованием, эта установка будет полезна тем, что она дает коллекции модель по умолчанию для ссылок на нее, если это когда-нибудь потребуется. Обычно коллекция содержит экземпляры только одного типа модели, а не нескольких разных типов.

При создании коллекции вы можете дополнительно передать исходный массив моделей. Как и при работе с Backbone-моделями, если определена функция инициализации экземпляра, она будет вызвана при его создании:

```
var users = new Users([{name: "Toby Ziegler"}, {name: "Josh Lyman"}]);
```

Вместо этого вы можете добавлять модели к коллекции, используя функцию add():

```
var users = new Users;

// Добавление отдельной модели
users.add({name: "Donna Moss"});
```

продолжение ⇨

```
// Или добавление массива моделей
users.add([{name: "Josiah Bartlet"}, {name: "Charlie Young"}]);
```

При добавлении модели к коллекции инициируется событие add:

```
users.bind("add", function(user) {
  alert("Ahoy " + user.get("name") + "!");
});
```

Можно также удалить модель из коллекции, используя функцию remove(), которая инициирует событие remove:

```
users.bind("remove", function(user) {
  alert("Прощай " + user.get("name") + "!");
});

users.remove( users.models[0] );
```

Извлечь конкретную модель совсем не трудно. Если есть ID модели, можно воспользоваться функцией контроллера get():

```
var user = users.get("some-guid");
```

Если ID модели отсутствует, можно извлечь модель по *cid* — идентификатору клиента (client ID), созданного Backbone автоматически при создании новой модели:

```
var user = users.getByCid("некий_cid");
```

В дополнение к событиям add и remove при изменении модели, находящейся в коллекции, будет инициировано событие change:

```
var user = new User({name: "Adam Buxton"});

var users = new Backbone.Collection;
users.bind("change", function(rec){
  // Запись была изменена!
});
users.add(user);

user.set({name: "Joe Cornish"});
```

Управление порядком следования экземпляров моделей в коллекции

Предоставив функцию comparator(), возвращающую значение, на основе которого нужно отсортировать коллекцию, вы можете управлять порядком следования экземпляров моделей в коллекции:

```
var Users = Backbone.Collection.extend({
  comparator: function(user){
    return user.get("name");
  }
});
```

Можно вернуть либо строку, либо числовое значение, на основе которого нужно проводить сортировку (в отличие от обычной сортировки в JavaScript). В показанном выше примере мы обеспечили сортировку коллекции `Users` в алфавитном порядке имен. Выстраивание в указанном порядке будет осуществляться автоматически, но если когда-нибудь понадобится пересортировать коллекцию в принудительном порядке, можно вызвать функцию `sort()`.

Представления

Сами по себе Backbone-представления с шаблонами не работают, но они управляют классами, работающими с представлением данных модели. Это может сбить с толку, поскольку многие MVC-реализации ссылаются на представления, как на блоки кода HTML или шаблоны, работающие с событиями и визуализацией в контроллерах. Несмотря на это, в Backbone этот код считается представлением, «потому что он исполняет роль логической части пользовательского интерфейса, ответственного за содержимое единой DOM-модели».

Подобно моделям и коллекциям, представления создаются путем расширения одного из существующих в Backbone классов, в данном случае это класс `Backbone.View`:

```
var UserView = Backbone.View.extend({
  initialize: function(){ /* ... */ },
  render: function(){ /* ... */ }
});
```

Каждый экземпляр представления построен на идее текущего DOM-элемента, или `this.el`, независимо от того, было ли представление вставлено в страницу. Элемент `el` создается с использованием атрибутов из свойств представления `tagName`, `className` или `id`. Если ни одно из этих свойств не задано, `el` представляет собой пустой `div`-контейнер:

```
var UserView = Backbone.View.extend({
  tagName: "span",
  className: "users"
});

assertEqual( (new UserView).el.className, "users" );
```

Если нужно привязать представление к уже существующему на странице элементу, нужно просто установить `el` напрямую. Разумеется, нужно сделать так, чтобы это представление устанавливалось уже после загрузки страницы, иначе элемент еще не будет существовать:

```
var UserView = Backbone.View.extend({
  el: $(".users")
});
```

Можно также передать `el` в качестве дополнительного аргумента при создании экземпляра представления, воспользовавшись для этого свойствами `tagName`, `className` и `id`:

```
new UserView({id: "followers"});
```

Визуализация представлений

У каждого представления имеется также функция `render()`, которая изначально имеет форму no-op (пустой функции). Ваше представление может вызвать эту функцию при необходимости перерисовки представления. Вы можете переписать эту функцию под работу с вашим представлением, чтобы она занималась визуализацией шаблонов и наполнением `el` новым кодом HTML:

```
var TodoView = Backbone.View.extend({
  template: _.template($("#todo-template").html()),

  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Библиотеке Backbone совершенно все равно, как вы визуализируете представления. Вы можете создавать элементы сами или же использовать библиотеку для работы с шаблонами. Но я рекомендую последнее, потому что в целом правильнее всего держать HTML вне своих JavaScript-программ. Поскольку Underscore.js, как библиотека, от которой зависит Backbone, находится на странице, можно воспользоваться `_.template()`, весьма полезным средством создания шаблонов.

В показанном выше примере вы, наверное, обратили внимание, что мы используем локальное свойство под названием `this.model`. Оно фактически указывает на экземпляр модели и попадает к представлению при создании экземпляра этого представления:

```
new TodoView({model: new Todo});
```

Имеющаяся в модели функция `toJSON()`, по сути, возвращает необработанные атрибуты модели, готовые к использованию шаблоном.

Делегирование событий

Посредством делегирования Backbone-представления предоставляют простое сокращение для добавления в `el` обработчиков событий. Установить в представление хэш событий и соответствующие функции обратного вызова можно следующим образом:

```
var TodoView = Backbone.View.extend({
  events: {
    "change input[type=checkbox]" : "toggleDone",
    "click .destroy"              : "clear",
```

```
  },

  toggleDone: function(e){ /* ... */},
  clear: function(e){ /* ... */}
});
```

Хэш событий имеет формат `{"`*типСобытия селектор*`": "`*функция_обратного_вызова*`"}`. Селектор указывать необязательно, и если он не предоставлен, событие привязывается непосредственно к `el`. Если селектор предоставлен, событие делегируется, что, по сути, означает его динамическую привязку к любому имеющемуся в `el` дочернему элементу, который соответствует селектору. Делегирование использует всплытие событий, следовательно, события будут по-прежнему инициироваться, несмотря на изменения содержимого элемента `el`.

Функция обратного вызова указывается в виде строки, которая ссылается на имя функции экземпляра текущего представления. Когда инициируются события представления, связанные с функциями обратного вызова, эти функции вызываются в контексте текущего представления, а не текущего контекста цели события или окна. Это весьма полезное свойство, поскольку из любой функции обратного вызова вы имеете прямой доступ к `this.el` и `this.model`, как в представленных выше примерах функций `toggleDone()` и `clear()`.

Привязка и контекст

А как же вызывается имеющаяся в представлении функция `render()`? Обычно она вызывается моделью, с которой связано представление в том случае, если она изменяется, для чего используется событие `change`. Следовательно, представления и HTML вашего приложения поддерживаются в соответствии (в привязке) с данными вашей модели:

```
var TodoView = Backbone.View.extend({
  initialize: function() {
    _.bindAll(this, 'render', 'close');
    this.model.bind('change', this.render);
  },

  close: function(){ /* ... */ }
});
```

В функциях обратного вызова, связанных с событиями, нужно остерегаться изменения контекста. Библиотека Underscore предоставляет полезную функцию обхода данной проблемы: `_.bindAll(`*контекст*`, *`*именаФункций*`)`. Эта функция связывает вместе контекст и имена функций (в виде строк). Функция `_.bindAll()` гарантирует, что все указанные вами функции всегда вызываются в заданном контексте. Этот особенно полезно для функций обратного вызова, связанных с событиями, поскольку их контекст всегда изменяется. В показанном выше примере функции `render()` и `close()` будут всегда вызываться в контексте экземпляра класса `TodoView`.

Приспособление к деструктивным операциям в модели осуществляется аналогичным образом. Ваши представления нужно привязать к событию модели `delete`, удаляя `el` при инициировании этого события:

```
var TodoView = Backbone.View.extend({
  initialize: function() {
    _.bindAll(this, 'render', 'remove');
    this.model.bind('change',  this.render);
    this.model.bind('delete', this.remove);
  },

  remove: function(){
    $(this.el).remove();
  }
});
```

Следует заметить, что Backbone-представления можно визуализировать и без использования моделей и даже функций обратного вызова, связанных с событиями. Вы можете просто вызвать функцию `render()` из `initialize()`, визуализируя представление при первом создании его экземпляра. Но я рассмотрел объединение модели и представления, потому что представления обычно используются именно в этой связке, которая является одним из наиболее полезных и эффективных свойств Backbone.

Контроллеры

Backbone-контроллеры подключают состояние приложения к фрагменту URL, который следует за символом решетки, предоставляя адреса, которыми можно обмениваться и помещать в закладки. По сути дела, контроллеры состоят из набора маршрутов и функций, которые будут вызываться при переходах по этим маршрутам.

Маршруты представляют из себя хэш: ключ состоит из путей, параметров и сплатов (splats), а значение указывает на функцию, связанную с маршрутом:

```
routes: {                                  // Соответствует:
  "help":                   "help",      //  #help
  "search/:query":          "search",    //  #search/kiwis
  "search/:query/p:page": "search"     //  #search/kiwis/p7
  "file/*path":             "file"       //  #file/любые_символы/path.txt
}
```

В показанном выше примере можно увидеть, что параметры начинаются с символа :, за которым следует имя параметра. Любые параметры, имеющиеся в маршруте, будут переданы его действию при вызове этого маршрута. Сплаты, указанные с помощью символа *, по сути, являются символами-заместителями, которые соответствуют чему угодно. Как и параметры, сплаты будут переданы в виде соответствующих значений в действие вашего маршрута.

Маршруты анализируются в порядке, обратном тому, в котором они указаны в хэше. Иными словами, ваши наиболее универсальные маршруты, «отлавливающие абсолютно все», будут находиться в конце хэша маршрутов.

Обычно контроллеры создаются путем расширения Backbone.Controllers, с передачей объекта, содержащего свойства экземпляра:

```
var PageController = Backbone.Controller.extend({
  routes: {
    "":                       "index",
    "help":                   "help",    // #help
    "search/:query":          "search",  // #search/kiwis
    "search/:query/p:page": "search"   // #search/kiwis/p7
  },

  index: function(){ /* ... */ },

  help: function() {
    // ...
  },

  search: function(query, page) {
    // ...
  }
});
```

В показанном выше примере, когда пользователь переходит на http://example.com#search/coconut, либо самостоятельно, либо путем щелчка на кнопке возврата к предыдущей странице, вызывается функция search() с переменной query, указывающей на "coconut".

Если нужно сделать приложение совместимым со спецификацией Ajax Crawling и индексируемым поисковыми движками (как рассматривалось в главе 4), нужно поставить перед всеми маршрутами префикс !/, как в следующем примере:

```
var PageController = Backbone.Controller.extend({
  routes: {
    "!/page/:title": "page", // #!/page/foo-title
  }
  // ...
}):
```

Также нужно внести изменения на серверной стороне, как описано в спецификации.

Если нужны более широкие функциональные возможности маршрутизации, например обеспечение принадлежности конкретных параметров в целым числам, можно передать непосредственно функции route() регулярное выражение:

```
var PageController = Backbone.Controller.extend({
  initialize: function(){
    this.route(/pages\/(\d+)/, 'id', function(pageId){
      // ...
    });
  }
}):
```

Итак, маршруты связывают изменения URL-фрагментов с контроллерами, но как насчет установки фрагмента на первое место? Вместо установки `window.location.hash` вручную, Backbone предоставляет более удобный способ — функцию `saveLocation(fragment)`:

```
Backbone.history.saveLocation("/page/" + this.model.id);
```

При вызове функции `saveLocation()` и обновлении URL-фрагмента никакие маршруты контроллера не вызываются. Это означает, что вы можете совершенно свободно вызывать `saveLocation()` в функции `initialize()` представления, к примеру, без какого-либо вмешательства контроллера.

Согласно своему внутреннему устройству Backbone будет прислушиваться к событию `onhashchange` в тех браузерах, которые его поддерживают, или реализует обходной вариант, используя `iframe`-теги и таймеры. Но с помощью следующего вызова нужно будет инициировать поддержку истории Backbone:

```
Backbone.history.start();
```

Запуск истории Backbone нужно осуществлять только после загрузки страницы, когда станут доступны все ваши представления, модели и коллекции. Backbone не поддерживает новый исторический API HTML5, использующий функции `pushState()` и `replaceState()`. Причина в том, что в настоящее время эти функции нуждаются в специальной обработке на серверной стороне, и пока что они не поддерживаются в Internet Explorer. Соответствующая поддержка может быть добавлена в Backbone, как только будут приняты соответствующие меры. А пока вся маршрутизация осуществляется с помощью хэш-фрагмента URL.

Синхронизация с сервером

По умолчанию при сохранении модели Backbone уведомит ваш сервер с помощью Ajax-запроса, используя одну из библиотек: либо jQuery, либо Zepto.js. Backbone делает это путем вызова метода `Backbone.sync()` перед созданием, обновлением или удалением модели. Затем Backbone отправит RESTful JSON-запрос на ваш сервер, который, в случае успеха, обновит модель на стороне клиента.

Чтобы воспользоваться этим, нужно определить в вашей модели свойство экземпляра `url` и располагать REST-совместимым сервером. Обо всем остальном позаботится Backbone:

```
var User = Backbone.Model.extend({
  url: '/users'
});
```

Свойство `url` может быть либо строкой, либо функцией, возвращающей строку. Путь может быть относительным или абсолютным, но он должен возвращать конечную точку модели.

Backbone отображает действия создания, чтения, обновления и удаления (CRUD) на следующие методы:

```
create → POST   /коллекция
read   → GET    /коллекция[/id]
update → PUT    /коллекция/id
delete → DELETE /коллекция/id
```

Например, при создании экземпляра класса `User` Backbone отправит POST-запрос к /users. Аналогично, обновление экземпляра `User` отправит PUT-запрос на конечную точку /users/id, где id — идентификатор модели. В ответ на POST-, PUT- и GET-запросы Backbone ожидает от вас возвращения JSON-хэша атрибутов экземпляра, который будет использоваться для обновления экземпляра.

Для сохранения модели на сервере следует вызвать принадлежащую модели функцию `save([`*`attrs`*`], [`*`options`*`])`, которой можно также передать хэш атрибутов и дополнения к запросу. Если у модели есть `id`, предполагается, что она существует на стороне сервера, и функция `save()` отправит PUT-запрос на обновление (update) . В противном случае функция `save()` отправит POST-запрос на создание (create):

```
var user = new User();
user.set({name: "Bernard"});

user.save(null, {success: function(){
  // пользователь успешно сохранен
}});
```

Все вызовы `save()` имеют асинхронный характер, но можно слушать функции обратного вызова Ajax-запроса, передавая дополнения `success` и `failure`. Фактически, если Backbone использует jQuery, любые дополнения, передаваемые функции `save()`, будут также передаваться и функции `$.ajax()`. Иными словами, при сохранении моделей вы можете использовать любые *Ajax-дополнения* jQuery, например `timeout`.

Если сервер возвращает ошибку и провести сохранение не удается, в отношении модели инициируется событие `error`. Если все пройдет успешно, модель будет обновлена с помощью ответа сервера:

```
var user = new User();

user.bind("error", function(e){
  // Сервер вернул ошибку!
});

user.save({email: "Неправильный адрес электронной почты"});
```

Можно обновить модель, воспользовавшись функцией `fetch()`, которая запросит атрибуты модели с сервера (путем GET-запроса). Если удаленное представ-

ление модели отличается от ее текущих атрибутов, будет инициировано событие change:

```
var user = Users.get(1);
user.fetch();
```

Заполнение коллекций

Итак, мы рассмотрели создание и обновление моделей, а как насчет первоначального извлечения их из сервера? И здесь на сцену выходят коллекции Backbone, запрос удаленных моделей и сохранение их на локальной машине. Как и при работе с моделями, для указания конечной точки коллекции к ней нужно добавить свойство url. Если это свойство не предоставлено, Backbone воспользуется свойством url соответствующей модели:

```
var Followers = Backbone.Collection.extend({
  model: User,
  url: "/followers"
});

Followers.fetch();
```

Принадлежащая коллекции функция fetch() отправит GET-запрос на сервер, в данном случае к /followers, извлекая удаленные модели. Когда данные моделей вернутся с сервера, коллекция будет обновлена, инициируя событие refresh.

С помощью функции refresh() вы можете обновить коллекцию и по собственной инициативе, передав ей массив объектов моделей. Реально это может пригодиться при первой установке страницы. Вместо выдачи еще одного GET-запроса на загрузку страницы, вы можете заранее заполнить данные коллекции, передав посредством функции refresh() встроенный JSON-объект. Вот как, к примеру, это будет выглядеть с использованием Rails:

```
<script type="text/javascript">
  Followers.refresh(<%= @users.to_json %>);
</script>
```

На серверной стороне

Как уже ранее упоминалось, для интеграции с Backbone на вашем сервере должны быть реализованы несколько конечных точек RESTful:

```
create → POST   /коллекция
read   → GET    /коллекция
read   → GET    /коллекция/id
update → PUT    /коллекция/id
delete → DELETE /коллекция/id
```

Перед отправкой Backbone переведет модели в последовательный формат JSON. Наша модель User будет выглядеть следующим образом:

```
{"name": "Yasmine"}
```

Обратите внимание, что у данных нет префикса, указывающего на текущую модель, что особенно сбивает с толку Rails-разработчиков. Я собираюсь рассмотреть ряд особенностей интеграции Rails с Backbone, поэтому, если вы не пользуетесь данной средой, можете переходить к изучению следующего раздела.

Внутри ваших CRUD-методов нужно использовать простые аргументы без префиксов. Например, вот как может работать Rails-метод `update`, принадлежащий контроллеру:

```
def update
  user = User.find(params[:id])
  user.update_attributes!(params)
  render :json => user
end
```

Разумеется, вы можете обезопасить свою модель от злонамеренного ввода путем создания белого списка атрибутов с помощью метода `attr_accessible`, но это уже выходит за рамки тем, рассматриваемых в данной книге. Каждый метод контроллера, за исключением метода `destroy`, вернет JSON-представление записи.

Перевод атрибутов в последовательный формат JSON также является проблемой, поскольку по умолчанию Rails перед каждыми данными записи ставит префикс, указывающий на модель:

```
{"user": {"name": "Daniela"}}
```

К сожалению, Backbone не сможет правильно проанализировать этот объект. Нужно сделать так, чтобы Rails не включала имя модели вовнутрь последовательного JSON- формата записей, для чего следует создать файл-инициализатор:

```
# config/initializers/json.rb
ActiveRecord::Base.include_root_in_json = false
```

Настройка поведения

`Backbone.sync()` является функцией Backbone, которая вызывается при каждой попытке чтения или сохранения модели на сервере. Вы можете подменить ее поведение по умолчанию (отправку Ajax-запроса) с целью использования другой стратегии сохраняемости, например WebSockets, XML-транспортировки или локального хранилища. Давайте, к примеру, заменим `Backbone.sync()` функцией-пустышкой, которая просто протоколирует аргументы, с которыми она вызвана:

```
Backbone.sync = function(method, model, options) {
  console.log(method, model, options);
  options.success(model);
};
```

Как видно в данном примере, `Backbone.sync()` получает переданные метод (`method`), модель (`model`) и дополнительные аргументы (`options`), имеющие следующие свойства:

`method`

CRUD-метод (создание — `create`, чтение— `read`, обновление — `update` или удаление — `delete`)

`model`

Сохраняемая модель (или считываемая коллекция)

`options`

Дополнительные аргументы, включая функции обратного вызова, связанные с успехом и неудачей

Единственное, что Backbone от вас ожидает, это вызов одной из функций обратного вызова: либо `options.success()`, либо `options.error()`.

Также вместо глобальной подмены `sync` ее можно подменить для модели или коллекции:

```
Todo.prototype.sync = function(method, model, options){ /* ... */ };
```

Хорошим примером специально созданной функции `Backbone.sync()` может послужить *адаптер локального хранилища*. Включение адаптера и настройка дополнительного параметра `localStorage` соответствующей модели или коллекции позволяет Backbone воспользоваться вместо сервера базы данных локальным хранилищем HTML5 `localStorage`. Как показано в приводимом ниже примере, `Backbone.sync()` совершает CRUD-операции над объектом `store`, в зависимости от метода, и в завершение вызывает метод `options.success()` с соответствующей моделью:

```
// Сохранение всех элементов todo в пространстве имен локального хранилища
// "todos" localStorage.
Todos.prototype.localStorage = new Store("todos");

// Подмена Backbone.sync() для использования делегирования
// к свойству localStorage
// модели или коллекции, которая должна быть экземпляром Store.
Backbone.sync = function(method, model, options) {

  var resp;
  var store = model.localStorage || model.collection.localStorage;

  switch (method) {
    case "read":    resp = model.id ? store.find(model) : store.findAll();
                    break;
    case "create":  resp = store.create(model);
                    break;
    case "update":  resp = store.update(model);
                    break;
    case "delete":  resp = store.destroy(model);
                    break;
  }
```

```
  if (resp) {
    options.success(resp);
  } else {
    options.error("Запись не найдена");
  }
};
```

Создание списка To-Do (текущих дел)

Давайте применим все, что узнали о Backbone, на практике и создадим небольшое приложение, работающее со списком текущих дел. Нам нужно, чтобы пользователь мог совершать CRUD-операции с отдельными делами и чтобы элементы списка сохранялись между обновлениями страницы. Вы можете создать приложение, используя показанные ниже примеры, или же посмотреть его законченный вариант в assets/ch12/todos.

Исходная структура страницы, в которую мы загружаем CSS, JavaScript-библиотеки и наше Backbone-приложение, содержащееся в todos.js, имеет следующий вид:

```
<html>
<head>
  <link href="todos.css" media="all" rel="stylesheet" type="text/css"/>
  <script src="lib/json2.js"></script>
  <script src="lib/jquery.js"></script>
  <script src="lib/jquery.tmpl.js"></script>
  <script src="lib/underscore.js"></script>
  <script src="lib/backbone.js"></script>
  <script src="lib/backbone.localstorage.js"></script>
  <script src="todos.js"></script>
</head>
<body>
  <div id="todoapp">
    <div class="title">
      <h1>Todos</h1>
    </div>
    <div class="content">
      <div id="create-todo">
        <input id="new-todo" placeholder="Что должно быть сделано?"
                   type="text" />
      </div>
      <div id="todos">
        <ul id="todo-list"></ul>
      </div>
    </div>
  </div>
</body>
</html>
```

Структура страницы довольно проста: в ней всего лишь содержится текстовое поле ввода для создания новой задачи (`#new-todo`) и список, показывающий уже существующие задачи (`#todo-list`).

Теперь давайте перейдем к сценарию todos.js, где находится ядро нашего Backbone-приложения. Мы собираемся заключить в `jQuery()` все, что помещаем в этот класс, обеспечивая тем самым запуск только после загрузки страницы:

```
// todos.js
jQuery(function($){
  // Сюда помещается приложение...
})
```

Давайте создадим основную модель `Todo` с атрибутами `content` и `done`. Чтобы легче было инвертировать атрибут модели `done`, мы предоставляем вспомогательную функцию `toggle()`:

```
window.Todo = Backbone.Model.extend({
  defaults: {
    done: false
  },

  toggle: function() {
    this.save({done: !this.get("done")});
  }
});
```

Чтобы обеспечить глобальную доступность, мы устанавливаем модель `Todo` для объекта `window`. Кроме того, использование данной схемы облегчает просмотр глобальных переменных, объявленных в сценарии, нужно лишь просмотреть сценарий в поисках ссылок на `window`.

Следующим шагом будет установка коллекции `TodoList`, где будет храниться массив моделей `Todo`:

```
window.TodoList = Backbone.Collection.extend({
  model: Todo,

  // Сохранение всех to-do элементов в пространстве имен "todos".
  localStorage: new Store("todos"),

  // Фильтрация всех выполненных дел для их попадания
  // в нижнюю часть списка.
  done: function() {
    return this.filter(function(todo){ return todo.get('done'); });
  },

  remaining: function() {
    return this.without.apply(this, this.done());
  }
});
```

```
// Создание нашей глобальной коллекции Todos.
window.Todos = new TodoList;
```

Мы используем Backbone-провайдер (`backbone.localstorage.js`), который требует от нас установки атрибута `localStorage` для любых коллекций или моделей, ожидающих сохранения данных. Остальные две функции в `TodoList`, `done()` и `remaining()`, занимаются фильтрацией коллекции, перезапуском моделей текущих дел, которые уже были или не были завершены. Поскольку у нас будет только один экземпляр `TodoList`, мы создаем этот экземпляр с глобальной доступностью: `window.Todos`.

А теперь представление, которое будет отображать отдельные дела, `TodoView`. Оно будет привязано к событию `change` моделей `Todo` для его повторной визуализации при инициировании этого события:

```
window.TodoView = Backbone.View.extend({

  // Представление представляет собой тег списка.
  tagName:  "li",

  // Кэширование функции template для отдельного элемента.
  template: $("#item-template").template(),

  // Делегирование событий  для функций представления
  events: {
    "change   .check"        : "toggleDone",
    "dblclick .todo-content" : "edit",
    "click    .todo-destroy" : "destroy",
    "keypress .todo-input"   : "updateOnEnter",
    "blur     .todo-input"   : "close"
  },

  initialize: function() {
    // Обеспечение запуска функций в нужной области видимости
    _.bindAll(this, 'render', 'close', 'remove');

    // Прислушивание к изменениям модели
    this.model.bind('change', this.render);
    this.model.bind('destroy', this.remove);
  },

  render: function() {
    // Обновление el сохраненным шаблоном
    var element = jQuery.tmpl(this.template, this.model.toJSON());
    $(this.el).html(element);
    return this;
  },

  // Переключение статуса модели done (сделано) при установке флажка
  toggleDone: function() {
```

продолжение ⇗

```
    this.model.toggle();
  },

  // Переключение этого представления в режим `"editing"` (редактирование),
  // с отображением поля ввода.
  edit: function() {
    $(this.el).addClass("editing");
    this.input.focus();
  },

  // Закрытие режима `"editing"`, с сохранением изменений в списке текущих дел.
  close: function(e) {
    this.model.save({content: this.input.val()});
    $(this.el).removeClass("editing");
  },

  // Если нажать клавишу `ввод`, редактирование элемента завершается.
  // Инициирование события blur в отношении поля ввода,
  // вызывая функцию close()
  updateOnEnter: function(e) {
    if (e.keyCode == 13) e.target.blur();
  },

  // Удаление элемента при ликвидации модели
  remove: function() {
    $(this.el).remove();
  },

  // Ликвидация модели при щелчке на '.todo-destroy'
  destroy: function() {
    this.model.destroy();
  }
});
```

Как видите, мы делегировали пакет событий представлению, управляющему обновлением, заполнением и удалением текущих дел. Например, при изменении состояния флажка вызывается функция `toggleDone()`, переключающая состояние атрибута модели `done`. Это, в свою очередь, инициирует событие модели `change`, которое заставляет представление заново провести визуализацию.

Для обработки HTML-шаблонов мы используем jQuery.tmpl, заменяя содержимое `el` при визуализации представления регенерируемым шаблоном. Шаблон ссылается на элемент, имеющий ID, равный `#item-template`, который мы еще не определили. Давайте сделаем это сейчас, поместив шаблон в body-тег нашего файла index.html:

```
<script type="text/template" id="item-template">
  <div class="todo {{if done}}done{{/if}}">
    <div class="display" title="Дважды щелкнуть для редактирования...">
```

```
      <input class="check" type="checkbox"
                  {{if done}}checked="checked"{{/if}} />
      <div class="todo-content">${content}</div>
      <span class="todo-destroy"></span>
    </div>
    <div class="edit">
      <input class="todo-input" type="text" value="${content}" />
    </div>
  </div>
</script>
```

Этот синтаксис работы с шаблонами должен быть вам достаточно знаком по главе 5, где модуль jQuery.tmpl был рассмотрен несколько глубже. По сути, мы взаимодействуем с содержимым внутри элементов `#todo-content` и `#todo-input`. Кроме того, обеспечивается правильное состояние флажка `"checked"`.

Представление `TodoView` является достаточно независимым — нужно только передать его модели при создании экземпляра и добавить его атрибут `el` к списку текущих дел. В основном это задача интерфейса `AppView`, обеспечивающего заполнение нашего списка текущих дел создаваемыми экземплярами `TodoView`. Еще одна роль `AppView` состоит в создании новых записей `Todo`, когда пользователь нажимает клавишу **Ввод**, если фокус находится в поле ввода `#new-todo`:

```
// Весь наш AppView является частью UI высокого уровня.
window.AppView = Backbone.View.extend({

  // Вместо создания нового элемента привязка к существующей основе
  // приложения, уже присутствующей в HTML.
  el: $("#todoapp"),

  events: {
    "keypress #new-todo":  "createOnEnter",
    "click .todo-clear a": "clearCompleted"
  },

  // При инициализации мы привязываемся к соответствующему событию
  // в коллекции `Todos`, когда элементы добавляются или изменяются.
  // Продолжим загрузкой любого заранее существующего текущего
  // дела, которое может быть сохранено в локальном хранилище
  // *localStorage*.
  initialize: function() {
    _.bindAll(this, 'addOne', 'addAll', 'render');
    this.input = this.$("#new-todo");
    Todos.bind('add',     this.addOne);
    Todos.bind('refresh', this.addAll);
    Todos.fetch();
  },
```

продолжение ⇨

```
  // Добаление отдельного текущего дела к списку путем создания для него
  // представления и добавления его элемента к `<ul>`.
  addOne: function(todo) {
    var view = new TodoView({model: todo});
    this.$("#todo-list").append(view.render().el);
  },

  // Добавление сразу всех элементов к коллекции Todos.
  addAll: function() {
    Todos.each(this.addOne);
  },

  // Если вы нажали клавишу Ввод на основном поле ввода,
  // создается новая модель
  // Todo
  createOnEnter: function(e) {
    if (e.keyCode != 13) return;
    var value = this.input.val();
    if ( !value ) return;
    Todos.create({content: value});
    this.input.val('');
  },
  clearCompleted: function() {
    _.each(Todos.done(), function(todo){ todo.destroy(); });
    return false;
  }
});

// И наконец, продолжим это созданием экземпляра App.
window.App = new AppView;
```

При начальной загрузке страницы коллекция `Todos` будет заполнена, и будет инициировано событие `refresh`. Оно приведет к вызову функции `addAll()`, которая извлекает модели `Todo`, генерирует представления `TodoView` и добавляет их к `#todo-list`. Кроме того, когда новая модель `Todo` добавляется к `Todos`, инициируется новое событие `Todos`, которое вызывает функцию `addOne()` и добавляет новое представление `TodoView` к списку. Иными словами, исходное заполнение и создание `Todo` обрабатывается `AppView`, а отдельные представления `TodoView` занимаются обновлением и удалением самих себя.

Теперь давайте обновим страницу и посмотрим на результат нашей ручной работы. Если не брать в расчет некоторые ошибки и опечатки, вы должны увидеть что-то похожее на рис. 12.1.

У нас есть функциональные возможности для добавления, проверки, обновления и удаления текущих дел, и все это с помощью относительно небольшого объема кода. Поскольку мы используем Backbone-адаптер локального хранилища, текущие дела сохраняются между перезагрузками страницы. Этот пример должен дать

вам неплохое представление о пользе Backbone, а также о том, как создавать свои собственные приложения.

Полную версию приложения можно найти в assets/ch12/todos.

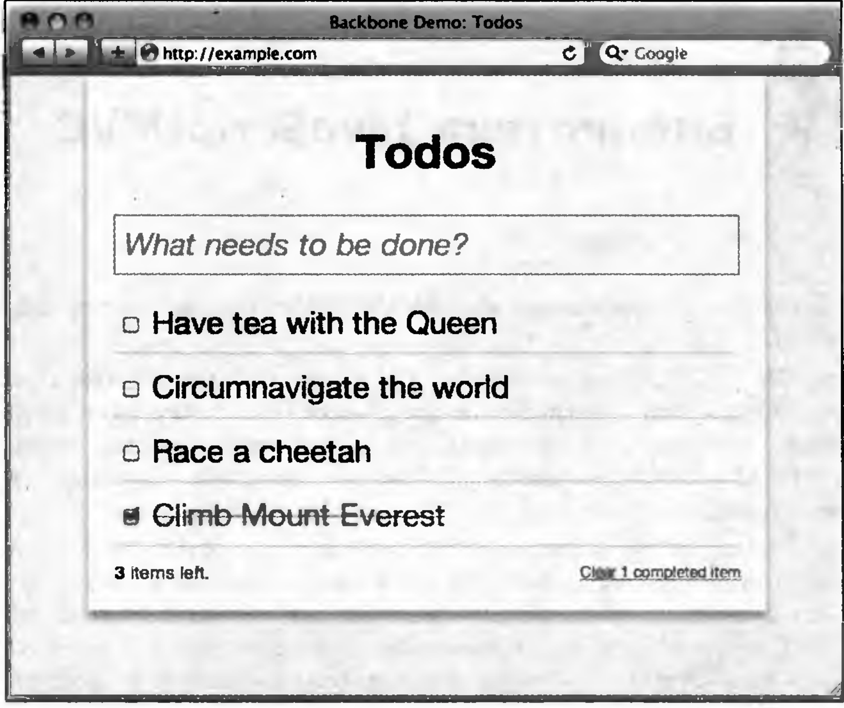

Рис. 12.1. Завершенное Backbone-приложение Todo

13 Библиотека JavaScriptMVC

Эту главу любезно предоставил Джастин Мейер (Justin Meyer), автор библиотеки JavaScriptMVC

JavaScriptMVC (JMVC) является JavaScript-средой с открытым кодом, созданной на основе jQuery. Это практически всесторонняя (цельная) структура разработки внешнего интерфейса, вобравшая в себя полезные средства для тестирования, управления зависимостями, документацию и являющаяся сборником полезных дополнительных модулей jQuery.

И все же каждая часть JavaScriptMVC может использоваться без любой другой ее части, облегчая тем самым библиотеку. В совокупности ее класс, модель, представление и контроллер занимают в минифицированном и сжатом состоянии всего 7 Кб, и даже они могут использоваться независимо друг от друга. Независимость JavaScriptMVC позволяет вам начинать с малого и расширяться до размеров, позволяющих отвечать потребностям наиболее сложных приложений Интернета.

В данной главе рассматриваются имеющиеся в JavaScriptMVC компоненты `$.Class`, `$.Model`, `$.View` и `$.Controller`. Рассмотрим описание каждого из них.

`$.Class`
: Система классов, основанная на JavaScript

`$.Model`
: Традиционный уровень модели

`$.View`
: Система работы с шаблонами на стороне клиента

`$.Controller`
: Средство jQuery для создания виджетов

Имеющееся в JavaScriptMVC соглашение об именах немного отличается от традиционной схемы разработки модель-представление-контроллер (MVC). Например, `$.Controller` используется для создания традиционных контроллеров представлений, вроде кнопок разбиения на страницы и списков, а также тради-

ционных контроллеров, являющихся координаторами между традиционными представлениями и моделями.

Установка

Библиотека JavaScriptMVC может использоваться в виде единой загрузки, включающей всю структуру. Но поскольку в данной главе рассматривается только MVC-части, нужно перейти к *компоновщику загрузки* (download builder, http://javascriptmvc.com/builder.html), установить флажки напротив Controller, Model и View EJS templates и щелкнуть на ссылке Download.

Произойдет загрузка минифицированной и неминифицированной версий jQuery и выбранных вами дополнительных модулей. Загрузите их с помощью `script`-тегов в вашу страницу:

```
<script type='text/javascript' src='jquery-1.6.1.js'></script>
<script type='text/javascript' src='jquerymx-1.0.custom.js'></script>
```

Классы

JMVC-контроллер и модель наследуются из вспомогательного класса библиотеки `$.Class`. Для создания класса нужно вызвать функцию `$.Class(`*`имя`*`, [`*`свойстваКласса`*`, ] `*`свойстваЭкземпляра`*`])`:

```
$.Class("Animal",{
breathe : function(){
console.log('breathe');
}
});
```

В показанном выше примере экземпляры класса `Animal` имеют метод `breathe()`. Мы можем создать новый экземпляр класса `Animal` и вызвать в отношении него метод `breathe()`:

```
var man = new Animal();
man.breathe();
```

Если нужно создать подкласс, следует просто вызвать базовый класс с именем подкласса и свойствами:

```
Animal("Dog",{
    wag : function(){
        console.log('wag');
    }
})

var dog = new Dog;
dog.wag();
dog.breathe();
```

Создание экземпляра

При создании нового экземпляра класса вызывается метод класса `init` с аргументами, переданными функции-конструктору:

```
$.Class('Person',{
    init : function(name){
        this.name = name;
    },

    speak : function(){
        return "I am " + this.name + ".";
    }
});

var payal = new Person("Payal");
assertEqual( payal.speak() , 'I am Payal.' );
```

Вызов основных методов

Основные методы следует вызывать с помощью `this._super`. Следующий код переписывает `person`, чтобы предоставить более «стильное» приветствие:

```
Person("ClassyPerson", {
    speak : function(){
        return "Salutations, " + this._super();
    }
});

var fancypants = new ClassyPerson("Mr. Fancy");
assertEquals( fancypants.speak() , 'Salutations, I am Mr. Fancy.')
```

Представительства

Принадлежащий классу метод обратного вызова возвращает функцию, имеющую соответствующую установку `this`, подобную `$.proxy`. Следующий код создает класс `Clicky`, подсчитывающий количество щелчков:

```
$.Class("Clicky",{
    init : function(){
        this.clickCount = 0;
    },

    clicked: function(){
        this.clickCount++;
    },

    listen: function(el){
        el.click( this.callback('clicked') );
    }
})
```

```
var clicky = new Clicky();
clicky.listen( $('#foo') );
clicky.listen( $('#bar') ) ;
```

Статическое наследование

Классы позволяют определять наследуемые статические свойства и методы. Представленный ниже код позволяет извлекать экземпляр person с сервера путем вызова Person.findOne(ID, success(person)). В случае успеха происходит обратный вызов с экземпляром Person, у которого имеется метод speak:

```
$.Class("Person",{
    findOne : function(id, success){
        $.get('/person/'+id, function(attrs){
            success( new Person( attrs ) );
        },'json')
    }
},{
    init : function(attrs){
        $.extend(this, attrs)
    },
    speak : function(){
        return "I am "+this.name+".";
    }
})

Person.findOne(5, function(person){
    assertEqual( person.speak(), "I am Payal." );
})
```

Самодиагностика

Класс предоставляет пространство имен и доступ к имени класса и объекту пространства имен:

```
$.Class("Jupiter.Person");

Jupiter.Person.shortName; //-> 'Person'
Jupiter.Person.fullName; //-> 'Jupiter.Person'
Jupiter.Person.namespace; //-> Jupiter

var person = new Jupiter.Person();

person.Class.shortName; //-> 'Person'
```

Пример модели

Объединяя все в единое целое, мы можем создать основной уровень модели ORM-стиля. Путем простого наследования из Model мы можем запросить данные

у REST-служб и вернуть их заключенными в экземпляры унаследованной модели Model:

```
$.Class("Model",{
    findOne : function(id, success){
        $.get('/' + this.fullName.toLowerCase() + '/' + id,
        this.callback(function(attrs){
            success( new this( attrs ) );
        })
    },'json')
}
},{
    init : function(attrs){
        $.extend(this, attrs)
    }
})

Model("Person",{
    speak : function(){
        return "I am "+this.name+".";
    }
});

Person.findOne(5, function(person){
    alert( person.speak() );
});

Model("Task");

Task.findOne(7,function(task){
    alert(task.name);
});
```

Это работает так же, как уровень модели JavaScriptMVC.

Модель

Модель JavaScriptMVC и связанные с ней дополнительные модули предоставляют множество инструментов, предназначенных для организации данных модели, например для проведения проверок, создания связей, списков и многого другого. Но основная функциональность сконцентрирована на инкапсуляции служб, преобразовании типов и событиях.

Атрибуты и наблюдаемые объекты

На уровне модели, безусловно, важным обстоятельством является возможность получать и устанавливать свойства моделируемых данных и слушать изменения экземпляра модели. Это схема Наблюдателя, которая лежит в самой основе MVC-подхода — представления слушают изменения в модели.

К счастью, применение JavaScriptMVC существенно облегчает наблюдение за любыми данными. Хорошим примером может послужить разбиение на страницы. Зачастую на странице присутствуют сразу несколько объектов управления разбиением на страницы. Например, один объект управления может предоставлять кнопки страниц Next (Следующая) и Previous (Предыдущая), другой объект управления может уточнять, какие элементы просматриваются на текущей странице (например, «Просмотр элементов 1–20»).

Все органы управления разбиением на страницы нуждаются в одних и тех же данных.

`offset`
: Индекс первого показываемого элемента

`limit`
: Количество показываемых элементов

`count`
: Общее количество элементов

Мы можем смоделировать эти данные с помощью компонента JavaScriptMVC-библиотеки `$.Model`:

```
var paginate = new $.Model({
    offset: 0,
    limit: 20,
    count: 200
});
```

Теперь переменная `paginate` является наблюдаемым объектом. Мы можем передать ее в объекты управления разбиением на страницы, которые могут читать из нее данные, записывать в нее данные и слушать изменения свойств. Можно читать свойства обычным образом или с использованием метода `model.attr(ИМЯ)`:

```
assertEqual( paginate.offset, 0 );
assertEqual( paginate.attr('limit') , 20 );
```

Если производится щелчок на кнопке следующего элемента, нужно увеличить индекс первого показываемого элемента (`offset`). Вносить изменения в значения свойств следует с помощью метода `model.attr(ИМЯ, ЗНАЧЕНИЕ)`. Следующий код переместит `offset` на следующую страницу:

```
paginate.attr('offset', 20);
```

Когда состояние систем разбиения на страницы изменяется одним объектом управления, другие объекты управления должны быть об этом уведомлены. Привязаться к изменению конкретного атрибута и обновить данные объекта управления можно с помощью метода `model.bind(ATTR, success( ev, newVal ) )`:

```
paginate.bind('offset', function(ev, newVal){
    $('#details').text( 'Просмотр элементов ' + (newVal + 1 ) +
            '-' + this.count )
})
```

Прослушивать изменения любого атрибута можно также путем привязки к событию 'updated.attr':

```
paginate.bind('updated.attr', function(ev, newVal){
    $('#details').text( 'Просмотр элементов ' + (newVal+1 )+ '-' +
          this.count )
})
```

Следующий код показывает дополнительный модуль jQuery следующая-предыдущая страница, который принимает данные системы разбиения на страницы:

```
$.fn.nextPrev = function(paginate){
    this.delegate('.next','click', function(){
        var nextOffset = paginate.offset + paginate.limit;
        if( nextOffset < paginate.count){
            paginate.attr('offset', nextOffset );
        }
    });

    this.delegate('.prev','click', function(){
        var nextOffset = paginate.offset-paginate.limit;
        if( 0 < paginate.offset ){
            paginate.attr('offset', Math.max(0, nextOffset) );
        }
    });

    var self = this;
    paginate.bind('updated.attr', function(){
        var next = self.find('.next'),
        prev = self.find('.prev');
        if( this.offset == 0 ){
            prev.removeClass('enabled');
        } else {
            prev.removeClass('disabled');
        }
        if( this.offset > this.count - this.limit ){
            next.removeClass('enabled');
        } else {
            next.removeClass('disabled');
        }
    });
};
```

При использовании этого дополнительного модуля возникает несколько проблем. Во-первых, если объект управления удаляется со страницы, он не отвязывает себя от paginate. Мы займемся этой проблемой, когда будем рассматривать контроллеры.

Во-вторых, логика защиты от отрицательного индекса первого показываемого элемента (offset) или от его значения, превосходящего общее количество, заложена в дополнительный модуль. Эта логика должна закладываться в модель. Для решения данной проблемы нам нужно создать класс разбиения на страницы, где можно будет добавлять дополнительные ограничения, накладываемые на значения limit, offset и count.

Расширенные модели

Модель JavaScriptMVC наследуется из `$.Class`. Соответственно класс модели создается путем наследования из `$.Model(ИМЯ, [STATIC,] ПРОТОТИП)`:

```
$.Model('Paginate',{
    staticProperty: 'foo'
},{
    prototypeProperty: 'bar'
})
```

Существует несколько способов наилучшего использования модели Paginate. Путем добавления рассматриваемых в следующем разделе методов-установщиков можно ограничить значения, которые будут устанавливаться для общего количества элементов и индекса первого показываемого элемента.

Методы-установщики

Методы-установщики являются методами прототипа модели, имеющими имена set*ИМЯ*. Они вызываются со значением val, переданным методу `model.attr(ИМЯ, val)`, а также с функциями обратного вызова, связанными с удачным выполнением и с ошибкой. Обычно метод должен вернуть значение, которое установлено для экземпляра модели, или вызвать функцию обработки ошибки (error) с сообщением об ошибке. Функция обработки успешного завершения (success) используется для методов-установщиков, работающих в асинхронном режиме.

В модели Paginate методы-установщики используются для предотвращения установки неправильных значений общего количества элементов (count) и индекса первого показываемого элемента (offset). Например, мы обеспечиваем невозможность отрицательного значения:

```
$.Model('Paginate',{
    setCount : function(newCount, success, error){
        return newCount < 0 ? 0 : newCount;
    },

    setOffset : function(newOffset, success, error){
        return newOffset < 0 ? 0 :
        Math.min(newOffset, !isNaN(this.count - 1) ?
                this.count : Infinity )
    }
});
```

Теперь дополнительный модуль nextPrev может устанавливать индексы первого показываемого элемента (offset), отказываясь при этом присваивать неверные значения:

```
this.delegate('.next','click', function(){
    paginate.attr('offset', paginate.offset+paginate.limit);
});

this.delegate('.prev','click', function(){
    paginate.attr('offset', paginate.offset-paginate.limit );
});
```

Умолчания

Значения по умолчанию можно добавлять к экземплярам Paginate путем установки статического свойства defaults. Если при создании нового экземпляра paginate не предоставлено никакого значения, происходит инициализация значением по умолчанию:

```
$.Model('Paginate',{
    defaults : {
        count: Infinity,
        offset: 0,
        limit: 100
    }
},{
    setCount : function(newCount, success, error){ ... },
    setOffset : function(newOffset, success, error){ ... }
});

var paginate = new Paginate({count: 500});
assertEqual(paginate.limit, 100);
assertEqual(paginate.count, 500);
```

Перемещение к следующей или к предыдущей странице и выяснение возможности такого перемещения в модели Paginate можно сделать еще проще путем добавления методов-помощников.

Методы-помощники

Так называются методы прототипа, помогающие устанавливать или получать полезные данные, относящиеся к экземплярам модели. Следующая завершенная модель Paginate включает методы next и prev, которые будут осуществлять перемещения на следующую и предыдущую страницы, если это возможно. Эта модель также предоставляет методы canNext и canPrev, которые возвращают сведения о возможности или невозможности перемещения экземпляра на следующую страницу:

```
$.Model('Paginate',{
    defaults : {
        count: Infinity,
```

```
        offset: 0,
        limit: 100
    }
},{
    setCount : function( newCount ){
        return Math.max(0, newCount );
    },
        setOffset : function( newOffset ){
        return Math.max( 0 , Math.min(newOffset, this.count ) )
    },
        next : function(){
        this.attr('offset', this.offset+this.limit);
    },
    prev : function(){
        this.attr('offset', this.offset - this.limit )
    },
    canNext : function(){
        return this.offset > this.count - this.limit
    },
    canPrev : function(){
        return this.offset > 0
    }
})
```

Таким образом, наш jQuery-виджет становится намного совершеннее:

```
$.fn.nextPrev = function(paginate){
  this.delegate('.next','click', function(){
    paginate.attr('offset', paginate.offset+paginate.limit);
  })
  this.delegate('.prev','click', function(){
    paginate.attr('offset', paginate.offset-paginate.limit );
  });
  var self = this;
  paginate.bind('updated.attr', function(){
    self.find('.prev')[paginate.canPrev() ? 'addClass' :
                        'removeClass']('enabled')
    self.find('.next')[paginate.canNext() ? 'addClass' :
                       'removeClass']('enabled');
  })
};
```

Инкапсуляция служб

Мы уже убедились в том, насколько полезен компонент `$.Model` при моделировании состояния на стороне клиента. Но для большинства приложений самые важные данные находятся на сервере, а не у клиента. Клиенту нужно создавать, извлекать, обновлять и удалять (проводить CRUD-операции) данные, имеющиеся на сервере. Поддержание двойственности данных на клиенте и сервере — не такая уж и простая задача, и компонент `$.Model` значительно упрощает ее решение.

`$.Model` обладает исключительной гибкостью. Его можно приспособить для работы со всеми видами служб и типами данных. В данной книге рассматривается только вопрос работы `$.Model` с наиболее распространенными и популярными службами и типами данных: с передачей состояния представления — Representational State Transfer (REST) и с JSON.

Служба REST использует URL-адреса и HTTP-глаголы POST, GET, PUT и DELETE соответственно для создания, извлечения, обновления и удаления данных. Например, служба задач, позволяющая создавать, извлекать, обновлять и удалять задачи (tasks), может иметь следующий вид:

```
create → POST /tasks
read all → GET /tasks
read → GET /tasks/2
update → PUT /tasks/2
delete → DELETE /tasks/2
```

Следующий код осуществляет подключение к службам задач, позволяя нам создавать, извлекать, обновлять и удалять задачи, имеющиеся на сервере:

```
$.Model("Task",{
  create : "POST /tasks.json",
  findOne : "GET /tasks/{id}.json",
  findAll : "GET /tasks.json",
  update : "PUT /tasks/{id}.json",
  destroy : "DELETE /tasks/{id}.json"
},{ });
```

Давайте пройдем по всем шагам, необходимым для использования модели `Task` для выполнения CRUD-операций с задачами.

Создание задачи

```
new Task({ name: 'do the dishes'}).save(
  success( task, data ),
  error( jqXHR)
) //=> taskDeferred
```

Для создания экземпляра модели на сервере сначала нужно создать экземпляр с помощью кода `new Model(attributes)`. Затем нужно вызвать `save()`. Метод `save()` проверяет, есть ли у задачи идентификатор ID. В данном случае его нет, поэтому `save()` делает запрос на создание с атрибутами задачи. Этому методу передаются два аргумента.

`success`

: Функция, вызываемая, если сохранение проходит успешно. Функция `success` вызывается с экземпляром задачи и данными, возвращенными сервером.

`error`

: Функция, вызываемая, если в процессе выполнения запроса произошла ошибка. Она вызывается XHR-объектом, заключенным в jQuery.

Метод `save()` возвращает deferred-объект (значение которого определится только со временем), разрешаемый в созданную задачу.

Получение задачи

```
Task.findOne(params, ·
  success( task ),
  error( jqXHR)
) //=> taskDeferred
```

Извлечение отдельной задачи с сервера. Передаются три аргумента.

`params`

Данные для передачи на сервер; обычно это ID в виде `{id: 2}`.

`success`

Функция, вызываемая, если запрос завершается успешно. Функция `success` вызывается с экземпляром задачи.

`error`

Функция, вызываемая, если в процессе выполнения запроса произошла ошибка.

Метод `findOne()` возвращает deferred-объект, разрешаемый в задачу.

Получение задач

```
Task.findAll(params,
success( tasks ),
error( jqXHR)
) //=> tasksDeferred
```

Извлечение с сервера массива задач. Передаются три аргумента.

`params`

Данные для передачи на сервер. Обычно это пустой объект (`{}`) или фильтры `{limit: 20, offset: 100}`.

`success`

Функция, вызываемая, если запрос завершается успешно. Функция `success` вызывается с массивом экземпляров задач.

`error`

Функция, вызываемая, если в процессе выполнения запроса произошла ошибка.

Метод `findAll()`возвращает deferred-объект, разрешаемый в массив задач.

Обновление задачи

```
task.attr('name','take out recycling');
task.save(
  success( task, data ),
  error( jqXHR)
) //=> taskDeferred
```

Для обновления данных на сервере сначала нужно с помощью метода `attr` изменить атрибуты экземпляра модели. Затем нужно вызвать `save()`. Методу `save()` передаются такие же аргументы и им возвращается такой же отложенный ответ, как и при создании задачи.

Удаление задачи

```
task.destroy(
  success( task, data ),
  error( jqXHR)
) //=> taskDeferred
```

Метод `destroy()` удаляет задачу с сервера. Ему передаются два аргумента.

`success`

Функция, вызываемая, если удаление завершается успешно. Функция `success` вызывается с экземпляром задачи и с данными, возвращаемыми сервером.

`error`

Функция, вызываемая, если в процессе выполнения запроса произошла ошибка.

Как и `save()`, метод `destroy()` возвращает deferred-объект, разрешаемый в удаленную задачу. По сути, модель `Task` стала заказом на наши службы!

Преобразование типов

Вы заметили, как сервер отправляет в ответ значение свойства времени создания (`createdAt`) в виде чисел, похожих на `1303173531164`? Это число на самом деле означает 18 апреля 2011 года. Вместо получения числа в ответ на вызов `task.createdAt` было бы намного полезнее, если бы возвращалась дата JavaScript, созданная с помощью кода `new Date(1303173531164)`. Этого можно добиться с помощью метода-установщика `setCreatedAt`, но при наличии большого количества данных типа дата (`Date`) довольно скоро возникнет слишком много повторений.

Чтобы упростить решение проблемы, `$.Model` позволяет определить тип атрибута, а также функцию преобразования для таких типов данных. Тип атрибута нужно установить на статическом объекте `attributes`, а методы конвертирования нужно установить для статического объекта `convert`:

```
$.Model('Task',{
  attributes : {
    createdAt : 'date'
  },
  convert : {
    date : function(date){
      return typeof date == 'number' ? new Date(date) : date;
    }
  }
},{});
```

Теперь `Task` будет переводить значение `createdAt` в тип `Date`. Для вывода в списке года каждой задачи нужно создать следующий код:

```
Task.findAll({}, function(tasks){
  $.each(tasks, function(){
    console.log( "Year = "+this.createdAt.fullYear() )
  })
});
```

CRUD-события

Модель публикует события при создании, обновлении или удалении экземпляра. Вы можете слушать эти события глобально в отношении всей модели или в отношении отдельного экземпляра модели. Для прослушивания событий создания, обновления или удаления нужно воспользоваться методом `MODEL.bind(`*`СОБЫТИЕ`*`, callback( ev, instance ) )`.

Пусть нужно узнать о моменте создания задачи, чтобы мы могли ее добавить к нашей странице. После ее добавления мы будем слушать обновление этой задачи, чтобы обеспечить правильное отображение ее названия. Вот как это можно сделать:

```
Task.bind('created', function(ev, task){
  var el = $('<li>').html(todo.name);
  el.appendTo($('#todos'));

  task.bind('updated', function(){
    el.html(this.name);
  }).bind('destroyed', function(){
    el.remove();
  })
});
```

Использование в представлении шаблонов на стороне клиента

JavaScriptMVC-представления в действительности являются шаблонами на стороне клиента, которым передаются данные, а ими возвращается строковое значение. Обычно строки представляют собой HTML, предназначенный для вставки в DOM.

Компонент `$.View` является интерфейсом для работы с шаблонами, который сам использует шаблоны, чтобы взять на себя все сложности. Он предлагает:

- удобный и унифицированный синтаксис;
- загрузку шаблонов из HTML-элементов или внешних файлов;
- синхронную и асинхронную загрузку шаблонов;
- предварительную загрузку шаблонов;

- кэширование обработанных шаблонов;
- объединение обработанных шаблонов в производственные сборки;
- поддержку компонента `$.Deferred`.

Библиотека JavaScriptMVC поставляется с заранее составленными пакетами из четырех различных механизмов шаблонов:

- EJS
- JAML
- Micro
- Tmpl

В данном руководстве используются EJS-шаблоны, но следующие технические приемы будут работать с любыми механизмами шаблонов (с незначительными синтаксическими различиями).

Основное применение

При использовании представлений практически всегда возникает потребность во вставке результатов интерпретации шаблона в страницу. `jQuery.View` переписывает jQuery-модификаторы, поэтому использование представления сводится к следующему простому коду:

```
$("#foo").html('mytemplate.ejs',{message: 'hello world'})
```

Этот код:

1. Загружает шаблон в файл mytemplate.ejs. Он может иметь следующий вид:
   ```
   <h2><%= message %></h2>
   ```
2. Интерпретирует его с использованием `{message: 'hello world'}`, в результате чего получается следующий код:
   ```
   <h2>hello world</h2>
   ```
3. Вставляет результат в элемент `foo`, что может выглядеть следующим образом:
   ```
   <div id='foo'><h2>hello world</h2></div>
   ```

Модификаторы jQuery

Шаблон можно использовать со следующими методами-модификаторами jQuery:

```
$('#bar').after('temp.ejs',{});
$('#bar').append('temp.ejs',{});
$('#bar').before('temp.ejs',{});
$('#bar').html('temp.ejs',{});
$('#bar').prepend('temp.ejs',{});
$('#bar').replaceWith('temp.ejs',{});
$('#bar').text('temp.ejs',{});
```

Загрузка из script-тега

Представление может загружаться из `script`-тегов или из файлов. Для загрузки из `script`-тега нужно создать `script`-тег с атрибутом `type`, для которого установлено значение типа шаблона (`text/ejs`), и с атрибутом `id`, чтобы пометить шаблон:

```
<script type='text/ejs' id='recipesEJS'>
  <% for(var i=0; i < recipes.length; i++){ %>
  <li><%=recipes[i].name %></li>
  <%} %>
</script>
```

Визуализация с данным шаблоном выглядит следующим образом:

```
$("#foo").html('recipesEJS', recipeData)
```

Обратите внимание, что мы передали `id` того элемента, который хотим визуализировать.

$.View и подшаблоны

Иногда нужно просто вывести строку. В таком случае можно воспользоваться кодом `$.View(`*`ШАБЛОН , данные`* `)` напрямую. Методу `$.View` нужно передать шаблон, а также данные, которые нужно вывести:

```
var html = $.View("template/items.ejs", items );
```

Наиболее часто используются подшаблоны. Существует устоявшаяся практика выделения шаблона отдельного элемента из шаблона списка (items.ejs). Мы заставим template/items.ejs вывести <> для каждого элемента, но для содержимого каждого элемента воспользуемся шаблоном в template/item.ejs:

```
<% for( var i = 0; i < this.length; i++){ %>
  <li>
  <%= $.View("template/item.ejs", this[i]);
  </li>
< % } %>
```

`this` ссылается на данные, переданные шаблону. В случае использования template/items.ejs `this` является массивом элементов. В template/item.ejs `this` будет отдельным элементом.

Deferred-объекты

Отправка Ajax-запроса и получение шаблона для визуализации результата является вполне типичным поведением веб-приложения. Используя модель `Task` из предыдущего раздела, посвященного `$.Model`, мы можем визуализировать задачи с помощью следующего кода:

```
Task.findAll({}, function(tasks){
  $('#tasks').html("views/tasks.ejs" , tasks )
})
```

`$.View` поддерживает deferred-объекты (http://api.jquery.com/category/deferred-object/), допускающие применение весьма мощного, сжатого и высокоэффективного синтаксиса. Если в выводимых данных, передаваемых `$.View` или jQuery-модификаторам, находится deferred-объект, `$.View` загрузит шаблон в асинхронном режиме и, перед тем как визуализировать шаблон, будет ждать, пока не будут загружены все deferred-объекты и шаблоны.

Deferred-объекты возвращаются методами модели `findAll`, `findOne`, `save` и `destroy`. Это позволяет нам переписать визуализацию задач в одну строку:

```
$('#tasks').html("views/tasks.ejs" , Task.findAll() )
```

Этот код будет работать и с несколькими deferred-объектами:

```
$('#app').html("views/app.ejs" , {
  tasks: Task.findAll(),
  users: User.findAll()
})
```

Упаковка, предварительная загрузка и производительность

По умолчанию `$.View` загружает шаблоны в синхронном режиме, поскольку ожидается, что вы:

- помещаете шаблоны в `script`-теги;
- упаковываете шаблоны с помощью компоновщика JavaScript;
- осуществляете предварительную загрузку шаблонов.

Для JavaScriptMVC помещать шаблоны в `script`-теги не рекомендуется. Шаблоны в `script`-тегах затрудняют повторное использование шаблонов разными JavaScript-приложениями. Они также могут снизить производительность загрузки, если ваше приложение не испытывает немедленной потребности в данных шаблонах.

Для JavaScriptMVC рекомендуется сначала упаковать используемые шаблоны с JavaScript-кодом вашего приложения, а затем предварительно загрузить те шаблоны, надобность в которых наступит позже.

Имеющаяся в JavaScriptMVC система компоновки StealJS может обрабатывать и паковать шаблоны, добавляя их к минифицированной товарной сборке. Для этого нужно просто указать в коде `steal.views(PATH, ...)` на ваш шаблон:

```
steal.views('tasks.ejs','task.ejs');
```

Впоследствии, когда `$.View` будет искать этот шаблон, будет использоваться копия, находящаяся в кэш-памяти, что поможет избежать дополнительного Ajax-запроса.

В отношении шаблонов, не подлежащих немедленному применению, нужно применить предварительную загрузку и кэширование с помощью `jQuery.get`. Для этого нужно просто предоставить URL шаблона и `dataType` со значением `'view'`

(лучше это сделать через небольшой промежуток времени после загрузки исходной страницы):

```
$(window).load(function(){
  setTimeout(function(){
    $.get('users.ejs',function(){},'view');
    $.get('user.ejs',function(){},'view');
  },500)
})
```

$.Controller: фабрика по производству дополнительных модулей для jQuery

JavaScriptMVC-контроллеры являются многофункциональным средством. Они являются настоящей фабрикой по производству дополнительных модулей для jQuery. Их можно использовать в качестве традиционного представления, создавая элементы интерфейса разбиения на страницы и структурные элементы управления. Они также могут использоваться в качестве традиционного контроллера, инициализируя контроллеры и подключая их к моделям. По большому счету, контроллеры действительно являются отличным способом организации кода вашего приложения.

Контроллеры предоставляют множество полезных возможностей, среди которых:

- создание дополнительных модулей jQuery;
- автоматическая привязка;
- параметры, используемые по умолчанию;
- автоматический детерминизм.

Но наиболее важное свойство контроллера не является очевидным для всех и известно лишь самым рьяным приверженцам JavaScript. Следующий код создает элемент интерфейса, похожий на контекстные подсказки, который выводится до тех пор, пока не будет осуществлен щелчок на документе:

```
$.fn.tooltip = function(){
  var el = this[0];

  $(document).click(function(ev){
    if (ev.target !== el)
    $(el).remove();
  });

  $(el).show();
  return this;
});
```

Чтобы им воспользоваться, следует добавить элемент, отображаемый на странице, а затем вызвать в отношении этого элемента метод tooltip:

```
$("<div class='tooltip'>Некая информация</div>")
.appendTo(document.body)
.tooltip()
```

Но в этом коде есть одна проблема. Сможете ли вы определить, в чем ее суть? Дадим наводку: что, если ваше приложение рассчитано на длительный срок использования и создано множество подобных элементов подсказок?

Проблема в том, что на этот код бесполезно тратится слишком много памяти! Каждый элемент подсказки и любой дочерний элемент подсказки хранятся в памяти на постоянной основе. Причина в том, что обработчик щелчка не удаляется из документа и у него имеется замкнутая ссылка на элемент.

Такую ошибку допустить совсем нетрудно. jQuery удаляет все обработчики событий из элементов, удаляя их и из страницы, поэтому разработчикам зачастую не следует волноваться насчет не связанных с элементами обработчиков событий. Но в данном случае имеется привязка не к элементу, а к документу, поэтому обработчик событий отвязать не удается.

Но, согласно архитектуре модель-представление-контроллер (MVC), контроллеры слушают представления, а представления слушают модель. Вы постоянно слушаете события за пределами отображаемых элементов. Например, виджет nextPrev из раздела $.Model слушает обновления из модели разбиения на страницы paginate:

```
paginate.bind('updated.attr', function(){
  self.find('.prev')[this.canPrev() ? 'addClass' :
           'removeClass']('enabled')
  self.find('.next')[this.canNext() ? 'addClass' :
           'removeClass']('enabled');
})
```

Но он не отвязан от paginate! Упущения, связанные с тем, что обработчики событий не удаляются, являются потенциальным источником ошибок. Но наличие и tooltip, и nextPrev не будет являться ошибкой. И тем не менее вместе взятые они будут молча убивать производительность приложения. К счастью, $.Controller облегчает ситуацию и приводит все в надлежащий порядок. Вот как можно написать код tooltip:

```
$.Controller('Tooltip',{
  init: function(){
    this.element.show()
  },
  "{document} click": function(el, ev){
    if(ev.target !== this.element[0]){
      this.element.remove()
    }
  }
})
```

Когда происходит щелчок на документе и элемент удаляется из DOM, `$.Controller` отвяжет обработчик щелчка на документе автоматически.

Аналогичные действия `$.Controller` может делать и для виджета `nextPrev`, привязанного к модели `Paginate`:

```
$.Controller('Nextprev',{
  ".next click" : function(){
    var paginate = this.options.paginate;
    paginate.attr('offset', paginate.offset+paginate.limit);
  },
  ".prev click" : function(){
    var paginate = this.options.paginate;
    paginate.attr('offset', paginate.offset-paginate.limit );
  },
  "{paginate} updated.attr" : function(ev, paginate){
    this.find('.prev')[paginate.canPrev() ? 'addClass' :
             'removeClass']('enabled')
    this.find('.next')[paginate.canNext() ? 'addClass' :
             'removeClass']('enabled');
  }
})

// создание элемента управления nextprev
$('#pagebuttons').nextprev({ paginate: new Paginate() })
```

Если элемент `#pagebuttons` удаляется со страницы, экземпляр контроллера `NextPrev` будет автоматически отвязан от модели `Paginate`.

Теперь, когда ваш аппетит, связанный с кодом, свободным от ошибок, в достаточной степени удовлетворен, перейдем к следующим подробностям работы `$.Controller`.

Общее представление

`$.Controller` наследуется из класса `$.Class`. Для создания класса контроллера нужно вызвать `$.Controller(`*`имя, свойстваКласса, свойстваЭкземпляра`*`)` с именем вашего контроллера, статическими методами и методами экземпляра. Следующий код является началом многократно используемого виджета списка:

```
$.Controller("List", {
  defaults : {}
},{
  init : function(){ },
  "li click" : function(){ }
})
```

При создании класса контроллера создается метод-помощник jQuery, имеющий такое же имя. Этот метод используется, главным образом, для создания новых экземпляров контроллеров для элементов, имеющихся на странице. Имя метода-помощника составляется из имени контроллера с использованием символов

подчеркивания, где любые точки заменяются подчеркиваниями. Например, помощником для `$.Controller('App.FooBar')` будет `$(el).app_foo_bar()`.

Создание экземпляра контроллера

Для создания экземпляра контроллера вы можете осуществить вызов `new Controller(`*`элемент, дополнительные_аргументы`*`)`, указав HTML-элемент или элемент, заключенный в `jQuery`, а также объекты дополнительных аргументов, предназначенные для конфигурации контроллера. Например:

```
new List($('ul#tasks'), {model : Task});
```

Для создания экземпляра контроллера `List` в отношении элемента `#tasks` можно также воспользоваться методом-помощником jQuery:

```
$('ul#tasks').list({model : Task})
```

При создании контроллера этот метод вызывает метод `init` прототипа контроллера.

`this.element`

Установленный на HTML-элемент, заключенный в `jQuery`.

`this.options`

Установленный на дополнительные аргументы, переданные контроллеру, объединенные с принадлежащими классу объектами по умолчанию `defaults`.

Следующий код обновляет код контроллера `List` для запроса задач из модели и последующего отображения их вместе с дополнительным шаблоном, передаваемым списку.

```
$.Controller("List", {
  defaults : {
    template: "items.ejs"
  }
}, {
  init : function(){
    this.element.html( this.options.template, this.options.model.findAll()
);
  },
  "li click" : function(){ }
});
```

Теперь мы можем сконфигурировать контроллеры списков `List` на отображение задач с помощью предоставляемых шаблонов. Посмотрите, насколько все гибко получается!

```
$('#tasks').list({model: Task, template: "tasks.ejs"});
$('#users').list({model: User, template: "users.ejs"})
```

Если шаблон не предоставить, `List` будет по умолчанию использовать items.ejs.

Привязка событий

Как упоминалось во введении в `$.Controller`, наиболее эффективным свойством этого компонента является возможность привязывать и отвязывать обработчики событий.

При создании контроллера ведется поиск методов действий. Методами действий являются такие методы, которые похожи на обработчики событий, например `"li click"`. Эти действия привязываются с помощью `jQuery.bind` или `jQuery.delegate`. Когда контроллер уничтожается путем удаления элемента контроллера со страницы или вызова в отношении контроллера метода удаления, эти события отвязываются, предотвращая утечки памяти.

Рассмотрим примеры действий с описаниями того, что они слушают.

`"li click"`

Щелчки на элементах `li` или в пределах этих элементов в пределах элемента контроллера.

`"mousemove"`

Перемещения указателя мыши в пределах элемента контроллера.

`"{window} click"`

Щелчки на окне или в пределах окна.

Функции действия подвергаются обратному вызову с элементом, заключенным в `jQuery`, или с объектом, на котором произошло событие, а также с событием. Например:

```
"li click": function( el, ev ) {
  assertEqual(el[0].nodeName, "li" )
  assertEqual(ev.type, "click")
}
```

Шаблонные действия

`$.Controller` поддерживает шаблонные действия. Эти действия могут использоваться для привязки к другим объектам, настройки типа события или настройки селектора.

Контроллер заменяет часть ваших действий, имеющих вид `{ВАРИАНТ}`, значением, имеющимся в дополнительных аргументах контроллера или в окне.

Рассмотрим схему меню, которая позволяет настроить ее на показ подменю под различные события.

```
$.Controller("Menu",{
  "li {openEvent}" : function(){
    // показ подчиненных дочерних элементов
  }
});
```

```
// создание меню, показывающего дочерние элементы по событию щелчка (click)
$("#clickMenu").menu({openEvent: 'click'});

// создание меню, показывающего дочерние элементы по установке указателя мыши
// на элемент (по событию mouseenter)
$("#hoverMenu").menu({openEvent: 'mouseenter'});
```

Мы можем продолжить усовершенствование меню, позволяя настраивать тег элемента menu:

```
$.Controller("Menu",{
  defaults : {menuTag : "li"}
},{
"{menuTag} {openEvent}" : function(){
  // показ подчиненных дочерних элементов
  }
});

$("#divMenu").menu({menuTag : "div"})
```

Шаблонные действия позволяют осуществлять привязку к элементам или объектам за пределами элемента контроллера. Например, модель Task из раздела $.Model создает событие «created» при создании нового экземпляра модели Task. Мы можем заставить наш виджет списка прослушивать создаваемые задачи, а затем добавлять эти задачи к списку автоматически:

```
$.Controller("List", {
  defaults : {
    template: "items.ejs"
  }
},{
  init : function(){
  this.element.html( this.options.template, this.options.model.findAll() );
},
  "{Task} created" : function(Task, ev, newTask){
    this.element.append(this.options.template, [newTask])
  }
})
```

"{Task} created" вызывается вместе с моделью Task, событием создания и только что созданным экземпляром задачи Task. Функция использует шаблон для вывода списка задач (в данном случае имеется только одна задача) и добавления получившегося HTML к элементу.

Но гораздо лучше будет заставить List работать с любой моделью. Вместо конкретного указания задачи мы заставим контроллер в качестве варианта брать модель:

```
$.Controller("List", {
  defaults : {
    template: "items.ejs",
```

```
    model: null
  }
},{
  init : function(){
    this.element.html( this.options.template, this.options.model.findAll()
);
  },
  "{model} created" : function(Model, ev, newItem){
    this.element.append(this.options.template, [newItem])
  }
});

// создание списка задач
$('#tasks').list({model: Task, template: "tasks.ejs"});
```

Объединение компонентов: обобщенный CRUD-список

Теперь мы усовершенствуем список так, чтобы он не только добавлял к себе элементы при их создании, но и обновлял и удалял их при уничтожении. Для этого мы начнем слушать обновления и уничтожения:

```
"{model} updated" : function(Model, ev, updatedItem){
  // поиск и обновление LI для updatedItem
},
"{model} destroyed" : function(Model, ev, destroyedItem){
  // поиск и удаление LI для destroyedItem
}
```

И здесь следует заменить наличие небольшой проблемы. Нам каким-то образом нужно найти элемент, представляющий конкретный экземпляр модели. Для этого нужно пометить принадлежность элемента экземпляру модели. К счастью, компоненты `$.Model` и `$.View` существенно упрощают маркировку элемента в соответствии с принадлежностью его к конкретному экземпляру и поиск этого элемента.

Для маркировки элемента в соответствии с принадлежностью к экземпляру модели внутри EJS-представления, нужно просто вписать экземпляр модели в элемент. Следующий код может относиться к tasks.ejs:

```
<% for(var i =0 ; i < this.length; i++){ %>
  <% var task = this[i]; %>
  <li <%= task %> > <%= task.name %> </li>
<% } %>
```

tasks.ejs осуществляет последовательный перебор списка задач. Для каждой задачи (task) создается элемент `li` с именем задачи. Но кроме этого, с помощью конструкции `<li><%=task %></li>` задача добавляется к jQuery-данным элемента.

Для последующего получения этого элемента по заданному экземпляру модели нужно осуществить вызов `modelInstance.elements([CONTEXT])`. Этот код вернет элементы, заключенные в `jQuery`, представляющие экземпляр модели.

Если собрать все воедино, список приобретет следующий вид:

```
$.Controller("List", {
  defaults : {
    template: "items.ejs",
    model: null
  }
},{
  init : function(){
    this.element.html( this.options.template, this.options.model.findAll()
);
  },
  "{model} created" : function(Model, ev, newItem){
    this.element.append(this.options.template, [newItem])
  },
  "{model} updated" : function(Model, ev, updatedItem){
    updatedItem.elements(this.element)
    .replaceWith(this.options.template, [updatedItem])
  },
  "{model} destroyed" : function(Model, ev, destroyedItem){
    destroyedItem.elements(this.element)
    .remove()
  }
});

// создание списка задач
$('#tasks').list({model: Task, template: "tasks.ejs"});
```

Видите, как настораживающе просто с помощью JavaScriptMVC можно создавать абстрактные, многократно используемые и экономящие память виджеты.

A Основы jQuery

Для облегчения работы с DOM было разработано множество библиотек, но лишь немногие из них могут сравниться по популярности и славе с jQuery. И на это есть веские причины: в jQuery имеется превосходный API, сама библиотека невелика по объему и имеет замкнутое пространство имен, поэтому она не будет конфликтовать ни с каким другим используемым вами кодом. Более того, jQuery легко расширяется. Для нее было разработано несметное количество дополнительных модулей, от проверки приемлемости данных средствами JavaScript и до индикаторов выполнения.

Все пространство имен jQuery заключено внутри переменной `jQuery`, псевдонимом которой служит знак доллара (`$`). В отличие от такой библиотеки, как Prototype, jQuery не занимается расширением каких-либо исходных объектов JavaScript, в значительной степени это обуславливается стремлением избегать конфликтов с другими библиотеками.

Еще одним важным понятием в jQuery являются селекторы. Если вы знакомы с CSS, то селекторы должны стать вашей второй натурой. Все методы экземпляров в jQuery выполнены на селекторах, поэтому вместо перебора всех элементов, можно для их сбора просто воспользоваться селектором. Любые функции, вызванные в отношении селектора jQuery, будут выполнены в отношении каждого выбранного с помощью селектора элемента.

Чтобы продемонстрировать эту возможность, позвольте показать пример добавления имени класса `selected` ко всем элементам с классом `foo`. Первый пример будет на чистом JavaScript, а во втором будет использоваться jQuery:

```
// Пример на чистом JavaScript
var elements = document.getElementsByClassName("foo");
for (var i=0; i < elements.length; i++) {
  elements[i].className += " selected";
}

// Пример с использованием jQuery
$(".foo").addClass("selected");
```

Итак, вы можете убедиться в том, как API селекторов jQuery существенно сокращает объем требуемого кода. Давайте рассмотрим эти селекторы более подробно. При выборе элемента по ID в CSS используется знак решетки (#), то же самое можно делать и при использовании jQuery:

```
// Выбор элемента по ID (wem)
var element = document.getElementById("wem");
var element = $("#wem");

// Выбор всех элементов по классу (bar)
var elements = document.getElementsByClassName("bar");
var elements = $(".bar");

// Выбор всех элементов по тегу (p)
var elements = document.getElementsByTagName("p");
var elements = $("p");
```

Точно так же как в CSS, для конкретизации выбора селекторы можно комбинировать:

```
// Выбор дочерних элементов, имеющих класс 'foo' в элементах с классом 'bar'
var foo = $(".bar .foo");
```

Можно выбирать даже по атрибуту элемента:

```
var username = $("input[name='username']");
```

Или же можно выбрать первый соответствующий элемент:

```
var example = $(".wem:first");
```

Если с селектором вызывается функция, ее действие распространяется на все выбранные элементы:

```
// Добавление класса ко всем элементам с классом 'foo'
$(".foo").addClass("bar");
```

Как уже упоминалось, все функции jQuery работают в своем собственном пространстве имен, поэтому если функция вызывается непосредственно для DOM-элемента, она работать не будет:

```
// Этот код даст сбой!
var element = document.getElementById("wem");
element.addClass("bar");
```

Вместо этого, если вам нужно воспользоваться API библиотеки jQuery, элемент нужно заключить в экземпляр `jQuery`:

```
var element = document.getElementById("wem");
$(element).addClass("bar");
```

Обход элементов DOM-модели

После выбора элементов jQuery предоставляет вам несколько способов поиска элементов, имеющих отношение к элементам в селекторе:

```
var wem = $("#wem");

// Нахождение дочерних элементов в области видимости
wem.find(".test");

// Выбор непосредственного родительского элемента
wem.parent();

// Или получение массива родительских элементов в области видимости
дополнительного
// селектора
wem.parents(".optionalSelector");

// Выбор непосредственных потомков (первого элемента)
wem.children();
```

Или же можно совершить обход элементов внутри селектора:

```
var wem = $("#wem");

// Возвращение элемента с указанным индексом (0)
wem.eq( 0 );

// Возвращение первого элемента (эквивалент выражения $.fn.eq(0))
wem.first();

// Сокращение количества элементов до тех, которые соответствуют селектору
(".foo")
wem.filter(".foo");

// Сокращение количества элементов до тех, которые проходят функцию
// тестирования
wem.filter(function(){
  // this указывает на текущий элемент
  return $(this).hasClass(".foo");
});

// Сокращение количества элементов до тех, у которых есть потомки,
соответствующие
// селектору (".selected")
wem.has(".selected");
```

В jQuery есть итераторы `map()` и `each()`, допускающие применение функции обратного вызова:

```
var wem = $("#wem");

// Передача функции каждого элемента, создавая новый массив на основе
// возвращаемых значений
wem.map(function(element, index){
  assertEqual(this, element);

  return this.id;
});

// Применение функции обратного вызова ко всем выбранным элементам, эквивалент
// цикла `for`.
wem.each(function(index, element){
  assertEqual(this, element);

  /* ... */
});
```

Можно также добавить элементы к селектору самостоятельно:

```
// Добавление к селектору всех элементов с тегом p
var wem = $("#wem");
wem.add( $("p") );
```

Работа с DOM

Но jQuery не ограничивается одними селекторами, в ней есть мощный API для работы и взаимодействия с DOM. В дополнение к селекторам конструктору jQuery можно передавать HTML-теги, которые можно использовать для создания новых элементов:

```
var element = $("p");
element.addClass("bar")
element.text("Некое содержимое ");
```

Добавление нового элемента к DOM осуществляется очень просто, для этого нужно воспользоваться одной из функций: `append()` или `prepend()`. Чтобы не снижать производительности, все манипуляции с созданными элементами лучше производить до их присоединения к DOM:

```
// Добавление элемента
var newDiv = $("<div />");
$("body").append(newDiv);

// Добавление элемента в качестве первого дочернего
$(".container").prepend($("<hr />"));
```

Можно вставить элемент перед другим элементом или за ним:

```
// Вставка элемента за другим элементом
$(".container").after( $("<p />") );

// Вставка элемента перед другим элементом
$(".container").before( $("<p />") );
```

Так же просто осуществляется удаление элементов:

```
// Удаление элементов
$("wem").remove();
```

А как насчет изменения атрибутов элементов? В jQuery имеется поддержка и этих действий. Например, вы можете добавить имена класса, используя функцию `addClass()`:

```
$("#foo").addClass("bar");

// Удаление класса
$("#foo").removeClass("bar");

// Если ли у элемента этот класс?
var hasBar = $("#foo").hasClass("bar");
```

Так же просто осуществляется установка и извлечение стиля CSS. Функция `css()` работает и как извлекатель, и как установщик, в зависимости от типа переданных ей аргументов:

```
var getColor = $(".foo").css("color");

// Установка цветового стиля
$(".foo").css("color", "#000");

// Или передача хэша для установки сразу нескольких стилей
$(".foo").css({color: "#000", backgroundColor: "#FFF"});
```

Для наиболее часто используемых изменений стиля в jQuery имеется несколько сокращений:

```
// Установка атрибута display в none, чтобы скрыть элементы
$(".bar").hide();

// Установка атрибута display в block, для показа элементов
$(".bar").show();
```

Или, если нужно получить медленное изменение атрибута прозрачности:

```
$(".foo").fadeOut();
$(".foo").fadeIn();
```

Имеющиеся в jQuery функции-извлекатели и функции-установщики CSS не ограничиваются. Можно, например, установить содержимое элементов, используя функцию html():

```
// Извлечение HTML первого элемента в селекторе
var getHTML = $("#bar").html();

// Установка HTML для выбранных элементов
$("#bar").html("<p>Hi</p>");
```

То же самое распространяется и на функцию text(), но без тегов:

```
var getText = $("#bar").text();
$("#bar").text("Содержимое, представленное обычным текстом");
```

И наконец, для удаления всех дочерних элементов, используется функция empty():

```
$("#bar").empty();
```

События

У обработки событий в браузерах сложилась бурная история, вылившаяся в несовместимые API. Библиотека jQuery решает для вас эту проблему, сглаживая все различия браузеров и предоставляя замечательный API. Рассмотрим краткий обзор обработки событий в jQuery, а более полную информацию можно получить в главе 2 и в официальной документации (http://api.jquery.com/category/events).

Чтобы добавить обработчик события, нужно воспользоваться функцией bind(), передавая ей тип события и функцию обратного вызова:

```
$(".clicky").bind("click", function(event){
  // Выполняется по щелчку
});
```

Для наиболее востребованных событий в jQuery предоставляются сокращения, поэтому вместо вызова bind() можно, к примеру, использовать следующий код:

```
$(".clicky").click(function(){ /* ... */ });
```

Одним из событий, которое, скорее всего, будет востребовано, является document.ready. Оно инициируется в процессе загрузки страницы, когда DOM уже готова, но перед тем, как будут загружены такие элементы, как изображения. В jQuery для этого события предоставляется весьма изящное сокращение: функцию просто нужно передать непосредственно объекту jQuery:

```
jQuery(function($){
  // Выполняется при наступлении события document.ready
});
```

У новичков jQuery часто возникает путаница с изменением контекста внутри функций обратного вызова. Например, в показанном выше примере контекст

функции обратного вызова изменяется для ссылки на элемент, в данном случае это элемент `$(".clicky")`:

```
$(".clicky").click(function(){
  // 'this' является эквивалентом цели события
  assert( $(this).hasClass(".clicky") );
});
```

При использовании `this` в функции обратного вызова возникает проблема изменения контекста. Весьма распространен прием сохранения контекста в локальной переменной, которую часто называют `self`:

```
var self = this;
$(".clicky").click(function(){
  self.clickedClick();
});
```

Кроме этого, можно заключить функцию обратного вызова в прокси-функцию, воспользовавшись `jQuery.proxy()`:

```
$(".clicky").click($.proxy(function(){
  // Context isn't changed
}, this));
```

Более подробно делегирование событий и контекст рассмотрены в главе 2.

Ajax

Ajax, или XMLHttpRequest — еще одно свойство, у которого есть широкий спектр различных реализаций в браузерах. В данном случае jQuery снова абстрагируется от этих реализаций, сглаживая любые различия и предоставляя вам превосходный API. Более подробно имеющийся в jQuery Ajax API рассмотрен в главе 3, а здесь будет дан лишь краткий обзор.

В jQuery имеется низкоуровневая функция `ajax()` и несколько ее высокоуровневых абстракций. Функции `ajax()` передается хэш необязательных аргументов, таких как конечный URL, тип запроса и функция обратного вызова в случае успеха (`success`):

```
$.ajax({
  url: "http://example.com",
  type: "GET",
  success: function(){ /* ... */ }
});
```

Но имеющиеся в jQuery сокращения делают API еще более лаконичным:

```
$.get("http://example.com", function(){ /* при успехе */ })
$.post("http://example.com", {some: "data"});
```

Используемый в jQuery необязательный аргумент `dataType` сообщает jQuery, как нужно распорядиться Ajax-ответами. Если он не предоставлен, jQuery выстроит догадку на основе заголовка типа данных, имеющегося в ответе. Если вы знаете, каким будет ответ, лучше установить тип данных явным образом:

```
// Запрос данных в формате JSON
$.ajax({
  url: "http://example.com/endpoint.jso",
  type: "GET",
  dataType: "json",
  success: function(json){ /* ... */ }
});
```

jQuery также предоставляет сокращения для самых распространенных типов данных, например `getJSON()`, которое эквивалентно показанной выше функции `ajax()`:

```
$.getJSON("http://example.com/endpoint.json", function(json){ /* .. */ });
```

За более глубоким анализом необязательных аргументов в jQuery Ajax API обратитесь к главе 3, а также к дополнительной документации.

Исполнение роли законопослушной гражданки

Библиотека jQuery гордится тем, что является законопослушной веб-гражданкой. В качестве таковой она работает целиком в своем собственном пространстве имен и не засоряет глобальную область видимости. Но объект `jQuery` использует в качестве псевдонима символ `$`, который часто используется другими библиотеками, такими как Prototype. Поэтому, чтобы уберечь библиотеки от конфликтов, нужно воспользоваться режимом jQuery `noConflict` для изменения псевдонима и освобождения символа `$`:

```
var $J = jQuery.noConflict();

assertEqual( $, undefined );
```

При написании jQuery-расширений нужно предполагать, что был включен режим избегания конфликтов jQuery и что символ `$` не ссылается на `jQuery`. Но на практике символ `$` является весьма полезным сокращением, поэтому нужно просто обеспечить его использование в статусе локальной переменной:

```
(function($){

  // $ является локальной переменной
  $(".foo").addClass("wem");

})(jQuery);
```

Чтобы упростить ситуацию, jQuery также передаст ссылку на себя с событием document.ready:

```
jQuery(function($){
  // Выполняется при загрузке страницы
  assertEqual( $, jQuery );
});
```

Расширения

Процесс создания расширения jQuery, как говорится, проще некуда. Если нужно добавить функции работы с классами, следует просто создать функцию в отношении объекта jQuery:

```
jQuery.myExt = function(arg1){ /*...*/ };

// А затем, для ее использования
$.myExt("anyArgs");
```

Если нужно добавить функции экземпляров, доступные в отношении селектора элементов, следует просто установить функцию в отношении объекта jQuery.fn, являющегося псевдонимом для jQuery.prototype. В соответствии со сложившейся практикой, в конце расширения нужно вернуть контекст (т. е. this), что позволит осуществлять выстраивание функций в цепочку:

```
jQuery.fn.wemExt = function(arg1){
  $(this).html("Bar");
  return this;
};

$("#element").wemExt(1).addClass("foo");
```

Также сложилась практика инкапсулировать расширения с использованием схемы модуля, предотвращающей любые утечки области видимости и конфликты переменных. Заключайте свои расширения в безымянные функции, сохраняя локальность всех переменных:

```
(function($){
  // Здесь содержится локальный контекст
  var replaceLinks = function(){
    var re =
    /((http|https|ftp):\/\/[\w?=&.\/-;#~%-]+(?![\w\s?&.\/;#~%"=-]*>))/g;
    $(this).html(
     $(this).html().replace(re, '<a target="_blank" href="$1">$1</a> ')
    );
  };

  $.fn.autolink = function() {
    return this.each(replaceLinks);
  };
})(jQuery);
```

Создание дополнительного модуля jQuery Growl

Применим наши знания jQuery на практике и создадим библиотеку Growl. Для тех, кто не знаком с Growl, это библиотека уведомлений для Mac OS X, которую приложения могут использовать для ненавязчивого показа сообщений на рабочем столе. Мы собираемся неким образом имитировать библиотеку OS X и показывать сообщения из JavaScript на странице, как продемонстрировано на рис. А.1.

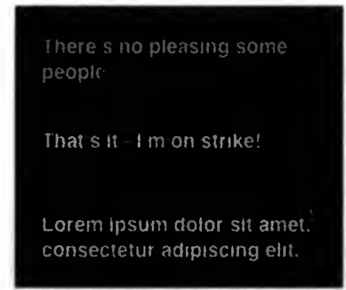

Рис. А.1. Пример Growl-сообщений

Первым делом нужно создать div-контейнер #container, из которого будут опускаться все наши элементы сообщений. Можно увидеть, что мы включили обе библиотеки, и jQuery, и jQuery UI, вторая из которых будет позже использована для добавления нескольких эффектов. Когда страница загружается, будем добавлять к ней div-контейнер:

```
//= require <jquery>
//= require <jquery.ui>

(function($){
  var container = $("<div />");
  container.attr({id: "growl"});

  $(function(){
    // Добавление div при загрузке страницы
    $("body").append(container);
  });

  /* ... */
})(jQuery);
```

Теперь про логику дополнительного модуля. Как только у нас будет новое сообщение, мы добавляем div к элементу container. Мы добавляем к сообщению эффект

ниспадания (drop), а затем, через какой-то период времени, оно плавно выцветает и удаляется, имитируя поведение Growl в OS X:

```
$.growl = function(body){
  // Создание div-контейнера для Growl
  var msg = $("<div />").addClass("msg");
  msg.html(body);

  // Добавление его к списку
  container.append(msg);

  // Добавление эффекта ниспадания с последующим удалением
  msg.show("drop", {
    direction: "down",
    distance: 50
  }, 300).
    delay(2000).
    fadeOut(300, function(){
      $(this).remove();
    });

  return msg;
};
```

И это весь требуемый код JavaScript. На данный момент он выглядит несколько неприглядно, поэтому мы можем его немного приукрасить с помощью CSS3. Нам нужно получить абсолютное позиционирование div-контейнера #container в нижнем правом углу страницы:

```
#growl {
  position: absolute;
  bottom: 10px;
  right: 20px;
  overflow: hidden;
}
```

Теперь придадим стилевое оформление элементам сообщений. Мне очень нравится тема HUD Growl, поэтому давайте попробуем ее сымитировать. Сделаем фон слегка прозрачным, используя rgba, а затем добавим прямоугольную тень (inset box-shadow), предоставив элементу появление источника света:

```
#growl .msg {
  width: 200px;
  min-height: 30px;
  padding: 10px;
  margin-bottom: 10px;

  border: 1px solid #171717;
  color: #E4E4E4;
  text-shadow: 0 -1px 1px #0A131A;
```

продолжение ⇗

```
  font-weight: bold;
  font-size: 15px;

  background: #141517;
  background: -webkit-gradient(
    linear, left top, left bottom,
    from(rgba(255, 255, 255, 0.3)),
    color-stop(0.8, rgba(255, 255, 255, 0))),
    rgba(0, 0, 0, 0.8);

  -webkit-box-shadow: inset 0 1px 1px #8E8E8E;
  -moz-box-shadow: inset 0 1px 1px #8E8E8E;
  box-shadow: inset 0 1px 1px #8E8E8E;

  -webkit-border-radius: 7px;
  -moz-border-radius: 7px;
  border-radius: 7px;
}
```

Вот и все. Вы видите, что создавать дополнительные модули для jQuery совсем нетрудно. Как и в случае с другими примерами, полную версию исходного кода можно найти в файле assets/appA/growl.html.

Б CSS-расширения

По словам его автора, Алексиса Селлье (Alexis Sellier), Less (http://lesscss.org) является «динамическим языком таблиц стилей, который надстраивается поверх синтаксиса CSS». Less является надмножеством CSS, расширяющим эту спецификацию переменными, миксинами, операциями и вложенными правилами.

Польза от него заключается в существенном сокращении объема набираемого CSS, особенно когда дело доходит до правил CSS3, специфичных для того или иного производителя. Затем вы можете скомпилировать свои Less-файлы в чистый CSS.

Иными словами, вместо написания следующего кода:

```
.panel {
background: #CCC;
background: -webkit-gradient(linear, left top, left bottom, from(#FFF),
to(#CCC));
background: -moz-linear-gradient(top, #FFF, #CCC);
}
```

вы можете написать этот код:

```
.panel {
  .vbg-gradient(#FFF, #CCC);
}
```

Переменные

Если вы повторно используете атрибуты цветового оформления и правил, применение Less-переменных позволит объединить их в одном месте, разрешая вносить глобальные изменения без применения операций поиска и замены.

Определение переменной выглядит довольно просто:

```
@panel-color: #CCC;
```

Затем ее можно использовать внутри ваших стилевых правил:

```
header {
  color: @panel-color;
}
```

Миксины

Less-миксины ведут себя во многом похоже на макросы языка С. В сущности, вы определяете миксин, которому можно передать дополнительные аргументы примерно следующим образом:

```
.vbg-gradient(@fc: #FFF, @tc: #CCC) {
  background: @fc;
  background: -webkit-gradient(linear, left top, left bottom, from(@fc),
              to(@tc));
  background: -moz-linear-gradient(top, @fc, @tc);
  background: linear-gradient(top, @fc, @tc);
}
```

В показанном выше примере передаются два аргумента: `fc` и `tc`, имеющие значения по умолчанию `#FFF` и `#CCC` соответственно. Затем это вставляется в содержимое класса. Это нужно представлять как определение переменной, но для целого класса.

Поскольку для CSS3 процесс определения стандарта еще не завершен, производители браузеров, как правило, определяют свои собственные префиксы, такие как `-webkit` и `-moz`. И это, в некотором смысле, очень хорошо, поскольку мы можем сразу же приступить к использованию свойств; но зачастую приходится использовать избыточный синтаксис в силу необходимости определения стилей два или три раза для разных браузеров.

Как вы уже, наверное, догадались, Less действительно может сократить объем набираемого кода, нужно только лишь превратить стили, специфичные для того или иного производителя, в миксин.

Вот еще несколько миксинов, которые могут пригодиться:

```
/* Закругленные границы */
.border-radius(@r: 3px) {
  -moz-border-radius: @r;
  -webkit-border-radius: @r;
  border-radius: @r;
}
/* Тень */
.box-shadow (@h: 0px, @v: 0px, @b: 4px, @c: #333) {
  -moz-box-shadow: @h @v @b @c;
  -webkit-box-shadow: @h @v @b @c;
  box-shadow: @h @v @b @c;
}
```

Вложенные правила

Вместо определения длинных имен селекторов для получения элементов вы можете пользоваться вложенными селекторами. Полный селектор создается

закулисно, а вложенные правила делают ваши таблицы стилей понятнее и легче для чтения:

```
button {
  .border-radius(3px);
  .box-shadow(0, 1px, 1px, #FFF);
  .vbg-gradient(#F9F9F9, #E3E3E3);

  :active {
    .vbg-gradient(#E3E3E3, #F9F9F9);
  }
}
```

Но следует все же предупредить: я бы не стал выходить за рамки двух уровней вложения, поскольку по неосторожности можно серьезно нарушить режим работы этого свойства, и ваши таблицы стилей будут выглядеть значительно хуже.

Включение других таблиц стилей

Если вы планируете разбить свою таблицу стилей на части, что я настоятельно рекомендую сделать, вы можете воспользоваться инструкцией `@import` для включения других таблиц стилей в состав текущей таблицы. Less извлечет такую таблицу и включит ее в код, что повысит производительность, поскольку клиентам не придется делать еще один HTTP-запрос.

Этот прием часто используется вместе с миксинами. Скажем, у вас есть файл с миксинами CSS3, тогда его можно импортировать следующим образом:

```
@import "utils";
```

Цвета

Это свойство появилось в Less совсем недавно, поэтому оно еще не отражено в документации, но оно настолько полезно, что вполне заслуживает упоминания. Less позволяет вам работать с цветами, используя различные функции:

```
background: saturate(#319, 10%);
background: desaturate(#319, 10%);
background: darken(#319, 10%);
background: lighten(#319, 10%)
```

На одних и тех же цветах основано множество конструкций, но в них используются разные оттенки. Несомненно, в сочетании с переменными мы сможете очень быстро создавать темы, выделяющиеся своей маркой.

Как можно воспользоваться Less?

Существует несколько различных способов компиляции кода Less в CSS.

С помощью окна командной строки

Установите gem-пакет Less, а затем вызовите команду lessc:

```
gem install less
lessc style.less
```

С помощью модуля Rack

Если используется среда на основе Rack-модуля, такая как Rails 3, есть еще более простое решение: gem-пакет rack-less. Нужно просто включить соответствующий gem-пакет в ваш Gemfile:

```
gem "rack-less"
```

и вставить промежуточное программное обеспечение в файл application.rb:

```
require "rack/less"
config.middleware.use "Rack::Less"
```

Любые таблицы стилей на языке Less, имеющиеся в каталоге /app/stylesheets, будут скомпилированы автоматически. Вы можете даже кэшировать и сжать результат, сконфигурировав rack-less в вашем файле конфигурации production.rb:

```
Rack::Less.configure do |config|
config.cache = true
config.compress = :yui
end
```

С помощью JavaScript

Разработка, похоже, задержалась на Ruby-библиотеках, но, к счастью, есть более современный вариант: библиотека Less.js, представляющая собой Less, написанный на JavaScript. Вы можете указать таблицы стилей Less на странице и включить JavaScript-файл less.js, который проведет автоматическую компиляцию:

```
<link rel="stylesheet/less" href="main.less" type="text/css">
<script src="less.js" type="text/javascript"></script>
```

Less.js работает в 40 раз быстрее Ruby-версии библиотеки. Но вам может понадобиться предварительно скомпилировать таблицы стилей Less, чтобы не снижалась производительность машин клиентов. Если установлена библиотека Node.js, вы можете провести компиляцию с помощью командной строки:

```
node bin/lessc style.less
```

Less.app

Это приложение, работающее под управлением Mac OS X, еще больше упрощает использование Less. Оно использует Less.js закулисно, и вы можете указать кон-

кретные «отслеживаемые» папки, т. е. таблицы стилей Less будут автоматически скомпилированы в CSS при их сохранении в этих папках (рис. Б.1).

Рис. Б.1. Автоматическое компилирование Less-файлов с помощью Less.app

В Справочник по CSS3

Создание привлекательных интерфейсов в CSS2.1 было задачей не из легких, потому что зачастую требовало множества дополнительных разметок, изображений и JavaScript-кода. В CSS3 предпринята попытка решить связанные с этим проблемы путем предоставления широкого спектра по-настоящему полезных атрибутов и селекторов, помогающих создавать превосходные пользовательские интерфейсы.

Часто при разработке веб-приложений я отставлял в сторону Photoshop и переходил непосредственно к CSS3 и HTML5. Теперь, при наличии столь мощных технологий, проектирование статических PSD-макетов кажется уже излишним. Клиентам также все больше нравятся эти перемены, поскольку у них появляется возможность взаимодействовать с HTML-прототипом программного продукта, получая при этом намного больше впечатлений в роли пользователя.

Я слышу ваши возражения: «А как же быть со старыми браузерами? Неужели не понятно, что времена CSS3 еще не настали?» Ответом на данный вопрос будет *постепенная деградация*. Устаревающие браузеры будут игнорировать ваши CSS3-стили, возвращаясь к стандартным установкам. Например, в Chrome ваши пользователи будут видеть приложение во всем его великолепии, с плавными переходами и со всеми другими особенностями, а в IE7 приложение, конечно же, будет работать, но существенно потеряв при этом свою привлекательность.

Что же касается Internet Explorer 6, я настоятельно советую вообще отказаться от его поддержки, как это уже начали делать Facebook, Amazon и Google, а незначительный процент пользователей IE6 не оправдывает усилия по поддержке этого браузера.

Основными браузерами считаются IE, Firefox, Chrome и Safari. Chrome и Safari работают на разных движках JavaScript, но используют одно и то же средство визуализации — WebKit. Хотя между этими двумя браузерами существуют небольшие различия — они используют разные графические библиотеки, — какие-то доработки для Chrome вызывают ответную реакцию в WebKit, и наоборот.

Ко времени написания данной книги Microsoft выпустила IE9. Надеюсь, в скором времени этот браузер найдет широкое применение, поскольку он намного более совершенен, чем предыдущие браузеры этой компании, и включает существенную поддержку CSS3.

Для разработчиков внешних интерфейсов настают очень интересные времена, и вы непременно должны сориентироваться на использование этих новых технологий.

Префиксы

Производители браузеров приступили к реализации CSS3 еще до окончательного утверждения стандарта. Поэтому, пока синтаксис еще окончательно не устоялся, некоторые CSS3-стили имеют специфичный для браузера префикс. Например, градиентный стиль CSS3 в Firefox и в Safari имеет разный вид: Firefox использует `-moz-linear-gradient`, а Safari (WebKit) использует `-webkit-gradient`; оба синтаксиса имеют префикс, указывающий на производителя браузера.

В данный момент используются следующие префиксы:

- Chrome: `-webkit-`
- Safari: `-webkit-`
- Firefox: `-moz-`
- IE: `-ms-`
- Opera: `-o-`

Какое-то время нужно будет указывать стили с префиксом производителя, а затем указывать их без этого префикса. Тем самым будет гарантироваться, что когда браузеры удалят префикс и перейдут на ставшую стандартом CSS3-спецификацию, ваши стили будут работать, как и раньше:

```
#prefix-example {
  -moz-box-shadow: 0 3px 5px #FFF;
  -webkit-box-shadow: 0 3px 5px #FFF;
  box-shadow: 0 3px 5px #FFF;
}
```

Цвета

CSS3 предоставляет вам несколько новых способов указания цветовых настроек, включая альфа-прозрачность (alpha transparency).

Старый способ задания прозрачности цветов заключался в использовании фоновых изображений размером 1 × 1 пиксел, но теперь все это можно оставить в прошлом.

`Rgb`-стиль позволяет указывать цвета в виде красного, зеленого и синего заполнения, в основных цветах, а не в традиционных шестнадцатеричных значениях. С помощью веб-инспектора Safari можно легко конвертировать одно значение в другое, нужно лишь щелкнуть на цвете в разделе **Styles**.

Показанный ниже пример является эквивалентом шестнадцатеричного значения #FFF, означающего белый цвет:

```
#rgb-example {
  //    rgb(красный, зеленый, синий);
  color: rgb(255, 255, 255);
}
```

Можно также воспользоваться hsl-стилем, означающим тон (hue), насыщенность (saturation) и яркость (lightness).

HSL передаются три значения:

Hue

Угол на цветовом диске; 0 (или 360) означает красный, 120 — зеленый, 240 — синий. Значения между ними соответствуют разным оттенкам.

Saturation

Процентное значение; 100% показывает полный цвет.

Lightness

Процентное значение; 0% означает темный (черный) цвет, 100% — светлый (белый) цвет, а 50% означает среднюю яркость.

Добавление к rgb или к hsl альфа-прозрачности осуществляется довольно просто, для этого используются, соответственно, функции rgba и hsla. Альфа-прозрачность указывается числом между 0 (прозрачно) и 1 (непрозрачно).

```
#alpha-example {
  background: hsla(324, 100%, 50%, .5);
  border: 1em solid rgba(0, 0, 0, .3);
  color: rgba(255, 255, 255, .8);
}
```

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: полная поддержка
- Opera: полная поддержка
- Safari: полная поддержка
- IE: полная поддержка
- Opera: полная поддержка

Скругленные углы

В CSS2.1 скругленные углы давались крайне нелегко, требуя зачастую большого объема дополнительной разметки, нескольких изображений и даже JavaScript-кода.

Теперь все стало гораздо проще — нужно лишь воспользоваться стилем border-radius. Как и в случае использования стилей padding и margin, можно указать

несколько радиусов, предназначенных для разных углов, два значения, предназначенные для радиусов по горизонтали и по вертикали, или один радиус для всех углов. Предоставляя достаточно большие значения радиуса, можно даже создать окружность:

```
border-radius: 20px;

// горизонталь, вертикаль
border-radius: 20px 20px;

// верхний левый, верхний правый, нижний правый, нижний левый
border-radius: 20px 20px 20px 20px;
```

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: полная поддержка
- Safari: с `-webkit-`
- IE >= 9.0: полная поддержка
- Opera: полная поддержка

Отбрасываемые тени

До появления CSS3 многих не волновали отбрасываемые тени, поскольку на пути их создания было слишком много препятствий. Но теперь CSS3 дает вам стиль `box-shadow`, который превращает их реализацию в сущий пустяк. Только не переусердствуйте, пуская пыль в глаза пользователю, поскольку отбрасываемые тени могут повлиять на производительность системы.

Стилю `box-shadow` передаются несколько параметров: смещение по горизонтали, смещение по вертикали, радиус размытия, необязательный параметр расстояния распространения и цвет. При предоставлении параметра `inset` тень будет нарисована внутри элемента, в противном случае она будет нарисована снаружи, как это и делается по умолчанию. Можно также включить несколько теней, отделив их друг от друга запятыми, как в показанном ниже примере:

```
// смещение по горизонтали, смещение по вертикали, радиус размытия, цвет
box-shadow: 10px 5px 15px #000;

// внутренние тени
box-shadow: 10px 5px 15px #000 inset;

// смещение по горизонтали, смещение по вертикали, радиус размытия,
// расстояние распространения, цвет
box-shadow: 10px 5px 15px 15px #000;

// несколько теней
box-shadow: 0 1px 1px #FFF inset, 5px 5px 10px #000;
```

Дизайнеры часто в своих работах определяют источник света, который придает интерфейсу осязаемость и интерактивность. Вы без особого труда можете добиться этого с помощью `box-shadow` — нужно просто указать однопиксельную белую внутреннюю тень. В данном случае свет будет исходить из верхней части страницы; нужно сохранять однообразие во всех наших стилях:

```
#shadow-example {
  -moz-box-shadow: 0 1px 1px #FFF inset;
  -webkit-box-shadow: 0 1px 1px #FFF inset;
  box-shadow: 0 1px 1px #FFF inset;
}
```

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: с `-webkit-`
- Safari: с `-webkit-`
- IE >= 9.0: полная поддержка
- Opera: полная поддержка

Тени для текста

До появления CSS3 единственный способ добиться тени для текста заключался в замене текста изображениями, что было весьма трудоемкой работой. CSS3 позволяет добавить тени к тексту с помощью стиля `text-shadow`. Нужно просто передать этому стилю смещение по горизонтали, смещение по вертикали, необязательный радиус размытия и цвет:

```
// смещение по горизонтали, смещение по вертикали, цвет
text-shadow: 1px 1px #FFF;

// смещение по горизонтали, смещение по вертикали, радиус размытия, цвет
text-shadow: 1px 1px .3em rgba(255, 255, 255, .8);
```

Тени для текста отличаются от прямоугольных теней (`box-shadow`), поскольку в них нет поддержки расстояния распространения или внутренних теней. Но вы можете заставить поверить в наличие внешней или внутренней тени за счет позиции тени. Если у тени отрицательное смещение по вертикали и она находится над текстом, то она смотрится как внутренняя. Соответственно, если тень ниже текста, то она смотрится как внешняя.

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: полная поддержка
- Safari: полная поддержка
- IE: не поддерживает
- Opera: полная поддержка

Градиенты

Ранее градиенты создавались с помощью повторяющихся фоновых изображений. Это означало, что у них была фиксированная ширина или высота, и для внесения изменений приходилось открывать графический редактор.

В CSS3 добавлена поддержка линейных и радиальных градиентов, что является одной из наиболее полезных особенностей этой спецификации. Для создания градиентов нужно вызвать несколько CSS-функций, которые можно использовать везде, где обычно используется цвет.

Для получения линейных градиентов нужно просто передать функции `linear-gradient` список цветов, через которые нужно осуществить переход:

```
linear-gradient(#CCC, #DDD, white)
```

По умолчанию градиенты имеют вертикальный характер, но эту установку можно изменить, передав позицию:

```
// горизонтальный градиент
linear-gradient(left, #CCC, #DDD, #FFF);

// или указав угол
linear-gradient(-45deg , #CCC , #FFF)
```

Если нужно еще и проконтролировать место начала градиента, можно воспользоваться остановками цвета. Для этого нужно вместе с цветом указать процентное или пиксельное значение:

```
linear-gradient(white , #DDD 20% , black)
```

Можно также задать переход к прозрачности или от нее:

```
radial-gradient( rgba(255, 255, 255, .8) , transparent )
```

В Safari в настоящее время используется совершенно другой синтаксис. Вскоре он будет приведен к стандарту, а пока его нужно использовать следующим образом:

```
// -webkit-gradient(<тип>, <позиция> [, <пределы>]?, <позиция> [, <пределы>]?
// [, <остановка>]*);
-webkit-gradient(linear, left top, left bottom,
from(#FFF), color-stop(10%, #DDD), to(#CCC));
```

Хотя большинство браузеров поддерживают CSS-стандарты градиента, каждый из них требует указания префикса с именем своего производителя:

- Firefox: с `-moz-`
- Chrome: с `-webkit-`
- Safari: альтернативная реализация
- IE >= 10: с `-ms-`
- Opera >= 11.1: с `-o-`

Поэтому градиенты для кроссбраузерной работы имеют следующий вид:

```
#gradient-example {
  /* Нейтрализация */
  background: #FFF;
  /* Chrome < 10, Safari < 5.1 */
 background: -webkit-gradient(linear, left top, left bottom,
             from(#FFF), to(#CCC));
  /* Chrome >= 10, Safari >= 5.1 */
  background: -webkit-linear-gradient(#FFF, #CCC);
  /* Firefox >= 3.6 */
  background: -moz-linear-gradient(#FFF, #CCC);
  /* Opera >= 11.1 */
  background: -o-linear-gradient(#FFF, #CCC);
  /* IE >= 10 */
  background: -ms-linear-gradient(#FFF, #CCC);
  /* Стандарт */
  background: linear-gradient(#FFF, #CCC);
}
```

Получается довольно многословный код! Но есть такие упрощающие код проекты, как Less и Sass, подробности применения которых будут рассмотрены далее.

Составной фон

В CSS3 можно указать не только несколько теней, но и составной фон. Раньше, чтобы получить множество фоновых изображений приходилось создавать большое количество вложенных элементов, т. е. создавать слишком много лишней разметки. CSS3 позволяет предоставить для фонового стиля список с запятой в качестве разделителя, что существенно сокращает объем требуемой разметки:

```
background: url(snowflakes.png) top repeat-x,
  url(chimney.png) bottom no-repeat,
  -moz-linear-gradient(white, #CCC),
  #CCC;
```

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: полная поддержка
- Safari: полная поддержка
- IE >= 9.0: полная поддержка
- Opera: полная поддержка

Селекторы

CSS3 предоставляет целый набор новых селекторов для указания элементов:

`:first-child`
: Выбор первого элемента в селекторе

`:last-child`
: Выбор последнего элемента в селекторе

`:only-child`
: Выбор элементов только с одним дочерним элементом

`:target`
: Выбор элементов, указанных после символа решетки в текущем URL

`:checked`
: Выбор установленных флажков

Ниже перечислены те селекторы, которые я счел нужным рассмотреть более подробно.

N-й дочерний элемент

Селектор `:nth-child` позволяет придавать альтернативное стилевое оформление каждому n-му дочернему элементу. Например, этот код выбирает каждый третий дочерний элемент:

```
#example:nth-child( 3n ) { /* ... */ }
```

Этим можно воспользоваться для выбора четных или нечетных дочерних элементов:

```
/* Четный дочерний элемент */
#example:nth-child( 2n )   { /* ... */ }
#example:nth-child( even ) { /* ... */ }

/* Нечетный дочерний элемент */
#example:nth-child( 2n+1 ) { /* ... */ }
#example:nth-child( odd )  { /* ... */ }
```

Можно также задать в селекторе обратный отсчет:

```
/* Последний дочерний элемент */
#example:nth-last-child( 1 )
```

Фактически `:first-child` является эквивалентом селектору `:nth-child(1)`, а `:last-child` — эквивалентом селектору `:nth-last-child(1)`.

Прямой потомок

Можно ограничить действие селектора только дочерними элементами, являющимися прямыми потомками, для чего нужно воспользоваться символом «больше чем» (>):

```
/* Только div-элементы, являющиеся прямыми потомками */
#example > div { }
```

Обращение селекторов

Вы можете воспользоваться обращением селекторов, воспользовавшись конструкцией :not, которой можно передать простой селектор. В настоящее время обращение для более сложных селекторов, например для p:not(h1 + p), не поддерживается:

```
/* Только прямые потомки, за исключением тех, у которых атрибут class имеет
   значение "current" */
#example > *:not(.current) {
  display: none
}
```

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: полная поддержка
- Safari: полная поддержка
- IE >= 9.0: полная поддержка
- Opera: полная поддержка

Переходы

В CSS3 добавлена поддержка переходов, позволяющая за счет изменения стиля создавать простую анимацию. Для этого нужно передать продолжительность, свойство и дополнительно тип анимации для используемого свойства. Продолжительность можно задавать в секундах (s) или миллисекундах (ms):

```
/* продолжительность, свойство, тип анимации (необязательно) */
transition: 1.5s opacity ease-out

/* Несколько переходов */
transition: 2s opacity , .5s height ease-in
transition: .5s height , .5s .5s width
```

В первом примере при изменении непрозрачности (скажем, применяется встроенный стиль), будет анимирована замена исходного значения на новое.

Существуют различные типы функций распределения анимации по времени (timing functions):

- linear
- ease-in
- ease-out
- ease-in-out

Можно также указать свою собственную последовательность распределения по времени, используя кубическую кривую Безье, которая описывает скорость анимации, как в следующей анимации, имитирующей отскоки.

```
#transition-example {
  position: absolute;
  /* cubic-bezier(x1, y1, x2, y2) */
  transition: 5s left cubic-bezier(0.0, 0.35, .5, 1.3);
}
```

В Safari и Chrome, когда переход завершается, в отношении элемента инициируется событие `WebKitTransitionEvent`. В Firefox такое же событие называется `transitionend`. К сожалению, в отношении использования CSS3-переходов существует ряд предостережений: проигрывание анимации является практически неуправляемым процессом и не все значения могут использоваться в переходах. Переходы очень полезны для простых приемов анимации, и некоторые браузеры (например, Safari) даже используют для них аппаратное ускорение:

```
#transition-example {
  width: 50px;
  height: 50px;
  background: red;
  -webkit-transition: 2s background ease-in-out;
  -moz-transition: 2s background ease-in-out;
  -o-transition: 2s background ease-in-out;
  transition: 2s background ease-in-out;
}

#transition-example:hover {
  background: blue;
}
```

По той или иной причине переходы между градиентами можно применять только в том случае, если хотя бы один из них имеет хоть какую-нибудь степень альфа-прозрачности. Можно также задавать переход между некоторыми значениями, например от высоты `height:0` до высоты `height:auto`.

Поддержка осуществляется браузерами:

- Firefox: с `-moz-`
- Chrome: с `-webkit-`
- Safari: с `-webkit-`
- IE >= 10.0: с `-ms-`
- Opera: с `-o-`

Создание границ с помощью изображения

Свойство `border-image` позволяет создавать границы элемента с помощью изображения. Первый аргумент указывает URL изображения, следующие аргументы дают описание порядка разбиения изображения на части. Последние аргументы задают значения растягивания, описывают масштабирование и мозаичность

частей для сторон и середины. Доступными значениями для растягивания являются round (повторение целого количества раз с растяжением остатка), repeat (повторение) и stretch (растяжение):

```
border-image: url(border.png) 14 14 14 14 round round;

border-image: url(border.png) 14 14 14 14 stretch stretch;
```

Поддержка осуществляется браузерами:

- Firefox: с -moz-
- Chrome: с -webkit-
- Safari: с -webkit-
- IE: не поддерживает
- Opera: с -o-

Изменения алгоритма расчета ширины и высоты элемента

Хотелось ли вам когда-нибудь создать элемент со 100% шириной, но при этом иметь поля или отступы? При использовании традиционной модели прямоугольных блоков CSS вычисляет ширину в процентах, используя ширину родительского элемента с последующим добавлением полей и отступов. Иными словами, элемент с шириной 100%, имеющий отступы, поля или границы, всегда будет выходить за рамки отведенного пространства.

Но если установить для свойства box-sizing значение border-box вместо используемого по умолчанию значения content-box, можно изменить алгоритм расчета ширины и высоты элемента, взяв в расчет границы, поля, отступы и содержимое:

```
.border-box {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
```

Это свойство хорошо поддерживается практически всеми основными браузерами, и его можно применять, не опасаясь за последствия, если только не планируется поддержка браузеров, предшествовавших Internet Explorer 8.

Преобразования

Используя CSS3, мы получаем основные 2D-преобразования, которые позволяют осуществлять перемещения, вращения, масштабирования и наклоны элементов. Например, мы можем повернуть элемент на 30 градусов против часовой стрелки:

```
transform: rotate( -30deg );
```

Также можно наклонить элемент относительно осей x и y, указав соответствующие углы:

```
transform: skew( 30deg , -10deg );
```

Позиция элемента может быть преобразована по осям x или y с помощью функций `translateX` или `translateY`:

```
translateX(30px);
translateY(50opx);
```

С помощью преобразования масштаба можно увеличить или уменьшить размер элемента. По умолчанию масштаб элемента установлен в 1:

```
transform: scale(1.2);
```

Можно указать сразу несколько преобразований, объединив их в одной строке:

```
transform: rotate(30deg) skewX(30deg);
```

Поддержка осуществляется браузерами:

- Firefox: с `-moz-`
- Chrome: с `-webkit-`
- Safari: с `-webkit-`
- IE >= 9: с `-ms-`
- Opera: с `-o-`

Модель гибких прямоугольных блоков

В CSS3 представлена модель гибких прямоугольных блоков (flexible box model), являющаяся новым способом отображения содержимого. Ее польза состоит во введении в CSS таких возможностей, которые до сей поры имелись только в таких GUI-структурах, как Adobe Flex. Так сложилось, что при желании получить горизонтальное выравнивание списка использовались свойства `floats`. Модель гибких прямоугольных блоков позволяет сделать не только это, но и многое другое. Посмотрим на код:

```
.hbox {
  display: -webkit-box;
  -webkit-box-orient: horizontal;
  -webkit-box-align: stretch;
  -webkit-box-pack: left;

  display: -moz-box;
  -moz-box-orient: horizontal;
  -moz-box-align: stretch;
  -moz-box-pack: left;
}
```

продолжение ⇗

```
.vbox {
  display: -webkit-box;
  -webkit-box-orient: vertical;
  -webkit-box-align: stretch;

  display: -moz-box;
  -moz-box-orient: vertical;
  -moz-box-align: stretch;
}
```

Мы установили для свойства `display` значение `-webkit-box` или `-moz-box`, а затем установили направление расположения дочерних элементов. По умолчанию все дочерние элементы будут растянуты так, чтобы равномерно помещаться в своем родительском элементе. Но вы можете изменить их поведение, установив атрибут `box-flex`.

Установив для `box-flex` значение `0`, вы определите, что элемент не должен растягиваться, а установив для него значение `1` или выше, определите, что элемент будет растягиваться, чтобы поместиться в доступном контенте. Например, боковая панель может иметь атрибут `flex` со значением `0`, а основное содержимое может иметь атрибут `flex` со значением `1`:

```
#sidebar {
  -webkit-box-flex: 0;
  -moz-box-flex: 0;
  box-flex: 0;
  width: 200px;
}

#content {
  -webkit-box-flex: 1;
  -moz-box-flex: 1;
  box-flex: 1;
}
```

Поддержка осуществляется браузерами:

- Firefox: с `-moz-`
- Chrome: с `-webkit-`
- Safari: с `-webkit-`
- IE >= 9: с `-ms-`
- Opera: не поддерживает

Шрифты

Свойство `@font-face` позволяет использовать для отображения текста на веб-странице указанные вами шрифты. Следовательно, вы больше не зависите от ограниченного количества системных шрифтов, установленных пользователями.

Поддерживаются форматы шрифтов TrueType и OpenType. Шрифты подпадают под политику ограничений, заключающихся в использовании одного и того же домена — файлы должны быть на том же домене, что и использующая их страница.

Значение для `@font-face` можно указать так, как показано в следующем примере, передав значение для свойства `font-family` и URL-адрес размещения шрифта:

```
@font-face {
  font-family: "Bitstream Vera Serif Bold";
  src: url("/fonts/VeraSeBd.ttf");
}
```

Затем этим можно воспользоваться, когда понадобится другой шрифт:

```
#font-example {
  font-family: "Bitstream Vera Serif Bold";
}
```

Шрифты будут загружены в асинхронном режиме и применены после завершения загрузки. Это означает, что пользователь будет видеть один из шрифтов, используемых в его системе по умолчанию, пока не будет загружен тот шрифт, который вы определили. Поэтому неплохо было бы определить еще какой-нибудь нейтральный шрифт, доступный на локальной системе.

Поддержка осуществляется браузерами:

- Firefox: полная поддержка
- Chrome: полная поддержка
- Safari: полная поддержка
- IE >= 9: полная поддержка
- Opera: полная поддержка

Постепенная деградация

При правильном составлении CSS ваше приложение будет осуществлять постепенную деградацию. Оно будет работать в браузерах, не поддерживающих CSS3, потеряв при этом часть своей привлекательности.

Основой для постепенной деградации является то обстоятельство, что браузеры игнорируют все, что им непонятно, например CSS-свойства, значения и селекторы. CSS-свойства отменяют друг друга, поэтому если какое-нибудь свойство определено дважды в одном и том же правиле, первое свойство будет отменено. Сначала нужно объявлять свойство, совместимое с CSS2.1, чтобы оно было отменено, если поддерживается `rgba`:

```
#example-gd {
  background: white;
  background: rgba(255, 255, 255, .75);
}
```

А что происходит с префиксами производителей? Здесь действуют те же правила. Нужно просто включить в код префиксы для каждого браузера, а браузер будет использовать один из тех, который ему понятен. Версию без префикса нужно ставить на последнее место, поскольку она будет использоваться, когда браузерная поддержка CSS3 станет стандартом и префиксы будут отменены:

```
#example-gd {
  background: #FFF;
  background: -webkit-gradient(linear, left top, left bottom,
              from(#FFF), to(#CCC));
  background: -webkit-linear-gradient(#FFF, #CCC);
  background: -moz-linear-gradient(#FFF, #CCC);
  background: linear-gradient(#FFF, #CCC);
}
```

Modernizr

Библиотека Modernizr (http://www.modernizr.com) определяет поддержку различных CSS3-свойств, позволяя в таблице стилей нацелиться на специфическое поведение браузера:

```
.multiplebgs div p {
  /* свойства для браузеров,
     поддерживающих составной фон */
}
.no-multiplebgs div p {
  /* дополнительные нейтральные свойства для
     браузеров, не поддерживающих составной фон */
}
```

Библиотека Modernizr определяет существование поддержки следующих свойств:

- `@font-face`
- `rgba()`
- `hsla()`
- `border-image:`
- `border-radius:`
- `box-shadow:`
- `text-shadow:`
- Составной фон (Multiple backgrounds)
- Модель гибких прямоугольных блоков (Flexible box model)
- CSS-анимация
- CSS-градиенты
- CSS 2D-преобразования
- CSS-переходы

Для просмотра полного списка или для загрузки библиотеки Modernizr нужно зайти на страницу проекта (http://www.modernizr.com).

Использовать Modernizr очень легко, нужно просто включить файл JavaScript и добавить к `<html>`-тегу `class`-атрибут со значением `no-js`:

```
<script src="/javascripts/modernizr.js"></script>
<html class="no-js">
```

Затем Modernizr добавит некоторые классы к `<html>`-тегу, которые можно использовать в ваших селекторах для нацеливания на определенное поведение браузера:

```
/* Альтернативная разметка при недоступности модели гибких прямоугольных
блоков */
.no-flexbox #content {
  float: left;
}
```

Расширение Google Chrome Frame

Google Chrome Frame (GCF) является расширением Internet Explorer, позволяющим переключать механизм отображения страниц IE на механизм Google Chrome под названием Chromium (http://www.chromium.org).

После установки расширения можно разрешить использование GCF с помощью `meta`-тега в заголовке страницы:

```
<meta http-equiv="X-UA-Compatible" content="chrome=1">
```

Или поступить по-другому и добавить настройки в заголовок ответа:

```
X-UA-Compatible: chrome=1
```

Вот и все, что требуется для включения GCF-отображения для вашей веб-страницы. Но у GCF есть еще и дополнительные возможности, такие как предложение пользователям установить его, если они запускают IE (а это расширение еще не установлено). Предложение может просто накладываться на верхнюю часть страницы и обновляться автоматически при установке GCF, при этом браузер перезапускать не потребуется.

Сначала нужно включить JavaScript-код для GCF:

```
<script src="http://ajax.googleapis.com/ajax/libs/chrome-frame/1/
            CFInstall.min.js"
```

Затем в обработчике загрузки страницы или в нижней части страницы нужно вызвать функцию `CFInstall`:

```
<script>
  jQuery(function(){
    CFInstall.check({
      mode: "overlay",
    });
  });
</script>
```

Функции `CFInstall` передаются несколько необязательных аргументов:

`mode`

Режим встраивания (`inline`), накладки (`overlay`) или всплытия (`popup`)

`destination`

Адрес перехода к установке, обычно это текущая страница

`node`

Идентификатор ID элемента, который будет содержать предложение к установке

После установки GCF заголовок `User-Agent` браузера будет расширен строкой `chromeframe`. Для выполнения URL-запросов GCF разумно использует имеющийся в Internet Explorer сетевой стек. Это гарантирует, что у запросов при использовании GCF имеются одни и те же cookie-файлы, история и SSL-состояние, что, как правило, приводит к сохранению существующих сессий пользователя.

Дополнительную информацию можно получить в руководстве по началу работы с этим расширением (http://www.chromium.org/developers/how-tos/chrome-frame-getting-started).

Создание макета

Давайте воспользуемся всем изученным материалом и применим знания для создания макета, навеянного проектом Holla.

Сначала создадим макет основной страницы. На этой странице мы собираемся разместить заголовок и два столбца — боковую панель фиксированной ширины `sidebar` и контейнер с главным содержимым `main`:

```
<body>
  <header id="title">
    <h1>Holla</h1>
  </header>
  <div id="content">
    <div class="sidebar"></div>
    <div class="main"></div>
  </div>
</body>
```

Затем давайте добавим основные стили для возврата в исходное состояние и для тела (body):

```
body, html {
  margin: 0;
  padding: 0;
}

body {
  font-family: Helvetica, Arial, "MS Trebuchet", sans-serif;
  font-size: 16px;
```

```
  color: #363636;
  background: #D2D2D2;
  line-height: 1.2em;
}
```

А теперь добавим h-теги:

```
h1, h2 {
  font-weight: bold;
  text-shadow: 0 1px 1px #ffffff;
}

h1 {
  font-size: 21pt;
  color: #404040;
}

h2 {
  font-size: 24pt;
  color: #404040;
  margin: 1em 0 0.7em 0;
}

h3 {
  font-size: 15px;
  color: #404040;
  text-shadow: 0 1px 1px #ffffff;
}
```

Теперь определим заголовок для нашего макета. Для этого воспользуемся фоновыми градиентами CSS3, но с возможностью возврата по умолчанию к обычному шестнадцатеричному коду цвета, если эти свойства не поддерживаются:

```
#title {
  border-bottom: 1px solid #535353;
  overflow: hidden;
  height: 50px;
  line-height: 50px;

  background: #575859;
  background: -webkit-gradient(linear, left top, left bottom,
  from(#575859), to(#272425));
  background: -webkit-linear-gradient(top, #575859, #272425);
  background: -moz-linear-gradient(top, #575859, #272425);
  background: linear-gradient(top, #575859, #272425);
}

#title h1 {
  color: #ffffff;
  text-shadow: 0 1px 1px #000000;
  margin: 0 10px;
}
```

Теперь, если посмотреть на окно браузера, показанное на рис. В.1, там будет темный заголовок с именем нашего приложения.

Рис. В.1. Пока наш CSS применяется только для отображения заголовка с фоновым градиентом

Давайте создадим div-контейнер #content, в котором будет находиться основная часть нашего приложения. Нам нужно, чтобы он был растянут на всю страницу в обоих направлениях *x* и *y*, поэтому мы сделаем его позицию абсолютной. Его непосредственные дочерние элементы выравниваются по горизонтали, поэтому их отображение будет настроено на тип гибких прямоугольных блоков (flexible box):

```
#content {
  overflow: hidden;

  /*
    Содержимое div-контейнера будет занимать всю страницу,
    но оставит достаточно места для заголовка.
  */
  position: absolute;
  left: 0;
  right: 0;
  top: 50px;
  bottom: 0;

  /* Горизонтальное выравнивание дочерних элементов */
  display: -webkit-box;
  -webkit-box-orient: horizontal;
  -webkit-box-align: stretch;
  -webkit-box-pack: left;

  display: -moz-box;
```

```
  -moz-box-orient: horizontal;
  -moz-box-align: stretch;
  -moz-box-pack: left;
}
```

Теперь создадим самый левый столбец под названием `.sidebar`. Он получает фиксированную ширину, поэтому свойство `box-flex` устанавливается в `0`:

```
#content .sidebar {
  background: #EDEDED;
  width: 200px;

/*
Элемент получает фиксированную ширину,
мы не собираемся его расширять
*/
  -webkit-box-flex: 0;
  -moz-box-flex: 0;
  box-flex: 0;
}
```

Давайте создадим внутри `.sidebar` список элементов меню. Каждое меню отделено от остальных меню его заголовком, заключенным в тег `h3`. Как видите, нами использовано довольно много свойств CSS3, которые из-за префиксов производителей страдают большой повторяемостью. Мы можем очистить код, воспользовавшись Less-миксинами:

```
#content .sidebar ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

#content .sidebar ul li {
  display: block;
  padding: 10px 10px 7px 20px;
  border-bottom: 1px solid #cdcdcc;
  cursor: pointer;
  -moz-box-shadow: 0 1px 1px #fcfcfc;
  -webkit-box-shadow: 0 1px 1px #fcfcfc;
  box-shadow: 0 1px 1px #fcfcfc;
}

#content .sidebar ul li.active {
  color: #ffffff;
  text-shadow: 0 1px 1px #46677f;
  -webkit-box-shadow: none;
  -moz-box-shadow: none;
  background: #7bb5db;
  background: -webkit-gradient(linear, left top, left bottom,
  from(#7bb5db), to(#4775b8));
```

продолжение ⇗

```
  background: -webkit-linear-gradient(top, #7bb5db, #4775b8);
  background: -moz-linear-gradient(top, #7bb5db, #4775b8);
  background: linear-gradient(top, #7bb5db, #4775b8);
}
```

Добавим к HTML-макету примеры меню:

```
<div class="sidebar">
  <h3>Channels</h3>
  <ul>
    <li class="active">Developers</li>
    <li>Sales</li>
    <li>Marketing</li>
    <li>Ops</li>
  </ul>
</div>
```

Теперь осталось определить CSS для div-контейнера .main, который растягивается по всей странице:

```
#content .main {
  -moz-box-shadow: inset 0 1px 3px #7f7f7f;
  -webkit-box-shadow: inset 0 1px 3px #7f7f7f;
  box-shadow: inset 0 1px 3px #7f7f7f;
  /* Нужно, чтобы .main распространялся как можно дальше */
  -webkit-box-flex: 1;
  -moz-box-flex: 1;
  box-flex: 1;
}
```

Давайте еще раз взглянем на окно браузера. Как показано на рис. В.2, теперь у нас есть основной макет приложения, который мы можем расширить.

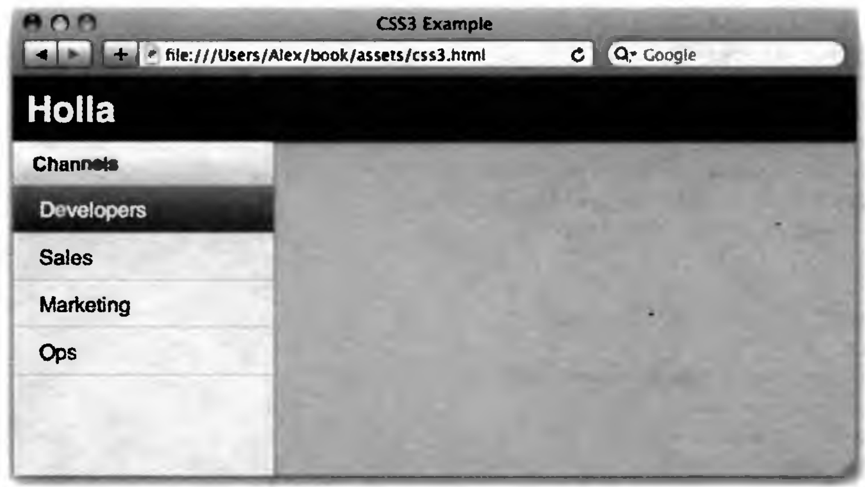

Рис. В.2. Основной макет приложения

Как упоминалось ранее, синтаксис CSS3 слишком многословный и повторяющийся из-за используемых нами префиксов производителей. Мы можем очистить код с помощью Less-миксинов. Например:

```
#content .sidebar h3 {
  .vbg-gradient(#FFF, #DEDFE0);
  .box-shadow(0, -5px, 10px, #E4E4E4);
}
```

Дополнительную информацию можно получить в Приложении Б и путем изучения ряда хороших примеров в таблицах стилей Holla.

Алекс Маккоу

Веб-приложения на JavaScript

Перевел с английского Н. Вильчинский

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Кривцов*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*О. Некруткина*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Ганчурина*
Верстка	*Л. Волошина*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 20.04.12. Формат 70x100/16. Усл. п. л. 23,220. Тираж 2000. Заказ № 339.
Отпечатано по технологии CtP в ГППО «Псковская областная типография».
180004, Псков, ул. Ротная, 34.