БИБЛИОТЕКА ПРОГРАММИСТА

Б. Хоган К. Уоррен М. Уэбер К. Джонсон А. Годин

Книга веб-программиста

Секреты профессиональной разработки веб-сайтов

- Как внедрить на сайт анимацию?
- Как планировать процесс веб-разработки?
- Как корректно использовать «резиновую» верстку?
- Как оптимизировать код?

Pragmatic Bookshelf

ПИТЕР

Web Development Recipes

Brian P. Hogan
Chris Warren
Mike Weber
Chris Johnson
Aaron Godin

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

БИБЛИОТЕКА ПРОГРАММИСТА

Б. Хоган, К. Уоррен, М. Уэбер, К. Джонсон, А. Годин

Книга веб-программиста

Секреты профессиональной разработки веб-сайтов

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.988.02-018
УДК 004.738.5
К53

Б. Хоган, К. Уоррен, М. Уэбер, К. Джонсон, А. Годин
К53 Книга веб-программиста: секреты профессиональной разработки веб-сайтов. — СПб.: Питер, 2013. — 288 с.: ил.

ISBN 978-5-459-01510-2

Эта книга предлагает широкий спектр передовых методов веб-разработки: от проектирования пользовательского интерфейса до тестирования проекта и оптимизации веб-хостинга.

Как внедрить на сайт анимацию, которая работает на мобильных устройствах без установки специальных плагинов? Как использовать «резиновую» верстку, которая корректно отображается не только на настольных ПК с различными разрешениями экрана, но и на мобильных устройствах? Как использовать фреймворки JavaScript — Backbone и Knockout — для разработки пользовательских интерфейсов? Как современные инструменты веб-разработчика, такие как CoffeeScript и Sass, помогут в оптимизации кода? Как провести кроссбраузерное тестирование кода? Как планировать процесс разработки сайта с помощью инструмента Git?

Ответы на эти и многие другие вопросы вы найдете в этой книге. Неважно, являетесь вы начинающим веб-программистом или уже имеете некоторый опыт разработки веб-приложений, это издание поможет вам освоить множество новых методов, приемов и подходов.

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1934356838 англ.
ISBN 978-5-459-01510-2

Краткое оглавление

Благодарности **12**

Введение **15**

Глава 1. Рецепты-конфетки **19**

Глава 2. Пользовательский интерфейс **47**

Глава 3. Данные **119**

Глава 4. Мобильные устройства **155**

Глава 5. Рабочий процесс **181**

Глава 6. Тестирование **219**

Глава 7. Хостинг и внедрение **255**

Приложение **285**

Библиография **287**

Оглавление

Благодарности 12

Введение 15

Для кого эта книга? 15
Что в этой книге? 15
Что вам нужно 16
HTML5 и jQuery 17
Оболочка 17
Ruby 17
QEDServer 17
Виртуальная машина 18
Сетевые ресурсы 18
От издательства 18

Глава 1. Рецепты-конфетки 19

Рецепт 1. Оформление кнопок и ссылок 20
Задача 20
Ингредиенты 20
Решение 20
Дополнительные возможности 23
Рецепт 2. Оформление цитат с помощью CSS 24
Задача 24
Ингредиенты 24
Решение 24
Дополнительные возможности 29
Рецепт 3. Анимация с помощью трансформаций CSS3 30
Задача 30
Ингредиенты 30
Решение 30
Дополнительные возможности 33
Рецепт 4. Интерактивные слайд-шоу с помощью jQuery 34
Ингредиенты 34
Решение 34
Добавление кнопок воспроизведения и паузы 36
Дополнительные возможности 37

Рецепт 5. Создание и оформление встроенных окон подсказки 39
Задача 39
Ингредиенты 39
Решение 39
Дополнительные возможности 46

Глава 2. Пользовательский интерфейс 47

Рецепт 6. Шаблон для HTML-сообщения 48
Задача 48
Ингредиенты 48
Решение 48
Рецепт 7. Навигация с помощью вкладок 58
Задача 58
Ингредиенты 58
Решение 58
Дополнительные возможности 63
Рецепт 8. Удобное раскрытие и сворачивание 64
Задача 64
Решение 64
Дополнительные возможности 68
Рецепт 9. Общение с веб-страницей с помощью горячих клавиш 70
Задача 70
Ингредиенты 70
Решение 70
Дополнительные возможности 76
Рецепт 10. HTML с помощью Mustache 77
Задача 77
Ингредиенты 77
Решение 77
Дополнительные возможности 81
Рецепт 11. Отображение данных с автоматической дозагрузкой 82
Задача 82
Ингредиенты 82
Решение 82
Дополнительные возможности 86
Рецепт 12. Ajax с контролем состояний 88
Задача 88
Ингредиенты 88
Решение 88
Дополнительные возможности 91
Рецепт 13. Модные клиентские интерфейсы с помощью Knockout.js 93
Задача 93
Ингредиенты 93

Решение ... 93
Дополнительные возможности ... 101
Рецепт 14. Как структурировать код с помощью Backbone.js ... 102
Задача ... 102
Ингредиенты ... 102
Решение ... 102
Дополнительные возможности ... 116

Глава 3. Данные ... 119

Рецепт 15. Встроенная карта Google ... 120
Задача ... 120
Ингредиенты ... 120
Решение ... 120
Дополнительные возможности ... 124
Рецепт 16. Диаграммы и графики с помощью Highcharts ... 125
Задача ... 125
Ингредиенты ... 125
Решение ... 125
Дополнительные возможности ... 131
Рецепт 17. Простая контактная форма ... 132
Задача ... 132
Ингредиенты ... 132
Решение ... 132
Дополнительные возможности ... 137
Рецепт 18. Межсайтовый доступ с помощью JSONP ... 139
Задача ... 139
Ингредиенты ... 139
Решение ... 139
Дополнительные возможности ... 141
Рецепт 19. Виджет, встраиваемый на внешние сайты ... 142
Задача ... 142
Ингредиенты ... 142
Решение ... 142
Дополнительные возможности ... 146
Рецепт 20. Уведомление о состоянии сайта с помощью JavaScript и CouchDB 147
Задача ... 147
Ингредиенты ... 147
Решение ... 147
Дополнительные возможности ... 153

Глава 4. Мобильные устройства ... 155

Рецепт 21. Специальное оформление для мобильных устройств ... 156
Задача ... 156
Ингредиенты ... 156

Решение ... 156
Дополнительные возможности ... 158
Рецепт 22. Меню, раскрывающееся при касании ... 160
Задача ... 160
Ингредиенты ... 160
Решение ... 160
Дополнительные возможности ... 162
Рецепт 23. Drag and Drop на мобильных устройствах ... 163
Ингредиенты ... 163
Решение ... 163
Дополнительные возможности ... 168
Рецепт 24. Интерфейсы с помощью jQuery Mobile ... 170
Задача ... 170
Ингредиенты ... 170
Решение ... 170
Дополнительные возможности ... 176
Рецепт 25. Спрайты с помощью CSS ... 178
Задача ... 178
Ингредиенты ... 178
Решение ... 178
Дополнительные возможности ... 179

Глава 5. Рабочий процесс ... 181

Рецепт 26. Удобный дизайн с помощью сетки ... 182
Задача ... 182
Ингредиенты ... 182
Решение ... 182
Дополнительные возможности ... 187
Рецепт 27. Простой блог с помощью Jekyll ... 190
Задача ... 190
Ингредиенты ... 190
Решение ... 190
Дополнительные возможности ... 196
Рецепт 28. Модульные таблицы стилей с помощью Sass ... 197
Задача ... 197
Ингредиенты ... 197
Решение ... 197
Дополнительные возможности ... 203
Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript ... 204
Задача ... 204
Ингредиенты ... 204
Решение ... 204
Дополнительные возможности ... 209

Рецепт 30. Управление файлами с помощью Git ... 211
Задача ... 211
Ингредиенты ... 211
Решение ... 211
Дополнительные возможности ... 218

Глава 6. Тестирование ... 219

Рецепт 31. Отладка JavaScript-кода ... 220
Задача ... 220
Ингредиенты ... 220
Решение ... 220
Дополнительные возможности ... 224
Рецепт 32. Определение областей активности пользователей с помощью «тепловой карты» ... 225
Задача ... 225
Решение ... 225
Дополнительные возможности ... 227
Рецепт 33. Тестирование в браузере с помощью Selenium ... 228
Задача ... 228
Ингредиенты ... 228
Решение ... 228
Дополнительные возможности ... 231
Рецепт 34. Тестирование с помощью Selenium и Cucumber ... 232
Задача ... 232
Ингредиенты ... 232
Решение ... 232
Дополнительные возможности ... 242
Рецепт 35. Тестирование JavaScript-кода с помощью Jasmine ... 244
Задача ... 244
Ингредиенты ... 244
Решение ... 244
Дополнительные возможности ... 253

Глава 7. Хостинг и внедрение ... 255

Рецепт 36. Как разместить статический сайт на Dropbox ... 256
Задача ... 256
Ингредиенты ... 256
Решение ... 256
Дополнительные возможности ... 258
Рецепт 37. Как установить виртуальную машину ... 259
Задача ... 259
Ингредиенты ... 259
Решение ... 259
Дополнительные возможности ... 262

Рецепт 38. Изменение файлов конфигурации веб-сервера с помощью Vim ... 263
Задача ... 263
Ингредиенты ... 263
Решение ... 263
Дополнительные возможности ... 266
Рецепт 39. Как защитить Apache с помощью SSL и HTTPS ... 267
Задача ... 267
Ингредиенты ... 267
Решение ... 267
Дополнительные возможности ... 269
Рецепт 40. Как обезопасить контент ... 270
Задача ... 270
Ингредиенты ... 270
Решение ... 270
Дополнительные возможности ... 272
Смотрите также ... 272
Рецепт 41. Как переписать URL, сохранив ссылки ... 273
Задача ... 273
Ингредиенты ... 273
Решение ... 273
Дополнительные возможности ... 276
Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake ... 277
Задача ... 277
Ингредиенты ... 277
Решение ... 277
Дополнительные возможности ... 283

Приложение ... 285

Установка Ruby ... 285
Windows ... 285
Mac OS X и Linux: установка через RVM ... 285

Библиография ... 287

Благодарности

Говорят, никто не пишет книгу в одиночку. Однако даже если авторов – пять, это всегда заканчивается тем, что вы привлекаете к работе еще множество других людей. Без помощи этих людей не было бы ни книги, ни опыта, полученного во время ее написания.

Сюзанна Пфальцер (Susannah Pfalzer), наш замечательный редактор-консультант, проделала исключительную работу, переспорив всех авторов и убедившись, что мы не отнеслись с пренебрежением к таким мелочам, как законченные фразы, введение и связность мысли. Мы хотели написать книгу, которая показала бы современному веб-разработчику обширную и разнородную коллекцию инструментария. Сюзанна всегда была рядом, чтобы убедиться, что мы объясняем не только «почему», но и «как», и от этого книга стала гораздо лучше.

Из-за стремления всех пятерых закончить работу побыстрее в текст вкрались ошибки и несвязанность. Однако благодаря нашим рецензентам Чарли Стену (Charley Stran), Джессике Яник (Jessica Janiuk), Кевину Гизи (Kevin Gisi), Мэту Марголису (Matt Margolis), Эрику Соренсону (Eric Sorenson), Скотту Андреасу (Scott Andreas), Джоелу Андричу (Joel Andritsch), Лайлу Джонсону (Lyle Johnson), Ким Шрайер (Kim Shrier), Стиву Хеферману (Steve Heffernan), Ноэлю Рэпину (Noel Rappin), Сэму Эллиоту (Sam Elliott), Дерику Бэйли (Derick Bailey) и Кэйтлин Джонсон (Kaitlin Johnson) конечный вариант книги получился гораздо лучше, чем тот, что был вначале.

Особая благодарность: Дэйву Гэмэчу (Dave Gamache) за совет по поводу Skeleton, Тревору Бернхэму (Trevor Burnham) за отзыв о CoffeeScript, Стиву Сандерсону (Steve Sanderson) за то, что наставил нас на путь истинный в вопросах Knockout.JS, и Бенуа Шесно (Benoit Chesneau) за быстрое решение некоторых проблем с установщиком CouchApp.

Кроме того, мы хотим поблагодарить наших деловых партнеров, в том числе Эриха Тески (Erich Tesky), Остин Отт (Austen Ott), Эмму Смит (Emma Smith), Джефа Холланда (Jeff Holland) и Ника ЛяМуро (Nick LaMuro), за помощь и отзывы, полученные от них в процессе работы.

Брайан Хоган

Это моя третья книга для «Pragmatic Bookshelf». И хотя здесь я написал всего одну пятую, эта книга потребовала наибольших усилий. Каждый из моих соавторов появился в нужное время и в нужном месте, и я с гордостью делю книгу с ними. Все они, Крис, Си-Джей, Майк и Аарон, привнесли в книгу увлекательные идеи и примеры, и я горжусь тем, что у нас получилось. Спасибо, ребята!

Но даже с той помощью, которая была у меня в этот раз, я не справился бы с работой без моей прекрасной жены Кэриссы. Спасибо тебе за то, что у меня было время все закончить. Спасибо, что позаботилась о второстепенных (а иногда и о важных) вещах, о которых я забыл.

Крис Уоррен

У меня не хватит слов, чтобы выразить благодарность моей потрясающей жене Кетлин за ее поддержку все то время, которое я занимался написанием и редактурой с раннего утра до поздней ночи.

Ты скрашивала тяжелые дни, делая их гораздо более сносными.

Спасибо моим соавторам за наш совместный опыт. Я уже давно знаю этих ребят, и было здорово впервые взяться писать книгу с друзьями. Особое спасибо Брайану, на протяжении многих лет играющему огромную роль в моем профессиональном развитии, за то, что вовлек меня в это начинание.

Наконец, спасибо моим родителям за содействие и поддержку как в писательском деле, так и в программировании все то время, пока я рос. Я не сказал вам, что написал это, и с волнением жду того момента, когда книга окажется у вас в руках.

Майк Уэбер

Я бы хотел поблагодарить Брайана Хогана за то, что он многие годы был моим наставником и сделал меня сначала веб-разработчиком, а теперь и публикующимся автором. Без него я не занимался бы ни тем, ни этим.

Я также хочу поблагодарить других моих соавторов, Криса, Си-Джея[1] и Аарона, за то, что они отправились со мной в это путешествие и помогали мне по дороге.

Также хочу поблагодарить мою семью, не дававшую мне отвлечься от поставленной задачи постоянным вопросом: «Как продвигается книга?»

И наконец, хочу поблагодарить мою жену Кэйли за то, что терпела работу допоздна вдали от нее, чтобы мы могли закончить книгу.

Крис Джонсон

Моей жене Лауре. Спасибо, что поддерживала меня на каждом этапе этого пути. Ты перестала проводить время со мной, чтобы я мог заниматься рукописью, уезжала в путешествия, чтобы я мог работать, и отказалась от множества летних развлечений, чтобы я мог писать.

Моим родителям. Спасибо, что научили меня работать ради того, чего мне хочется, и никогда не отступать. Папа, спасибо, что подождал со стартапом, чтобы я мог закончить книгу.

Спасибо Брайану, Крису, Майку и Аарону за совместную работу. Благодаря вашим отзывам и поддержке я стал писать лучше. Вы, ребята, заставляли меня доделывать те разделы, с которыми было нелегко, и я действительно ценю это.

Тем, кто с нами работал, спасибо, что выполняли функцию резонатора и дали технические отзывы на эту книгу.

Аарон Годин

Брайан, Крис, Майк и Си-Джей, каждый из них вдохновлял меня и был примером для подражания. Спасибо, что заставляли меня двигаться вперед, даже когда я не хотел. В особенности Брайану. Спасибо, что был лучшим наставником и другом, о котором я мог только мечтать.

[1] Крис Джонсон.

Спасибо Брайану Лонгу, который всегда слушал и проявлял интерес. Спасибо Тэйлор за сочувственное отношение и мотивацию, ты всегда была моей опорой, когда становилось тяжело.

Наконец, спасибо моим родителям Биллу и Синтии за безусловную поддержку, любовь и понимание. Вы оба научили меня держаться в жизни вещей, которые я люблю. Спасибо за то, что подготовили меня к жизни в этом мире, и за то, что делитесь той мудростью, которая нужна мне в самые трудные времена.

Введение

В современном мире требования к веб-разработчику не ограничиваются знанием синтаксиса HTML, CSS и JavaScript: сегодня высоко ценится умение писать хорошо тестируемый код, создавать интерактивные интерфейсы, использовать внешние сервисы и иногда даже заниматься настройкой сервера (или, по крайней мере, разбираться в устройстве серверной части приложения). В этой книге содержится более сорока практических указаний: от оригинальных дизайнерских CSS-приемов, которые не оставят равнодушным вашего клиента, до приемов настройки сервера, способных облегчить жизнь вам и вашим пользователям. Сочетание уже зарекомендовавших себя методов и самых современных решений поможет вам составить более полное представление об оптимальных средствах решения различных задач.

Для кого эта книга?

Если вы создаете веб-приложения, эта книга для вас. Если вы занимаетесь веб-дизайном или разработкой внешних интерфейсов и хотите расширить свои знания в сфере веб-разработки, вы сможете поиграть с новыми технологиями и библиотеками, которые помогут вам работать более эффективно, и параллельно с этим вникнуть в некоторые тонкости устройства серверной части.

Если вы занимаетесь разработкой серверной части приложений, но стремитесь быть в курсе современных методов создания внешних интерфейсов, вы также найдете здесь кое-что полезное, особенно в разделах о технологиях и тестировании.

И последнее: многие описанные здесь методы предполагают, что у вас есть опыт написания кода для клиентской части приложений с помощью JavaScript и jQuery. Если у вас такого опыта нет, все равно прочитайте их и внимательно разберите примеры кода. Трудности послужат стимулом для более детального изучения проблемы.

Что в этой книге?

Для начала, на пути к более совершенным методам веб-разработки, мы разберем ряд важных и интересных примеров. Каждый рецепт начинается с формулировки общей задачи, которая затем решается в контексте наиболее вероятного сценария. При этом задачи могут быть самыми разными: как протестировать сайт в разных браузерах; как быстро выполнить сборку и автоматическое внедрение простого статического сайта; как создать простую контактную форму, которая отправляет результаты по электронной почте; как настроить в Apache перенаправление URL и безопасное обслуживание страниц. Мы расскажем вам не только как можно решить проблему, но и почему ее лучше решать именно так. Так как эта книга представляет собой лишь собрание рецептов,

мы не можем позволить себе подробно обсуждать более сложные архитектуры. Тем не менее после каждого рецепта вы найдете раздел «Дополнительные возможности» с указанием направления дальнейших действий.

Рецепты сгруппированы по темам, однако совсем не обязательно читать их в том порядке, в котором они приведены в книге: выбирайте те, которые вам интересны. Внутри каждой главы рецепты приводятся начиная от простых и заканчивая более сложными.

В главе 1 «Рецепты-конфетки» говорится о том, как с помощью CSS и других инструментов сделать оформление страницы более живым и интересным.

В главе 2 «Пользовательский интерфейс» мы расскажем о том, как усовершенствовать пользовательский интерфейс с помощью таких средств, как JavaScript-фреймворки (например, Knockout и Backbone), а также как создавать более удобные шаблоны для отправки HTML-сообщений.

В главе 3 «Данные» вы научитесь работать с данными пользователя. Вы создадите простую форму обратной связи, а в качестве кульминации мы расскажем, как построить приложение, управляемое базой данных, с помощью CouchApp CouchDB.

В главе 4 «Мобильные устройства» мы продолжим разговор о пользовательских интерфейсах и научим вас работать с различными мобильными платформами. Часть разделов посвящена jQuery Mobile, часть — обработке событий мультитач. Кроме того, вы узнаете, в каком случае следует создать мобильную версию страницы и как это сделать.

Глава 5 «Рабочий процесс» посвящена методам усовершенствования процесса управления данными. Здесь мы расскажем, как использовать SASS при работе с большими таблицами стилей, а также познакомим вас с CoffeeScript — новым диалектом JavaScript, который позволяет создавать чистый код, совместимый с различными системами.

В главе 6 «Тестирование» вы научитесь создавать более «пуленепробиваемые» сайты с помощью автоматизированных тестов, а также тестировать ваш собственный JavaScript-код.

Наконец, мы обратимся к созданию конечного продукта в главе 7 «Хостинг и внедрение». Сначала мы создадим виртуальную машину, которая будет использоваться как среда тестирования, что позволит нам проверить работу отдельных компонентов до окончательного внедрения приложения. Далее мы расскажем, как сделать безопасный сайт, правильно оформить переадресацию и защитить ваш контент. Мы также покажем, как автоматически внедрить сайт, не забыв загрузить нужный файл.

Что вам нужно

В этой книге вы познакомитесь с множеством новых методов. Некоторые из них настолько новые, что еще вполне могут измениться, однако мы считаем эти методы достаточно устоявшимися, чтобы о них можно было кое-что рассказать. Тем не менее, принимая во внимание стремительное развитие веб-технологий, мы предоставляем вам возможность не отставать от нас, добавляя к исходному коду этой книги копии используемых библиотек.

Мы постарались свести требования к минимуму, однако кое-что все же следует узнать до того, как вы решите погрузиться в чтение.

HTML5 и jQuery

В наших примерах мы будем использовать HTML5-разметку. В частности, в нашей разметке вы не найдете самозакрывающихся тегов. В некоторых примерах вам встретятся новые теги, такие как `<header>` и `<section>`. Если вы не знакомы с HTML5, рекомендуем вам прочитать книгу Брайана Хогана «HTML5 и CSS3. Веб-разработка по стандартам нового поколения» (HTML5 and CSS3: Develop with Tomorrow's Standards Today) [Hog10].

Нам также понадобится библиотека jQuery: на ней основывается существенная часть библиотек, предлагаемых в этой книге. В большинстве случаев наши примеры вызывают jQuery 1.7 из сети доставки контента Google. Если примеры обращаются к другим версиям jQuery, в них содержатся соответствующие указания.

Оболочка

При решении задач старайтесь пользоваться командной строкой там, где это возможно. Это существенно увеличивает производительность, так как позволяет заменить несколько щелчков мыши одной командой. Кроме того, вы сможете создавать собственные скрипты на основе инструментов командной строки. Оболочка — это программа, которая интерпретирует такие команды. Если вы работаете в системе Windows, используйте cmd.exe. В системах OS X и Linux используйте Terminal.

Команды обычно выглядят следующим образом:

```
$ mkdir javascripts
```

`$` означает, что программа ожидает ввода команды, поэтому сам этот символ вводить не нужно. Так как команды и процессы являются платформенно независимыми, вам не составит труда выполнить указания, данные в книге, в любой системе — будь то Windows, OS X или Linux.

Ruby

Вам также потребуется установить язык программирования Ruby, так как некоторые рецепты используют его инструменты (в частности, Rake и Sass). Краткое руководство по установке Ruby можно найти в разделе «Приложение 1. Установка Ruby».

QEDServer

Некоторые рецепты используют уже существующее приложение для управления продуктом. Для этого можно установить QEDServer[1] — независимое приложение и базу данных, требующую минимума настроек. QEDServer работает в системах Windows, OS X и Linux. Единственное, что вам потребуется, — это Java Runtime Environment. Используя термин «среда разработки», мы имеем в виду именно это. В результате мы получаем надежную серверную часть, удобную для иллюстрации наших рецептов. С ее помощью можно легко обрабатывать Ajax-запросы на локальном компьютере.

Примеры, рассмотренные в этой книге, работают с версией QEDServer, которая размещена вместе с исходными кодами на сайте книги.

---

[1] Версия для этой книги доступна на http://webdevelopmentrecipes.com/.

Чтобы начать работу с QEDServer, запустите server.bat в Windows или ./server.sh в OS X или Linux. При этом будет создана папка public, которую можно использовать в качестве рабочей области. Созданный в этой папке файл index.html можно просмотреть в браузере, открыв http://localhost:8080/index.html.

Виртуальная машина

В некоторых главах книги используется веб-сервер на базе Linux с Apache и PHP. Созданию вашей собственной версии этого сервера посвящен Рецепт 37 «Как установить виртуальную машину». На сайте http://webdevelopmentrecipes.com/ можно получить уже готовую виртуальную машину. Для запуска виртуальной машины вам потребуется бесплатное приложение VirtualBox[1].

Сетевые ресурсы

На сайте[2] книги вы найдете ссылку на интерактивный форум, а также сможете сообщить об опечатках. Кроме того, там размещены исходные коды для всех проектов, разработанных в рамках этой книги. Электронные книги содержат прямые ссылки: чтобы загрузить фрагмент кода, щелкните мышью на ссылке, расположенной над листингом.

Мы надеемся, что вам понравится эта книга, и какие-то из ее идей пригодятся в вашем следующем проекте.

Брайан, **Крис**, **Си-Джей**, **Майк и Аарон**

От издательства

Ваши замечания, предложения и 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 по адресу http://www.piter.com.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

[1] http://www.virtualbox.org/

[2] http://pragprog.com/titles/wbdev/

1 Рецепты-конфетки

Серьезное приложение — это прекрасно. Но всего несколько мелких штрихов, касающихся пользовательского интерфейса, способны кардинально изменить ситуацию: работа с приложением станет приятной и удобной. А если это еще и легко реализовать...

В этом разделе мы расскажем, как оформлять кнопки и текст с помощью CSS, а также как добавлять анимацию, используя CSS и JavaScript.

Рецепт 1. Оформление кнопок и ссылок

Задача

Кнопки — важный интерактивный элемент сайта. Поэтому наш дизайн заметно выиграет, если мы оформим кнопки в духе всего сайта. Иногда нам приходится использовать кнопки и ссылки в одинаковом контексте: например, кнопку для отправки формы и ссылку для ее отмены. В таких ситуациях нам, конечно же, хочется, чтобы элементы хорошо сочетались друг с другом. Наконец, было бы здорово, если бы можно было менять оформление всех кнопок форм, не создавая каждый раз новое изображение.

Ингредиенты

- Браузер с поддержкой с CSS3 — например, Firefox 4, Safari 5, Google Chrome 5, Opera 10 или Internet Explorer 9.

Решение

CSS довольно часто используется для оформления ссылок и элементов форм. Однако с помощью класса и нескольких CSS-правил мы сможем создать такую таблицу стилей, которая сделает наши кнопки и ссылки одинаковыми, и таким образом избежать использования кнопок для оформления ссылок или ссылок для отправки форм.

Так как мы хотим, чтобы наши элементы выглядели как обычные кнопки, начнем с создания простой HTML-страницы с кнопкой и ссылкой.

cssbuttons/index.html

```
<p>
  <input type="button" value="A Button!" class="button" />
  <a href="http://pragprog.com" class="button">A Link!</a>
</p>
```

Обратите внимание на то, что оба элемента принадлежат классу `button`. С его помощью мы зададим оформление одновременно и ссылки, и элементов ввода, так что на странице их будет не отличить друг от друга.

Большинство атрибутов нашего класса `button` относятся к оформлению ссылок и элементов ввода. Однако чтобы добиться полного совпадения, нам понадобится несколько дополнительных атрибутов.

Сначала определим несколько базовых CSS-атрибутов, которые будут относиться к обоим элементам.

```
font-weight: bold;
background-color: #A69520;
text-transform: uppercase;
font-family: verdana;
border: 1px solid #282727;
```

Результат выглядит так:

Заметим, что нам уже удалось добиться некоторого соответствия. Однако до полного совпадения еще далеко. Разница в размере шрифта и отступах явно свидетельствует о том, что это разные элементы.

```
font-size: 1.2em;
line-height: 1.25em;
padding: 6px 20px;
```

Добавляя атрибуты `font-size`, `line-height` и `padding`, мы переопределяем соответствующие значения, заданные для элементов `link` и `input`. Остается разобраться с еще несколькими деталями, которые выдают происхождение наших элементов.

```
cursor: pointer;
color: #000;
text-decoration: none;
cursor: pointer;
color: #000;
text-decoration: none;
```

По умолчанию при наведении на кнопку курсор мыши не превращается из стрелки в указатель, как это происходит со ссылками. Поэтому нам придется выбрать какой-то из этих вариантов и применить его к обоим элементам. Кроме того, для текста ссылок используется цвет, установленный по умолчанию, и подчеркнутый шрифт.

Если приблизить наши кнопки в браузере, можно увидеть, что хотя их высота практически одинакова, ссылка все же чуть-чуть меньше. Такую разницу наверняка заметят пользователи мобильных устройств, поэтому мы добавим еще несколько строк, которые выравняют высоту кнопок.

```
input.button {
  line-height:1.22em;
}
```

Для элементов `input` с классом `button` мы зададим более высокое значение `line-height`. Это слегка подправит высоту элементов, и они будут выравнены. Чтобы понять, какую высоту следует задать, нужно просто приблизить элементы в браузере и подобрать такое значение `line-height`, при котором кнопки будут одинаковыми.

Таким образом мы окончательно избавимся от различий между ссылками и кнопками и сможем перейти к созданию их внешнего облика, а именно добавим закругленные уголки и немного тени. Это будет выглядеть так:

```
border-radius: 12px;
-webkit-border-radius: 12px;
-moz-border-radius: 12px;

box-shadow: 1px 3px 5px #555;
-moz-box-shadow: 1px 3px 5px #555;
-webkit-box-shadow: 1px 3px 5px #555;
border-radius: 12px;
```

продолжение ⇩

```
-webkit-border-radius: 12px;
-moz-border-radius: 12px;

box-shadow: 1px 3px 5px #555;
-moz-box-shadow: 1px 3px 5px #555;
-webkit-box-shadow: 1px 3px 5px #555;
```

Атрибуты радиуса и тени занимают по три строки. Это нужно для того, чтобы добиться желаемого эффекта в большем количестве браузеров. Первая строка каждой тройки (`border-radius` и `box-shadow`) предназначена для современных браузеров с поддержкой CSS3, а префиксы `-webkit-*` и `-moz-*` обеспечивают совместимость с более ранними версиями Safari и Firefox соответственно.

Последний штрих, относящийся к внешнему оформлению, — немного градиента для придания кнопкам объема. Это окажется полезным чуть позже, когда мы будем задавать внешний вид кнопок при нажатии.

```
background: -webkit-gradient(linear, 0 0, 0 100%, from(#FFF089), to(#A69520));
background: -moz-linear-gradient(#FFF089, #A69520);
background: -o-linear-gradient(#FFF089, #A69520);
background: linear-gradient(top center, #FFF089, #A69520);
background: -webkit-gradient(linear, 0 0, 0 100%, from(#FFF089), to(#A69520));
background: -moz-linear-gradient(#FFF089, #A69520);
background: -o-linear-gradient(#FFF089, #A69520);
background: linear-gradient(top center, #FFF089, #A69520);
```

И снова нам потребовалось несколько строк, чтобы добиться одного и того же эффекта в разных браузерах. В данном случае мы добавляем градиент к фону наших кнопок. Обратите внимание на префикс `-o-*`, необходимый для совместимости с Opera, который не использовался в предыдущем наборе CSS-атрибутов[1].

Наконец, мы хотим создать стиль, который будет обрабатывать события щелчка мыши и изменять внешний вид кнопки в случае, если она была нажата. Это настолько распространенный прием, что его отсутствие способно привести пользователя в замешательство. Существуют различные варианты отразить нажатие кнопки, и самый простой из них — перевернуть градиент.

```
.button:active, .button:focus {
  color: #000;
  background: -webkit-gradient(linear, 0 0, 100% 0, from(#A69520),
to(#FFF089));
  background: -moz-linear-gradient(#A69520, #FFF089);
  background: -o-linear-gradient(#A69520, #FFF089);
  background: linear-gradient(left center, #A69520, #FFF089);
}
```

[1] На http://www.westciv.com/tools/gradients/ вы найдете удобный инструмент, позволяющий правильно задать параметры градиента

Есть несколько способов перевернуть градиент. Но чтобы это работало одинаково во всех браузерах, проще всего поменять цвета местами. Если задать такой фон для `.button:active` и `.button:focus`, изменение будет происходить при щелчке мыши как на ссылке, так и на кнопке ввода.

Благодаря использованию CSS для ссылок и кнопок ввода мы можем добиться одинакового оформления элементов, предназначенных для разных целей: ссылки обеспечивают навигацию, а кнопки — отправку данных. При этом мы создаем надежный интерфейс, не прибегая к специальным уловкам: нам не нужно использовать JavaScript, чтобы заставить ссылку отправлять форму или заставить кнопку перенаправлять вас на другую страницу. В результате нам будет проще понять, как работает страница, а элементы сохранят свою функциональность даже в старых браузерах.

Дополнительные возможности

Если вы хотите, чтобы кнопка была недоступна для пользователя, вы можете либо совсем убрать ее, либо добавить к ней класс `disabled`. Как будет выглядеть этот класс? После того как вы создадите стиль для такой кнопки, что нужно сделать, чтобы по-настоящему ее отключить? К элементам ввода нужно добавить атрибут `disabled`, а вот для ссылок придется использовать JavaScript.

Смотрите также

- Рецепт 2. Оформление цитат с помощью CSS
- Рецепт 28. Модульные таблицы стилей с помощью Sass

Рецепт 2. Оформление цитат с помощью CSS

Задача

Мнения экспертов и отзывы пользователей, как правило, имеют большое значение, поэтому нам часто требуется акцентировать на них внимание пользователя с помощью визуального оформления. Это может быть дополнительный отступ, увеличение размера шрифта или, наконец, большие фигурные кавычки. На нашем сайте мы хотим реализовать простой и стандартный вариант, при котором все, что касается внешнего оформления цитат, хранилось бы отдельно от основного кода страницы.

Ингредиенты

- Браузер с поддержкой HTML5 и CSS3

Решение

Обычно CSS используется для того, чтобы разграничить контент и его внешнее оформление, и цитаты не должны являться исключением. Современные браузеры поддерживают ряд новых свойств, с помощью которых мы можем выделить цитаты, не добавляя много лишнего кода в разметку страницы.

Хотя в этом разделе мы будем обсуждать именно цитаты, приемы, описанные здесь, можно использовать и в других ситуациях. Например, если мы объединим таблицы стилей этого раздела с кодом из раздела «Рецепт 7. Навигация с помощью вкладок», мы сможем существенно улучшить дизайн страницы, подобрав разные цвета для разных групп данных. А с помощью идей из раздела «Рецепт 25. Спрайты с помощью CSS» можно добавлять к цитатам и другим элементам фоновые изображения.

Предположим, что нам нужно добавить к описаниям товаров нашего магазина краткие отзывы пользователей (длиной в несколько предложений). На каждой странице нужно разместить несколько цитат, так чтобы они не сливались с описанием товара. Давайте посмотрим, как в таком случае можно объединить возможности HTML и CSS.

Таблица стилей должна опираться на прочный фундамент, поэтому мы начнем с построения структуры HTML. Для цитаты и ее источника вполне разумно использовать теги `<blockquote>` и `<cite>` соответственно.

cssquotes/quote.html

```
<html>
  <head>
    <link rel="stylesheet" href="basic.css">
  </head>
  <body>
    <blockquote>
      <p>
        Determine that the thing can and shall be done,
        and then we shall find the way.
      </p>
    </blockquote>
    <cite>Abraham Lincoln</cite>
  </body>
</html>
```

Теперь, когда для цитат создана хорошая семантическая разметка, займемся их оформлением. Сначала пойдем по простому пути: заключим текст в рамку, увеличим размер шрифта и сделаем менее заметным имя автора, передвинув его вправо (рис. 1.1).

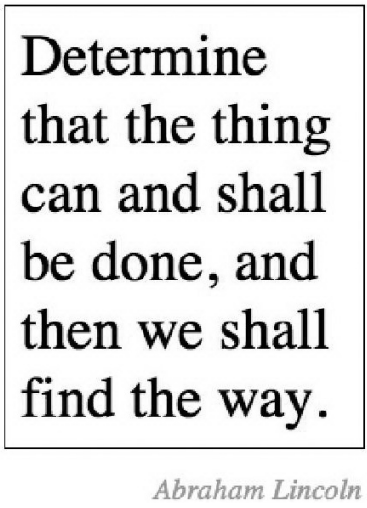

Рис. 1.1. Базовое оформление цитаты

cssquotes/basic.css

```
blockquote {
  width: 225px;
  padding: 5px;
  border: 1px solid black;
}

blockquote p {
  font-size: 2.4em;
  margin: 5px;
}

blockquote + cite {
  font-size: 1.2em;
  color: #AAA;
  text-align: right;
  display: block;
  width: 225px;
  padding: 0 50px;
}
```

В этой таблице стилей мы задаем одинаковую ширину двух основных элементов, `<blockquote>` и `<cite>`. Чтобы создать такое оформление только для элементов `<cite>`, следующих непосредственно после цитаты, мы используем смежный родственный

селектор. Затем мы меняем цвет имени автора, задаем все необходимые отступы и в результате получаем простую, но аккуратную цитату.

Теперь, когда базовый стиль цитаты готов, мы можем заняться более изящным оформлением. Давайте уберем границу и добавим перед цитатой большую ", которая привлечет внимание пользователя и одновременно расскажет, что это за объект (рис. 1.2).

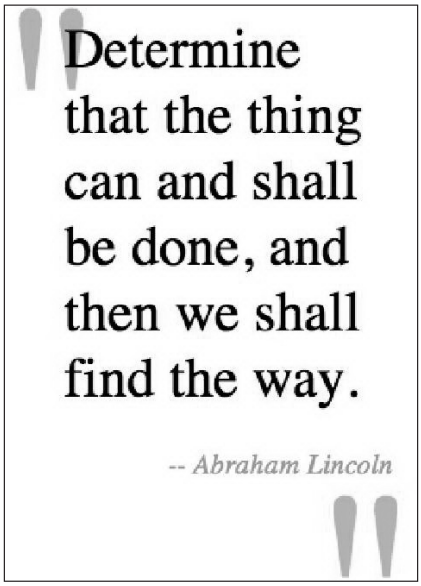

Рис. 1.2. Цитата после добавления кавычек с использованием CSS

cssquotes/quotation-marks.css

```
blockquote {
  width: 225px;
  padding: 5px;
}

blockquote p {
  font-size: 2.4em;
  margin: 5px;
  z-index: 10;
  position: relative;
}

blockquote + cite {
  font-size: 1.2em;
  color: #AAA;
  text-align: right;
  display: block;
  width: 225px;
  padding: 0 50px;
}
```

```
blockquote:before {
  content: open-quote;
  position: absolute;
  z-index: 1;
  top: -30px;
  left: 10px;
  font-size: 12em;
  color: #FAA;
  font-family: serif;
}

blockquote:after {
  content: close-quote;
  position: absolute;
  z-index: 1;
  bottom: 80px;
  left: 225px;
  font-size: 12em;
  color: #FAA;
  font-family: serif;
}
blockquote + cite:before {
  content: "-- ";
}
```

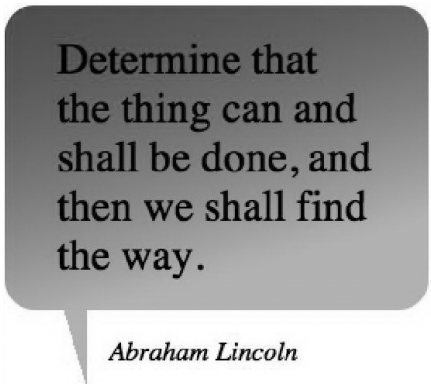

Рис. 1.3. Цитата в «облачке», оформленная с помощью CSS3

Эта таблица стилей помещает позади текста знаки кавычек, добавляет знак «–» перед именем автора и убирает черную рамку. При этом используются селекторы `:before` и `:after`, которые позволяют вставлять контент там, где встречаются определенные теги. Сам контент — будь то готовые значения, например `open-quote` и `close-quote`, или просто строка текста — можно задать с помощью одноименного атрибута (`content`).

Мы добавили еще несколько простых атрибутов, таких как цвет, семейство шрифта и размер шрифта, которые не нуждаются в пояснении. Внимание следует обратить на атрибуты `z-index`, а также на строку `position:relative;` внутри `blockquote p`. Атрибуты `position` вместе с `z-index` позволяют поместить кавычки позади текста, чтобы они не занимали лишнего места на странице; кроме того, текст поверх кавычек выглядит довольно симпатично. Чтобы закрывающая кавычка располагалась после текста независимо от длины цитаты, мы поместили `blockquote:after` внизу.

Для последней таблицы стилей нам придется собрать все свои знания, потому что мы хотим создать цитату в виде «облачка с текстом». С помощью новых атрибутов CSS3 мы сгладим острые углы и добавим градиент к фону и в результате получим цитату, изображенную на рис. 1.3.

cssquotes/speech-bubble.css

```
blockquote {
  width: 225px;
  padding: 15px 30px;
  margin: 0;
```

продолжение ⇨

```
  position: relative;
  background: #faa;
  background: -webkit-gradient(linear, 0 0, 20% 100%,
    from(#C40606), to(#faa));
  background: -moz-linear-gradient(#C40606, #faa);
  background: -o-linear-gradient(#C40606, #faa);
  background: linear-gradient(#C40606, #faa);
  -webkit-border-radius: 20px;
  -moz-border-radius: 20px;
  border-radius: 20px;
}

blockquote p {
  font-size: 1.8em;
  margin: 5px;
  z-index: 10;
  position: relative;
}

blockquote + cite {
  font-size: 1.1em;
  display: block;
  margin: 1em 0 0 4em;
}

blockquote:after {
  content: "";
  position: absolute;
  z-index: 1;
  bottom: -50px;
  left: 40px;
  border-width: 0 15px 50px 0px;
  border-style: solid;
  border-color: transparent #faa;
  display: block;
  width: 0;
}
```

Благодаря новым возможностям CSS3 мы можем создать «облачко с текстом», не используя никаких картинок. Сначала мы задаем цвет фона для `blockquote`, который будет отображаться во всех браузерах независимо от поддержки CSS3. Затем с помощью атрибута `linear-gradient` мы добавляем к фону градиент, а с помощью `border-radius` — закругленные уголки.

Поскольку атрибуты `linear-gradient` и `border-radius` имеют разный синтаксис в разных браузерах, для создания одинакового (или похожего) эффекта нам потребуется несколько строк кода. Префиксы `-moz` и `-webkit` относятся к Firefox и браузерам на движке WebKit (например, Safari или Google Chrome) соответственно. Далее, чтобы предусмотреть все остальные случаи, мы добавляем стандартный CSS3-атрибут.

В стилях `blockquote p` и `blockquote + cite` очень мало нововведений; для нескольких атрибутов мы задаем размер, но в целом все остается без изменений. Цвет и размер шрифта, а также отступы можно легко изменить, следуя общему стилю сайта.

Последний элемент `blockquote:after` рисует треугольный элемент под «облачком». В качестве значения атрибута `content` мы берем пустую строку, так как этот элемент не должен содержать текст — нас интересуют только его границы. Треугольник получается благодаря разной толщине границы вверху и внизу, а также слева и справа. CSS-атрибуты позволяют задавать отдельные значения для каждой стороны. В таком случае значения указываются по часовой стрелке начиная от верхнего: верхнее, правое, нижнее, левое. Мы используем этот прием как для толщины границ, так и для их цвета (`border-color`): верхнюю и нижнюю границы мы делаем прозрачными, правую и левую — цветными.

Дополнительные возможности

Какое еще оформление для цитаты вы можете предложить? В нашем последнем примере мы создали «облачко с текстом». Если поменять местами правую и левую границы в `blockquote:after`, мы получим фигуру, симметричную относительно вертикальной оси. А что нужно сделать, если мы хотим поместить имя автора — и, соответственно, треугольник — над текстом?

Фильтр градиента в браузерах Internet Explorer тоже позволяет создавать такое оформление, однако это несколько сложнее: градиент применяется не к элементу `background-image`, как в других браузерах, а к самому объекту. Сможете ли вы добавить поддержку для старых версий IE, воспользовавшись документацией[1] от Майкрософт?

Смотрите также

- Рецепт 1. Оформление кнопок и ссылок
- Рецепт 25. Спрайты с помощью CSS
- Рецепт 7. Навигация с помощью вкладок
- Рецепт 28. Модульные таблицы стилей с помощью Sass

[1] http://msdn.microsoft.com/en-us/library/ms532997.aspx

Рецепт 3. Анимация с помощью трансформаций CSS3

Задача

Для многих веб-разработчиков наиболее удобным инструментом при создании анимации является Flash, но что делать, если некоторые устройства, такие как iPad и iPhone, его не поддерживают? В тех случаях, когда анимация составляет важную часть контента, нам придется использовать другой способ, не требующий поддержки Flash.

Ингредиенты

- CSS3
- jQuery

Решение

С появлением CSS3-переходов и трансформаций у нас появилась возможность добавлять анимацию, не прибегая к различным плагинам, в частности к Flash. Такая анимация будут работать только в современных мобильных браузерах и последних версиях Firefox, Chrome, Safari и Opera; при этом, если анимация по какой-то причине не будет отображаться, пользователь все равно увидит изображение логотипа. Для остальных браузеров нам придется по-прежнему использовать Flash.

На сайте одного из наших клиентов анимация была выполнена с помощью Flash: при загрузке страницы по логотипу пробегал блик. Но после покупки нового iPad нашего клиента ждало страшное разочарование: на нем не только не работала анимация, но и вообще не отображался логотип. И хотя это не влияет на работу самого сайта, отсутствие логотипа снижает узнаваемость бренда. Поэтому мы постараемся сделать так, чтобы логотип отображался во всех браузерах, и добавим анимацию в тех браузерах, которые поддерживают трансформации CSS3.

Начнем с разметки заголовка, в котором будет содержаться наш логотип. К тегу `<img>` мы добавим специальный класс, чтобы потом создать для него таблицу стилей.

csssheen/index.html

```
<header>
  <div class="sheen"></div>
  <img src="logo.png" class="logo">
</header>
```

Чтобы добиться нужного эффекта, мы создадим полупрозрачный размытый наклоненный HTML-блок, который будет пробегать по экрану при загрузке DOM. Но для начала зададим базовое оформление нашего заголовка. В верхней части окна мы хотим видеть широкую голубую полосу. Для этого мы задаем ширину заголовка и помещаем логотип в его левом верхнем углу.

csssheen/style.css

```
body {
  background: #CCC;
  margin: 0;
}
header {
  background: #436999;
  margin: 0 auto;
  width: 800px;
  height: 150px;
  display: block;
  position: relative;
}
header img.logo {
  float: left;
  padding: 10px;
  height: 130px;
}
```

Теперь, когда у нас есть начальный макет, можно добавлять анимацию. Первым делом создадим размытый HTML-элемент; так как этот эффект является чисто декоративным и не имеет никакого отношения к контенту нашего сайта, мы хотим обойтись без лишней HTML-разметки. Для этого воспользуемся элементом `<div>` и его классом `sheen`, который мы добавили специально для этих целей.

csssheen/style.css

```
header .sheen {
  height: 200px;
  width: 15px;
  background: rgba(255, 255, 255, 0.5);
  float: left;
  -moz-transform:     rotate(20deg);
  -webkit-transform:  rotate(20deg);
  -o-transform:       rotate(20deg);
  position: absolute;
  left: -100px;
  top: -25px;
  -moz-box-shadow:    0 0 20px #FFF;
  -webkit-box-shadow: 0 0 20px #FFF;
  box-shadow:         0 0 20px #FFF;
}
```

Если мы посмотрим на нашу страницу (рис. 1.4), мы увидим тонкую белую прозрачную полоску, длина которой немного больше высоты заголовка. Отличное начало! Теперь сделаем блик размытым и передвинем так, чтобы он начинался слева от заголовка и был слегка наклонен.

Здесь нам придется немного схитрить. Дело в том, что браузеры еще не определились, как они будут отображать трансформации и переходы, поэтому нам понадобятся специфические префиксы браузеров. И хотя аргументы в этих объявлениях будут абсолютно

Рис. 1.4. Логотип с бликом до оформления

одинаковыми, префиксы будут отличаться — мы хотим быть уверены, что каждый браузер правильно отобразит этот элемент. Чтобы наша анимация отображалась и тогда, когда примут спецификацию CSS3, мы добавим объявление стиля без префикса. Например, в нашем коде нет объявления `-o-box-shadow`, потому что последние версии Opera уже не распознают этот стиль; версии Firefox 4+ больше не используют `-moz-box-shadow`, но распознают и заменяют его на `box-shadow`. Но для поддержки Firefox 3 мы решили оставить объявление `-moz-box-shadow`. Таким образом, нам пришлось пожертвовать чистотой кода в пользу функциональности.

Теперь мы почти готовы к тому, чтобы «оживить» наш блик. Следующим шагом будет добавление объявлений `transition`, необходимых для управления анимацией. И здесь нам снова потребуются префиксы браузеров.

```
header .sheen {
  -moz-transition:    all 2s ease-in-out;
  -webkit-transition: all 2s ease-in-out;
  -o-transition:      all 2s ease-in-out;
  transition:         all 2s ease-in-out;
}
```

Свойство `transition` принимает три аргумента. Первый сообщает браузеру, за изменением каких CSS-атрибутов он должен следить; в нашем случае это только атрибут `left`, так как мы всего лишь перемещаем блик по заголовку. Можно задать значение `all`, и тогда браузер будет отслеживать изменение всех атрибутов. Второй аргумент определяет, сколько времени (в секундах) должен занимать переход. Это значение может быть нецелым, например `0.5s`, или доходить до нескольких секунд, если изменения должны происходить медленно. Последний аргумент — название временной функции. Мы используем одну из встроенных функций, но вы можете задать свою собственную. Для этого хорошо подходит инструмент Ceaser[1].

Далее нам нужно определить, где блик должен остановиться. В нашем случае это будет правый край заголовка. *Можно* было бы воспользоваться событием `hover`:

```
header:hover .sheen{
  left: 900px;
}
```

Но в таком случае блик будет возвращаться в начальную точку каждый раз, когда пользователь будет уводить с заголовка курсор мыши. Если мы хотим, чтобы это происходило ровно один раз, нам придется изменить состояние страницы с помощью JavaScript. Добавим в нашу таблицу стилей специальный класс `loaded`, который будет перемещать блик в конечную позицию безвозвратно.

```
header.loaded .sheen {
  left: 900px;
}
```

[1] http://matthewlein.com/ceaser/

Если добавить этот класс к заголовку с помощью jQuery, произойдет запуск перехода.

```
$(function() { $('header').addClass('loaded') })
```

Если посмотреть на рис. 1.5, может показаться, что наша анимация — не более чем перемещение размытой полоски по экрану. Чтобы довести дело до конца, нам нужно подправить одну маленькую деталь. Для этого мы добавим стиль `overflow:hidden`, который поможет убрать часть блика, выходящую за рамки заголовка.

```
header {
  overflow: hidden;
}
```

Рис. 1.5. Логотип с бликом после оформления; блик выходит за рамки заголовка

С такой таблицей стилей мы можем запустить анимацию, просто изменив CSS-класс элемента. Теперь, чтобы создать на сайте плавно движущееся изображение, нам не нужно обращаться к специальным пакетам JavaScript или Flash.

Это решение имеет еще одно преимущество: оно не требует большой пропускной способности. И хотя для большинства пользователей это не является проблемой, нужно помнить, что кто-то может зайти на ваш сайт с мобильного устройства (например, iPad) через сотовую сеть. Наше решение требует загрузки меньшего числа файлов, и в результате страница отображается быстрее. При разработке сайта мы не должны забывать об оптимизации его работы.

В браузерах, не поддерживающих новые стили, на сайте будет отображаться простой логотип. Разделяя оформление и контент, мы не только обеспечиваем хорошую обратную совместимость, но и повышаем доступность контента программам экранного доступа (в теге `<img>` содержится название логотипа).

Чтобы анимация работала во всех браузерах, мы можем рассматривать такое решение как запасной вариант, сохранив Flash в качестве основного. Для этого нужно всего лишь поместить наш `<img>` в тег `<object>` вместе с Flash-файлом.

Дополнительные возможности

Многие переходы и трансформации, такие как масштабирование и наклон, остались за рамками нашего примера. Кроме того, существуют более тонкие настройки, позволяющие указывать время трансформаций или даже то, какие трансформации мы в данный момент хотим запустить. В некоторых браузерах можно создавать *свои собственные* переходы. Наконец-то в нашем распоряжении появились интересные встроенные возможности, позволяющие решить проблему анимации.

Смотрите также

- Рецепт 1. Оформление кнопок и ссылок
- Рецепт 2. Оформление цитат с помощью CSS
- Рецепт 28. Модульные таблицы стилей с помощью Sass

Рецепт 4. Интерактивные слайд-шоу с помощью jQuery

Еще несколько лет назад для создания анимированного слайд-шоу вы наверняка использовали бы Flash. Такое решение было бы очень просто реализовать, однако вам пришлось бы часто переделывать flash-ролик. Еще одна проблема состоит в том, что многие мобильные устройства не поддерживают Flash Player, поэтому их пользователи вообще не увидят слайд-шоу. Следовательно, нам необходимо альтернативное решение, простое в обслуживании и работающее на самых разных платформах.

Ингредиенты

- jQuery
- Плагин jQuery Cycle[1]

Решение

Используя jQuery и плагин jQuery Cycle, мы можем создать простую и элегантную презентацию изображений. Этот удобный инструмент находится в открытом доступе и требует только поддержки JavaScript.

Существует множество JavaScript-плагинов для просмотра изображений, но особенностью jQuery Cycle является простота использования. В нем есть встроенные эффекты переходов и элементы управления прокруткой изображений. Среди его несомненных плюсов простота обслуживания и популярность среди веб-разработчиков. Прекрасный выбор для нашего слайд-шоу.

Домашняя страница компании выглядит немного скучной: на ней не хватает движения. Поэтому начальник просит вас добавить слайд-шоу из лучших фотографий компании. Чтобы понять, как работает плагин jQuery Cycle, мы возьмем несколько фотографий и построим на их основе простой прототип.

Для начала построим простой шаблон домашней страницы index.html, используя стандартный код. На этой странице и будет помещаться галерея изображений.

image_cycling/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>AwesomeCo</title>
      </head>
      <body>
        <h1>AwesomeCo</h1>
      </body>
    </html>
```

Затем создадим папку images и поместим в нее несколько фотографий из тех, что нам дал руководитель компании. Эта папка хранится вместе с исходными кодами книги внутри папки image_cycling.

[1] https://github.com/malsup/cycle

Далее, в разделе `<head>` сразу после элемента `<title>` мы подключим jQuery и плагин jQuery Cycle. Нам также понадобится ссылка на файл rotate.js, в котором содержится JavaScript-код, необходимый для настройки слайдера изображений.

image_cycling/index.html

```
<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
</script>
<script type="text/javascript"
src="http://cloud.github.com/downloads/malsup/cycle/jquery.cycle.all.2.74.js">
</script>
<script type="text/javascript" src="rotate.js"></script>
```

Затем добавляем элемент `<div>` с идентификатором `slideshow`, внутри которого будут располагаться все наши изображения.

image_cycling/index.html

```
<div id="slideshow">
  <img src="images/house-light-slide.jpg" />
  <img src="images/lake-bench-slide.jpg" />
  <img src="images/old-building-slide.jpg" />
  <img src="images/oldbarn-slide.jpg" />
  <img src="images/streetsign-with-highlights-slide.jpg" />
  <img src="images/water-stairs-slide.jpg" />
</div>
```

На данном этапе наша страница выглядит примерно так, как показано на рис. 1.6. Именно такой она будет в случае, если у пользователя не будет работать JavaScript. Важно, что при этом на странице отображается весь имеющийся контент.

Так как мы еще не добавили код, запускающий плагин jQuery Cycle, на странице видны сразу все наши фотографии в исходном порядке. Чтобы начать прокрутку изображений, мы создадим файл rotate.js и добавим код, необходимый для настройки плагина.

image_cycling/rotate.js

```
$(function() {
  $('#slideshow').cycle({fx: 'fade'});
});
```

Плагин jQuery Cycle предоставляет нам множество дополнительных возможностей. В частности, можно добавлять эффекты переходов, такие как проявление и исчезновение (с масштабированием или без), вытеснение или дрожание. Полный список таких возможностей можно найти на сайте jQuery Cycle[1]. Мы возьмем простую и элегантную функцию `fade` и зададим ее внутри вызова `cycle()`.

```
fx: 'fade'
```

Теперь давайте еще раз посмотрим на нашу страницу. На этот раз мы видим только одно изображение, и через несколько секунд начинается прокрутка.

[1] http://jquery.malsup.com/cycle/options.html

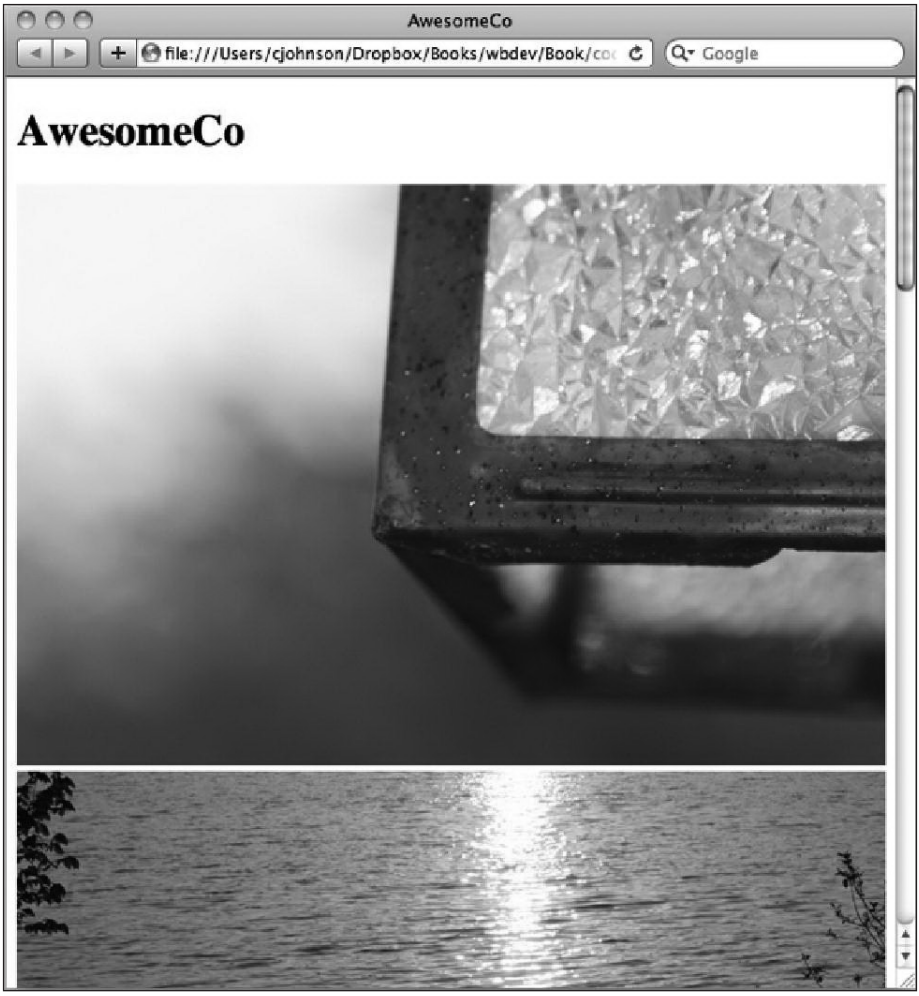

Рис. 1.6. Изображения перед добавлением прокрутки

Добавление кнопок воспроизведения и паузы

Итак, наше слайд-шоу прекрасно работает. Мы показываем его заказчику, и вот что он говорит: «Это, конечно, здорово, но хотелось бы еще иметь кнопку паузы, чтобы пользователь мог на время остановить прокрутку, если ему понравилась какая-то картинка». К счастью, jQuery Cycle предусматривает такую возможность.

Эти кнопки мы добавим с помощью JavaScript, так как они нужны только в процессе прокрутки. (Тем более что в браузерах без поддержки JavaScript эти кнопки будут абсолютно бесполезны.) Для этого мы создадим две функции: `setupButtons()` и `toggleControls()`. Первая добавит на нашу страницу кнопки и соответствующие им события `click()`. Эти события будут сообщать, когда прокрутку нужно приостановить или возобновить. Кроме того, событие `click()` должно вызывать `toggleControls()`. Эта функция будет следить за тем, чтобы отображалась только та кнопка, которую в данный момент можно нажать.

image_cycling/rotate.js

```
var setupButtons = function(){
  var slideShow = $('#slideshow');

  var pause = $('<span id="pause">Pause</span>');
  pause.click(function() {
    slideShow.cycle('pause');
    toggleControls();
  }).insertAfter(slideShow);

  var resume = $('<span id="resume">Resume</span>');
  resume.click(function() {
    slideShow.cycle('resume');
    toggleControls();
  }).insertAfter(slideShow);

  resume.toggle();
};

var toggleControls = function(){
    $('#pause').toggle();
    $('#resume').toggle();
};
```

Обратите внимание на то, что мы определяем селекторы jQuery как переменные. Это позволяет упростить работу с DOM. Кроме того, важно, что на выходе большинства функций, работающих от объектов jQuery, мы получаем также объекты jQuery. Благодаря этому мы можем присоединить `insertAfter()` к `click()`.

Для запуска функции `setupButtons()` мы должны вызвать ее после вызова `cycle()` в уже готовой функции jQuery.

image_cycling/rotate.js

```
$(function() {
  $('#slideshow').cycle({fx: 'fade'});
  setupButtons();
});
```

Посмотрим на нашу страницу еще раз (рис. 1.7): теперь там появилась кнопка паузы. Во время показа слайдов можно нажать кнопку Pause, после чего прокрутка остановится, а вместо кнопки Pause появится кнопка Resume. Если нажать на Resume, показ изображений возобновится.

Дополнительные возможности

Создать такое слайд-шоу было совсем не сложно, и теперь мы можем его усовершенствовать, воспользовавшись встроенными возможностями плагина[1].

[1] http://jquery.malsup.com/cycle/options.html

Рис. 1.7. Изображения с прокруткой и элементами управления

Визуально улучшить оформление можно с помощью различных эффектов, например `shuffle`, `toss` или `uncover`. Вы можете выбрать любой из них, просто поменяв значение `fx:` внутри вызова `cycle()`. Вместо изображений можно использовать и другие элементы, включая более сложные HTML-области. И это лишь некоторые возможности, изначально встроенные в плагин Cycle. Используйте их прямо сейчас!

Смотрите также

- Рецепт 3. Анимация с помощью трансформаций CSS3
- Рецепт 35. Тестирование JavaScript-кода с помощью Jasmine

Рецепт 5. Создание и оформление встроенных окон подсказки

Задача

Предположим, что на нашей странице есть много ссылок на небольшие блоки контента, расположенные в других частях сайта. Понятно, что в таком случае каждая ссылка открывается в новом окне, что существенно затрудняет чтение. Поэтому мы хотим найти элегантный способ отображать такой контент прямо на странице, стараясь не открывать новые окна без крайней необходимости.

Ингредиенты

- jQuery
- jQuery IU[1]
- jQuery Theme[2]

Решение

Итак, мы хотим, чтобы дополнительный контент «влился» в основной поток информации и чтобы наше решение работало в старых браузерах. Поэтому с помощью JavaScript мы уберем HTML-ссылки и добавим дополнительный контент прямо на страницу. Благодаря этому он будет отображаться и в браузерах, не поддерживающих JavaScript, тогда как пользователи более современных браузеров смогут насладиться изящным оформлением и удобством работы с контентом. При этом можно использовать любые эффекты jQuery, а также обычные и модальные диалоговые окна (рис. 1.8).

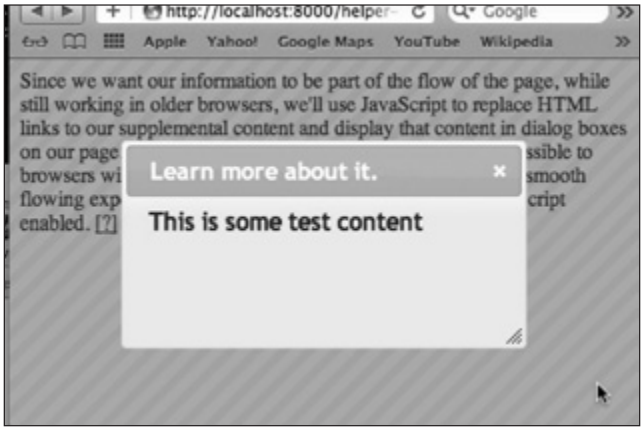

Рис. 1.8. Модальное диалоговое окно, всплывающее поверх контента

[1] http://jqueryui.com/
[2] http://jqueryui.com/themeroller/

Перед тем как перейти к JavaScript, мы создадим начальный вариант страницы, в котором мы подключим jQuery, jQuery UI и одну из тем jQuery, а также оформим первую ссылку.

inlinehelp/index.html

```
<html>
  <head>
    <link rel="stylesheet" href="jquery_theme.css"
    type="text/css" media="all" />

    <script type="text/javascript"
    src='http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js'>
    </script>
    <script type="text/javascript"
    src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.14/jquery-ui.min.
js'>
    </script>
    <script type="text/javascript" src='inlinehelp.js'></script>
  </head>
  <body>
    <p>
      This is some text.
      <a href="test-content.html"
        id="help_link_1"
        class="help_link"
        data-style="dialog"
        data-modal='true'>
        Learn more about it.
      </a>
    </p>
  </body>
</html>
```

Так как нам придется достаточно часто создавать на сайте такие объекты, код должен быть как можно более простым. Теперь, когда все готово, вот как будут выглядеть наши ссылки-подсказки:

```
<a href="a.html" id="help_a" class="help_link" data-style="dialog">
  More on A
</a>
<a href="b.html" id="help_b" class="help_link" data-style="dialog"
  data-modal="true">
  More on B
</a>
<a href="c.html" id="help_c" class="help_link" data-style="clip">
  More on C
</a>
<a href="d.html" class="help_link" data-style="fold">
  More on D
</a>
```

Обратите внимание на атрибуты `data-`, с помощью которых мы задаем оформление и модальность. Дело в том, что спецификация HTML5 допускает пользовательские атрибуты, благодаря которым мы можем изменять параметры элементов, оставаясь в рамках корректной разметки.

Следующий код задает настройки нашего будущего скрипта: в нем мы определяем несколько параметров, а затем вызываем функцию `displayHelpers()`. Теперь, чтобы отобразить содержимое ссылки прямо на странице, нам остается только добавить к этой ссылке класс; при желании можно задать стиль анимации и определить, должны ли диалоговые окна быть модальными.

inlinehelp/inlinehelp.js

```
$(function() {
  var options = {
    helperClass:"help_dialog"
  }

  displayHelpers(options);
});
```

Используя функцию jQuery `ready()`, мы можем быть уверены в том, что страница будет загружена до начала работы с DOM. Иными словами, весь исходный контент попадет на страницу, и при запуске кода ничего не потеряется. Чтобы окна выглядели изящнее, мы задаем несколько дополнительных параметров, которые не являются обязательными. Эти параметры передаются `displayHelpers()`. Это будет первым шагом к усовершенствованию визуального оформления.

inlinehelp/inlinehelp.js

```
function displayHelpers(options) {
  if (options != null) {
    setIconTo(options['icon']);
    setHelperClassTo(options['helper_class']);
  }

  else {
    setIconTo();
    setHelperClassTo();
  }

  $("a.help_link").each(function(index,element) {
    if ($(element).attr("id") == "") { $(element).attr("id", randomString()); }
    appendHelpTo(element);
  });
  $("a.help_link").click(function() { displayHelpFor(this); return false; });
}
```

Далее нам нужно создать специальный значок или текст, который подскажет пользователю, где искать дополнительные сведения. Так как ссылки должны сообщать пользо-

вателю что-то новое, имеет смысл убрать исходный текст. Кроме того, чтобы оформить диалоговые окна, мы создадим для них специальный класс.

inlinehelp/inlinehelp.js

```
function setIconTo(helpIcon) {
  isImage = /jpg|jpeg|png|gif$/
  if (helpIcon == undefined)
    { icon = "[?]"; }
  else if (isImage.test(helpIcon))
    { icon = "<img src='"+helpIcon+"'>"; }
  else
    { icon = helpIcon; }
}
```

Сначала функция `setIconTo()` проверяет, был ли задан параметр `help_icon`. Если нет, используется вариант по умолчанию: `[?]`. Если да, то она определяет, является ли переданный параметр путем к изображению, то есть заканчивается ли он одним из распространенных расширений. Если да, то этот путь помещается в элемент `<img>`. Если нет, функция отображает текст в том виде, в котором он был передан. Если окажется, что на вход был передан уже готовый элемент `<img>`, это нестрашно: он все равно отобразится правильно.

Далее мы хотим создать класс для оформления диалоговых окон. Здесь мы можем использовать как наши собственные таблицы стилей, так и темы jQuery UI.

inlinehelp/inlinehelp.js

```
function setHelperClassTo(className) {
  if (className == undefined)
    { helperClass = "help_dialog"; }
  else
    { helperClass = className; }
}
```

Функция `setHelperClassTo()` проверяет, был ли задан класс диалоговых окон в качестве параметра. Если да, то мы используем его, а если нет — класс по умолчанию `help_dialog`.

Важно также проверить, у всех ли ссылок есть идентификаторы: они будут нужны, чтобы соотнести ссылку с соответствующим ей элементом `<div>`. Поэтому если у ссылки нет идентификатора, необходимо его добавить.

```
$("a.help_link").each(function(index,element) {
  if ($(element).attr("id") == "") { $(element).attr("id", randomString()); }
  appendHelpTo(element);
});
```

Чтобы это проверить, мы просматриваем все ссылки с классом `help_link`. Если идентификатора не оказывается, мы используем вместо него случайно сгенерированную строку.

```
function randomString() {
  var chars = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXTZabcdefghiklmnopqrstuvwxyz";
```

```
  var stringLength = 8;
  var randomstring = '';
  for (var i=0; i<stringLength; i++) {
    var rnum = Math.floor(Math.random() * chars.length);
    randomstring += chars.substring(rnum,rnum+1);
  }
  return randomstring;
}
```

Функция `randomString()` генерирует случайную 8-символьную строку из букв и цифр. Число возможных строк настолько велико, что их заведомо хватит на все ссылки, у которых еще нет идентификаторов.

Теперь, когда у всех ссылок точно есть идентификаторы, мы вызываем функцию `appendHelpTo()`, которая добавляет значки к ссылкам и создает элементы для диалоговых окон, в которых будет содержаться дополнительный контент.

inlinehelp/inlinehelp.js

```
 1  function appendHelpTo(element) {
 -    if ($(element).attr("title") != undefined) {
 -      title = $(element).attr("title");
 -    } else {
 5      title = $(element).html();
 -    }
 -    var helperDiv = document.createElement('div');
 -    helperDiv.setAttribute("id",
 -      $(element).attr("id") + "_" + $(element).attr("data-style"));
10    helperDiv.setAttribute("class",
 -      $(element).attr("data-style") +" "+ helperClass);
 -    helperDiv.setAttribute("style", "display:none;");
 -    helperDiv.setAttribute("title", title);
 -
15    $(element).after(helperDiv);
 -    $(element).html(icon);
}
```

При вызове `appendHelpTo()` мы добавляем элемент `<div>`, в котором откроется наш дополнительный контент при нажатии на ссылке. Ему присваивается идентификатор, составленный из идентификатора ссылки и класса, ранее заданного в качестве параметра. Для элемента `<div>` мы определяем несколько классов: класс, заданный в качестве параметра, и класс, задающий стиль анимации. Наконец, мы задаем стиль отображения `display:none`, так как элемент не должен быть виден на странице до тех пор, пока пользователь не нажмет на ссылку.

Строка 16 функции `appendHelpTo()` заменяет ссылку значком. В итоге вместо ссылок на странице будут отображаться символы [?] или любые другие значки, заданные в качестве параметров.

inlinehelp/inlinehelp.js

```
$("a.help_link").click(function() { displayHelpFor(this); return false; });
```

Далее запускаем последнюю строку `displayHelpers()`, которая собирает все элементы с классом `help_link` и переопределяет их поведение при нажатии, вызывая функцию `displayHelpFor()` и возвращая значение `false`, чтобы предотвратить запуск обычного события щелчка.

inlinehelp/inlinehelp.js

```
function displayHelpFor(element) {
  url = $(element).attr("href");
  helpTextElement = "#"+$(element).attr("id") + "_" +
    $(element).attr("data-style");
  if ($(helpTextElement).html() == "") {
    $.get(url, { },
      function(data){
        $(helpTextElement).html(data);
        if ($(element).attr("data-style") == "dialog") {
          activateDialogFor(element, $(element).attr("data-modal"));
        }
        toggleDisplayOf(helpTextElement);
    });
  }
  else { toggleDisplayOf(helpTextElement); }
}
```

Сначала функция `displayHelpFor()` получает URL нажатой ссылки, чтобы мы знали, содержимое какой страницы нужно отобразить. Далее мы составляем идентификатор элемента `<div>` из частей так же, как мы делали это раньше, когда добавляли его к элементу. Именно сюда мы и добавим содержимое страницы с нашим URL. Но перед тем как начать загрузку контента, мы должны удостовериться, что элемент `<div>` пуст, то есть контент не был добавлен в него раньше. Если контент уже загружен, второй раз этого делать не нужно, поэтому остается только вызвать функцию `toggleDisplayOf()`. Так мы сможем эффективнее использовать пропускную способность, а пользователю не придется ждать загрузки страницы при повторном обращении к ней.

Если же контент еще не был загружен, мы загружаем его, обращаясь к URL через Ajax, и помещаем в `<div>`. После этого мы проверяем стиль оформления встраиваемого текста: если это диалоговое окно, то запускаем функцию `activateDialogFor()`, которая добавляет наше окно в DOM и задает значения модальности.

inlinehelp/inlinehelp.js

```
function activateDialogFor(element,modal) {
  var dialogOptions = { autoOpen: false };
  if (modal == "true") {
    dialogOptions = {
      modal: true,
      draggable: false,
      autoOpen: false
    };
  }
  $("#"+$(element).attr("id")+"_dialog").dialog(dialogOptions);
}
```

Этот код регистрирует диалоговое окно на странице, делая его доступным для пользователя. Чтобы активированное окно не открывалось автоматически, мы добавляем параметр `autoOpen:false`. Нам это нужно, потому что мы хотим открыть окно с помощью функции `toggleDisplayOf()` во избежание противоречий с другими диалоговыми окнами.

inlinehelp/inlinehelp.js

```
function toggleDisplayOf(element) {
  switch(displayMethodOf(element)) {
    case "dialog":
      if ($(element).dialog('isOpen')) {
        $(element).dialog('close');
      }
      else {
        $(element).dialog('open');
      }
      break;
    case "undefined":
      $(element).toggle("slide");
      break;
    default:
      $(element).toggle(displayMethod);
  }
}
function displayMethodOf(element) {
  helperClassRegex = new RegExp(" "+helperClass);
  if ($(element).hasClass("dialog"))
    { displayMethod = "dialog"; }
  else
    { displayMethod = $(element).attr("class").replace(helperClassRegex,""); }
  return displayMethod;
}
```

Благодаря функции `toggleDisplayOf()` на странице наконец-то появляется наш дополнительный контент. Сначала с помощью `displayMethodOf()` мы определяем, как его нужно отображать. Можно использовать любой метод из библиотек jQuery UI Effects или Dialog, так что сначала мы проверяем, есть ли у нашей ссылки стиль `dialog`. Если да, то мы возвращаем его; если нет, мы берем значение класса ссылки и удаляем из него вторую часть так, чтобы осталось только первое значение, задающее стиль анимации.

Функция `toggleDisplayOf()` определяет, когда нужно отобразить контент, а когда — скрыть. Если это диалоговое окно, мы с помощью `isOpen` проверяем, открыто ли оно, и в зависимости от этого закрываем или открываем его. Если нам не удается определить стиль анимации, мы используем значение по умолчанию `slide` и переключаем режим отображения элемента. Наконец, если `display_method` определен, мы переключаем режим отображения с его помощью.

Теперь, когда у нас есть такой код, мы можем легко создавать на нашей странице встраиваемые элементы, сохраняя хорошую совместимость со всеми браузерами. Кроме

того, наш код построен достаточно свободно, так что мы можем добавлять любые другие виды анимации, заботясь только о совместимости с новыми версиями jQuery.

Дополнительные возможности

Есть еще несколько параметров, которые было бы полезно объявить при инициализации кода, а именно класс для ссылок-подсказок и стиль анимации по умолчанию. Так как сейчас оба этих параметра трудно изменить, необходимо следить за тем, чтобы значение по умолчанию было задано даже там, где его изначально не было, — точно так же, как мы это делали, задавая класс для всех элементов `<div>` и значков.

Сейчас мы выкидываем текст ссылки и заменяем его значком. Но какие еще варианты возможны? Если оформить его как заголовок, пользователь сможет с помощью наведения курсора мыши узнать, что скрывается за этой ссылкой. А дизайн страницы будет ближе к своему первоначальному замыслу.

Смотрите также

- Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript
- Рецепт 35. Тестирование JavaScript-кода с помощью Jasmine

2

Пользовательский интерфейс

Удобный интерфейс необходим как сайтам со статическим контентом, так и интерактивным приложениям. Поэтому неважно, над чем вы работаете: вам в любом случае придется уделить внимание этому аспекту. В этой главе мы расскажем, в каком виде лучше преподносить информацию и как создавать эффективные и удобные в обслуживании пользовательские интерфейсы.

Рецепт 6. Шаблон для HTML-сообщения

Задача

HTML-почта как будто бы переносит нас на много лет назад — в те времена, когда еще не было CSS и в мире веб-дизайна безраздельно властвовали табличная верстка и теги `<font>`. Многое из того, что мы успели с тех пор узнать и полюбить, неприменимо к HTML-сообщениям, так как почтовые приложения просто не умеют их обрабатывать. Тестирование веб-страницы в нескольких браузерах кажется сущим пустяком по сравнению с той работой, которую необходимо выполнить при создании HTML-сообщения, так чтобы оно работало в Outlook, Hotmail, Gmail или Thunderbird, не говоря уже о почтовых приложениях мобильных устройств.

Но мы здесь не для того, чтобы жаловаться. Мы должны работать на результат, и нам предстоит многое сделать. Нам нужно не только создать HTML-сообщение, понятное для почтовых приложений, но и постараться, чтобы оно не было помечено как спам.

Ингредиенты

- Бесплатный пробный аккаунт на Litmus.com для тестирования сообщений

Решение

При создании HTML-сообщений мы вынуждены отказаться от многих современных достижений веб-разработки из-за ограничений почтовых клиентов. Пытаясь проследить за тем, чтобы наше сообщение не оказалось среди почтового мусора, нам придется избегать еще некоторых приемов. Нам также нужно, чтобы сообщение было легко тестировать на различных устройствах. Мы хотим создать нечто удобное, легко читаемое и эффективно работающее на разных платформах; лучший способ это сделать — вернуться к старой доброй HTML-разметке с табличной версткой.

Общие сведения об HTML-почте

По своей сути HTML-сообщения не так уж сложны. В конце концов, создание простой HTML-страницы не требует большого труда. Но, как и в случае с веб-страницами, мы не можем быть уверены в том, что пользователь увидит именно то, что мы задумали: разные почтовые клиенты немного по-разному отображают сообщения.

Начнем с того, что многие веб-клиенты, такие как Gmail, Hotmail и Yahoo!, часто удаляют из разметки объявления стилей или игнорируют их. Google Mail действительно удаляет стили, объявленные в теге `<style>`, чтобы они не накладывались на его собственные стили. На внешние таблицы стилей полагаться также не стоит, потому что многие почтовые клиенты не станут автоматически загружать удаленные файлы, не спросив об этом пользователя. Так что при верстке HTML-сообщений нам придется обходиться без CSS.

Поскольку Google Mail и Yahoo! удаляют или переименовывают тег `<body>`, нам лучше сразу использовать вместо него другой тег.

Некоторые клиенты плохо обрабатывают CSS-объявления в сокращенном виде, поэтому лучше писать их полностью. Например, такой код

```
#header{padding: 20px}
```

более старые почтовые клиенты просто проигнорируют, так что в наших интересах расписать его подробнее:

```
#header{
  padding-top: 20px;
  padding-right: 20px;
  padding-bottom: 20px;
  padding-left: 20px;
}
```

Клиенты для настольных ПК, такие как Outlook 2007 и Lotus Notes, не обрабатывают фоновые изображения, а Lotus Notes вдобавок не отображает формат PNG. На первый взгляд это может показаться небольшой проблемой, однако с этим почтовым клиентом работают миллионы корпоративных пользователей.

Здесь мы рассказали не обо всех возможных проблемах, уделив внимание только самым распространенным. На сайте Email Standard Project[1] есть исчерпывающие списки возможностей, которые поддерживаются в различных почтовых клиентах.

Вечеринка в стиле 1999

Если вдуматься, самые эффективные HTML-сообщения используют лишь базовые возможности HTML.

- Они строятся на основе простой HTML-разметки с минимальным CSS-оформлением.
- Для верстки используются HTML-таблицы (а не более современные методы).
- Не используется сложное шрифтовое оформление.
- Применяется очень простое CSS-оформление.

Одним словом, разработка сообщений должна выполняться так, как если бы последние десять лет веб-технологии вообще не развивались. Исходя из этого давайте создадим простой почтовый шаблон с табличной верстkой. Разработчик веб-приложений может взять этот шаблон и быстро разместить в нем весь необходимый контент. Но мы не будем спешить, ведь нам нужно, чтобы шаблон распознавался всеми распространенными почтовыми клиентами.

Наш счет будет состоять из тех элементов, которые есть во всех обычных счетах. Мы создадим «шапку» и «футер», а также разделы для нашего адреса и адреса плательщика. Мы добавим список объектов, которые приобрел клиент, и для каждого из них укажем цену, количество и промежуточную сумму. Нам также понадобится раздел для итоговой суммы и место для примечаний.

Так как некоторые почтовые клиенты выкидывают или переименовывают элемент `<body>`, нам придется использовать свой собственный элемент в качестве контейнера для сообщения. Наиболее надежный вариант — это создать для контейнера внешнюю таблицу и поместить все остальные таблицы (для «шапки», «футера» и основного контента) внутрь этого контейнера. Это будет выглядеть примерно так, как показано на рис. 2.1.

Итак, начнем с создания внешнего контейнера для нашего шаблона, используя версию HTML 4.0.

[1] http://www.email-standards.org/

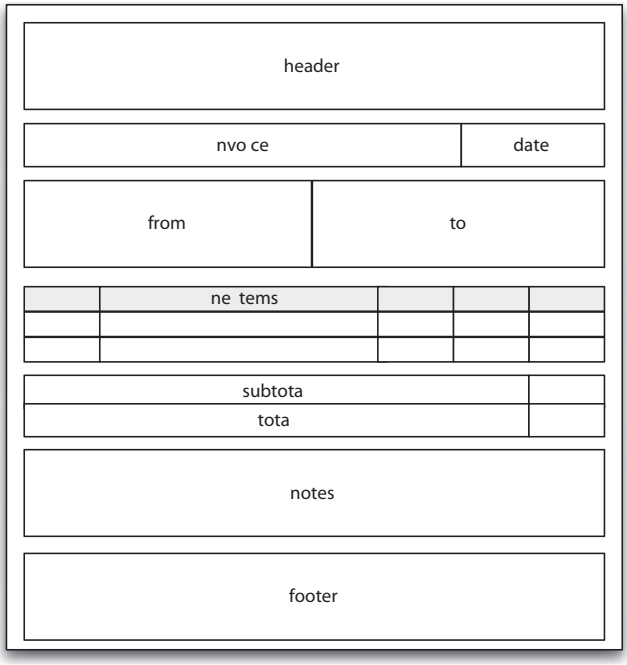

Рис. 2.1. Макет счета

htmlemail/template.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <title>Invoice</title>
</head>
<body>
  <center>
    <table id="inv_container"
      width="95%" border="0" cellpadding="0" cellspacing="0">
      <tr>
        <td align="center" valign="top">
        </td>
      </tr>
    </table>
  </center>
</body>
</html>
```

Чтобы выровнять счет по центру, нам придется прибегнуть к давно устаревшему тегу `<center>`, так как это единственный способ обеспечить (почти) правильное

отображение во всех почтовых клиентах. Не волнуйтесь, тег `<blink>` мы использовать не будем.

Далее мы создадим «шапку». Нам понадобится одна таблица для названия компании и еще одна — из двух столбцов — для номера счета и даты.

htmlemail/template.html

```
<table border="0" cellpadding="0" cellspacing="0" width="100%">
  <tr>
    <td align="center" bgcolor="#5d8eb6" valign="top">
    <h1><font color="white">AwesomeCo</font></h1>
    </td>
  </tr>
</table>

<table border="0" cellpadding="0" cellspacing="0" width="98%">
  <tr>
    <td align="left" width="70%"><h2>Invoice for Order #533102 </h2></td>
    <td align="right" width="30%"><h3>December 30th, 2011</h3></td>
  </tr>
</table>
```

Так как некоторые веб-клиенты удаляют таблицы стилей, цвет шрифта и фона нам придется задавать с помощью HTML-атрибутов. Заметим, что ширина первой таблицы равна 100%, а второй — 98%, и при этом таблицы выровнены по центру. Таким образом нам удалось добиться того, чтобы текст не располагался слишком близко к правому и левому краю внешней таблицы.

Далее добавим еще одну таблицу с адресами отправителя и получателя.

htmlemail/template.html

```
<table id="inv_addresses" border="0"
       cellpadding="2" cellspacing="0" width="98%">
  <tr>
    <td align="left" valign="top" width="50%">
      <h3>From</h3>
      AwesomeCo Inc. <br>
      123 Fake Street <br>
      Chicago, IL 55555
    </td>
    <td align="left" valign="top" width="50%">
      <h3>To</h3>
      GNB <br>
      456 Industry Way <br>
      New York, NY 55555
    </td>
  </tr>
</table>
```

Теперь добавим таблицу для самого счета.

htmlemail/template.html

```
<table border="0" cellpadding="2" cellspacing="0" width="98%">
  <caption>Order Summary</caption>
  <tr>
    <th bgcolor="#cccccc" align="left" valign="top">SKU</th>
    <th bgcolor="#cccccc" align="left" valign="top">Item</th>
    <th bgcolor="#cccccc" valign="top">Price</th>
    <th bgcolor="#cccccc" valign="top" width="10%">QTY</th>
    <th bgcolor="#cccccc" valign="top" width="10%">Total</th>
  </tr>
  <tr>
    <td valign="top">10042</td>
    <td valign="top">15-inch MacBook Pro</td>
    <td align="right" valign="top">$1799.00</td>
    <td align="center" valign="top">1</td>
    <td align="right" valign="top">$1799.00</td>
  </tr>
  <tr>
    <td valign="top">20005</td>
    <td valign="top">Mini-Display Port to VGA Adapter</td>
    <td align="right" valign="top">$19.99</td>
    <td align="center" valign="top">1</td>
    <td align="right" valign="top">$19.99</td>
  </tr>
</table>
```

Так как в нашей таблице содержатся фактические данные, нужно обязательно проследить за тем, чтобы все атрибуты, в частности названия столбцов и заголовок таблицы, были на месте.

Теперь добавим отдельную таблицу для итоговой суммы. Вы удивитесь, но это действительно необходимо, потому что некоторые почтовые клиенты до сих пор испытывают трудности при отображении таблиц с объединенными строками.

htmlemail/template.html

```
<hr>
<table border="0" cellpadding="2" cellspacing="0" width="98%">
  <tr>
    <td align="right" valign="top">Subtotal: </td>
    <td align="right" valign="top" width="10%">$1818.99</td>
  </tr>
  <tr>
    <td align="right" valign="top">Total Due: </td>
    <td align="right" valign="top"><b>$1818.99</b> </td>
  </tr>
</table>
```

Нам понадобится еще одна простая таблица для примечаний.

htmlemail/template.html

```
<table border="0" cellpadding="0" cellspacing="0" width="98%">
  <tr><td align="left">
    <h2>Notes</h2>
    <p>Thank you for your business!</p>
  </td></tr>
</table>
```

И наконец, добавим «футер». Как и в случае с «шапкой», мы будем использовать таблицу из одной ячейки с шириной 100 %.

```
<table id="inv_footer" border="0"
       cellpadding="0" cellspacing="0" width="100%">
  <tr>
    <td align="center" valign="top">
    <h4>Copyright &copy; 2012 AwesomeCo</h4>
    <h4>
       You are receiving this email because you purchased
       products from us.
    </h4>
    </td>
  </tr>
</table>
```

«Футер» — это самое подходящее место для того, чтобы объяснить пользователю, почему ему отправлено это письмо. В случае со счетом все более-менее понятно; если же это новостное сообщение, то логично поместить здесь различные ссылки, которые помогут пользователю настроить рассылку или отказаться от нее.

Итак, нам удалось создать простой HTML-счет, понятный большинству почтовых приложений. Но что делать с теми клиентами, которые вообще не распознают HTML-сообщения?

Как добавить поддержку там, где ее не может быть

Не все почтовые клиенты поддерживают HTML-почту, а те, которые поддерживают, делают это не идеально. Тем не менее пользователи таких приложений должны иметь возможность прочитать наше сообщение. Самый простой способ — добавить в начале сообщения ссылку на его копию, размещенную на наших серверах. Щелкнув на этой ссылке, пользователь сможет просмотреть сообщение в окне своего браузера.

В нашем случае мы можем поместить ссылку на копию счета внутри самого счета, а именно — в самой верхней части страницы, над таблицей с основным контентом, чтобы ее было хорошо видно. А если в почтовом приложении реализован предварительный просмотр писем, то пользователь сможет перейти по ссылке, даже не открывая сообщение.

htmlemail/template.html

```
<p>
  Unable to view this invoice?
  <a href="#">View it in your browser instead</a>.
</p>
```

Некоторые сторонние сервисы, такие как MailChimp и Campaign Monitor, активно используют этот прием, размещая HTML-сообщения на своих серверах в виде статических веб-страниц.

Возможен и другой вариант: отправить множественное (multipart) сообщение, то есть одновременно и простой текст, и его HTML-версию. В таком случае наше сообщение будет иметь два элемента `<body>` и несколько специальных заголовков, в которых мы сообщим почтовому клиенту, что сообщение содержит текстовую и HTML-версию. Для этого нам понадобится внятная текстовая версия счета. Или можно просто добавить ссылку на веб-страницу, на которой мы разместим счет.

Отправка множественных сообщений выходит за рамки этого раздела, однако большинство веб-фреймворков и почтовых клиентов позволяют это делать. О том, как эти сообщения работают, можно прочитать в Википедии в статье MIME[1].

ВОПРОС/ОТВЕТ. А РАЗВЕ НЕЛЬЗЯ СДЕЛАТЬ ВМЕСТО ТАБЛИЦ СЕМАНТИЧЕСКУЮ РАЗМЕТКУ?

Многие разработчики предпочитают следовать стандартам и не использовать таблицы вместо семантической разметки, полностью зависящей от CSS. То, что некоторые почтовые клиенты игнорируют таблицы стилей, их совершенно не волнует: контент все равно будет как-то отображаться, и его можно будет прочитать.

Однако если ваш заказчик требует, чтобы дизайн сообщения был одинаковым во всех почтовых клиентах, современные методы будут бессильны. Вот поэтому мы и используем табличную верстку.

Оформление с помощью CSS

Как мы уже говорили, из-за плохой поддержки CSS почтовыми клиентами мы вынуждены отказаться от таких средств, как плавающие элементы и абсолютное позиционирование. Поэтому нам приходится обращаться к таблицам. Справедливости ради стоит сказать, что эти клиенты удаляют из сообщений все CSS-стили вовсе не потому, что их разработчики питают лютую ненависть к стандартам. Просто если бы они этого не делали, стили сообщения могли бы вступить в противоречие со стилями веб-приложения.

Тем не менее есть две причины, почему мы все-таки оформим наше сообщение с помощью CSS. Во-первых, пользователи почтовых приложений с поддержкой CSS имеют право на более изощренный дизайн. Во-вторых, это оформление пригодится нам, когда мы захотим разместить счет на статической веб-странице (см. раздел «Как добавить поддержку там, где ее не может быть», с. 53).

Так как многие почтовые клиенты удаляют из документа раздел `<head>`, мы поместим информацию о стилях в тег `<style>` прямо над таблицей-контейнером.

Для начала давайте уберем лишние отступы в заголовках. Затем добавим к таблице фон и границу, а также увеличим расстояние между внутренними таблицами (кроме «футера»), чтобы отделить разные типы контента друг от друга.

htmlemail/template.html

```
<style>
  table#inv_addresses h3,
```

[1] http://ru.wikipedia.org/wiki/MIME

```
  table#inv_footer h4{
    margin: 0;
  }

  table{
    margin-bottom: 20px;
  }

  table#inv_footer{
    margin-bottom: 0;
  }

  body{
    background-color: #eeeeee;
  }

  table#inv_container{
    background-color: #ffffff;
    border: 1px solid #000000;
  }
</style>
```

Теперь наш счет выглядит так, как показано на рис. 2.2. Однако это еще не все: впереди нас ждет тестирование.

Рис. 2.2. Готовый счет

Тестирование сообщений

Перед тем как показать этот счет клиенту, нужно проверить, как он работает в разных почтовых приложениях. Можно отправить его паре десятков коллег; можно создать аккаунты Gmail, Yahoo Mail, Hotmail и пр. и посмотреть, как они отобразят сообщение. Однако ручное тестирование займет кучу времени.

Litmus[1] предлагает разработчикам набор инструментов для тестирования веб-страниц и почтовых сообщений. Он работает с множеством почтовых клиентов и браузеров, включая приложения для мобильных устройств. И хотя это платный сервис, в нем можно создать пробный аккаунт. С его помощью мы и проверим, как работает наше сообщение.

После регистрации на Litmus мы сможем создать тест и выбрать почтовые клиенты, в которых мы хотим проверить наше сообщение. После этого можно либо отправить счет по электронной почте на адреса, предложенные сервисом Litmus, либо передать HTML-файл с помощью специального веб-интерфейса. Вариант с передачей HTML-файла не поддерживает режим простого текста, поэтому в некоторых результатах мы увидим только исходный HTML-код, а не резервный текстовый вариант. Но для наших целей этого вполне достаточно.

Итак, Litmus берет наше сообщение, открывает его в почтовых приложениях из заданного списка и составляет подробный отчет (рис. 2.3).

Рис. 2.3. Результат тестирования

Похоже, что наше почтовое сообщение вполне адекватно отображается на основных платформах и остается читаемым на большинстве других платформ.

Сообщения и изображения

Рассматривая этот пример, мы ничего не говорили об изображениях по двум причинам. Во-первых, изображения пришлось бы хранить на сервере, а в письмо добавлять абсолютные ссылки. Во-вторых, большинство почтовых клиентов отключают изображения, так как многие компании с их помощью могут узнать, когда сообщение было открыто.

[1] http://litmus.com/

Если вы все-таки хотите, чтобы в вашем письме были картинки, необходимо придерживаться следующих правил.

- Постарайтесь, чтобы сервер, на котором хранятся ваши изображения, был всегда доступен, и не меняйте URL изображений. Никогда нельзя предугадать, в какое время получатель решит прочитать ваше письмо.
- Так как часто изображения по умолчанию отключены, не забудьте добавить к изображениям полезный описательный атрибут `alt`.
- Добавляйте изображения с помощью обычного тега `<img>`. Многие почтовые клиенты не поддерживают изображения, добавленные в качестве фона к ячейке таблицы; еще хуже дело обстоит с CSS-свойством `background`.
- Так как по умолчанию изображения часто бывают отключены, сообщение не должно состоять из одних только картинок. Как бы красиво ни выглядело ваше письмо, какой от него толк, если пользователь не сможет его прочитать?

Тем не менее картинки в сообщениях могут оказаться очень полезными, если все сделано аккуратно. Не бойтесь их использовать: просто внимательно относитесь к каждой детали.

Дополнительные возможности

Этот простой шаблон электронного сообщения позволяет отправлять счета, которые будут хорошо распознаваться почтовыми приложениями. Однако если бы мы имели дело с рекламным или новостным сообщением, такой дизайн был бы недостаточно привлекателен для пользователя. Нам бы понадобилось больше стилей, больше картинок, больше «особых ситуаций», связанных с работой различных почтовых приложений.

MailChimp[1] кое-что понимает в отправке электронных сообщений — в конце концов, это часть бизнеса. Если вас интересуют подробности, просмотрите шаблоны сообщений, который MailChimp хранит в открытом доступе[2]. Они уже прошли проверку в большинстве почтовых клиентов, а так как исходный код снабжен исчерпывающими комментариями, вы сможете легко понять, как разработчики всего этого добились.

[1] http://mailchimp.com/

[2] https://github.com/mailchimp/Email-Blueprints

Рецепт 7. Навигация с помощью вкладок

Задача

Иногда нам нужно разместить на сайте несколько однотипных фрагментов контента. Это может быть, к примеру, выражение на нескольких языках или пример кода на нескольких языках программирования. Можно, конечно, поместить их друг за другом, однако это займет много места, особенно если фрагменты большие. Должен быть другой способ компактно разместить контент, так чтобы пользователь мог переключаться между разными фрагментами и сравнивать их.

Ингредиенты

- jQuery

Решение

С помощью CSS и JavaScript мы можем превратить такой контент в аккуратный интерфейс с вкладками. Для каждого фрагмента, отмеченного специальным классом, будет создаваться отдельная вкладка; при этом в каждый момент времени открытой будет только одна. Кроме того, мы хотим создать гибкий дизайн, который позволил бы добавлять сколько угодно вкладок. В конечном итоге мы должны получить нечто похожее на рис. 2.4.

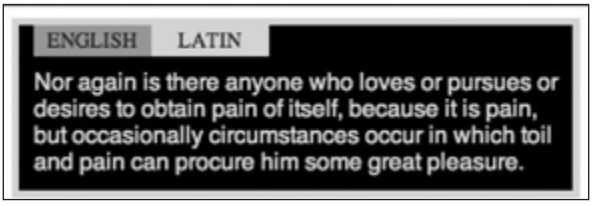

Рис. 2.4. Интерфейс с вкладками

Чтобы привлечь большую аудиторию, заказчик попросил нас разместить на странице описания товаров на разных языках. Мы создадим пробный вариант страницы, который подскажет нам, в каком направлении идти дальше.

Создание HTML

Начнем с создания HTML-разметки для тех элементов, которые мы хотим увидеть на странице. Для примера возьмем два текста: текст на английском языке и его перевод на латынь.

swapping/index.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Swapping Examples</title>
```

```
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
    </script>
    <link rel="stylesheet" href="swapping.css" type="text/css" media="all" />
    <script type="text/javascript" src="swapping.js"></script>
  </head>
  <body>
    <div class="examples">
      <div class="english example">
        Nor again is there anyone who loves or pursues or desires
        to obtain pain of itself, because it is pain, but occasionally
        circumstances occur in which toil and pain can procure him some
        great pleasure.
      </div>

      <div class="latin example">
        Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
        do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        Ut enim ad minim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat.
      </div>
    </div>
  </body>
</html>
```

Итак, мы задали базовую структуру наших элементов. В ней есть элемент `<div>` с классом `examples`, который содержит оба фрагмента с однотипным контентом. Каждый фрагмент мы дополнительно поместили в отдельный `<div>`.

Теперь, чтобы создать удобный интерфейс с вкладками, в котором пользователь смог бы переключаться между этими двумя примерами текстов, обратимся за помощью к JavaScript. Благодаря библиотеке jQuery мы получим доступ к вспомогательным методам и сможем пользоваться сокращенной записью.

Создание интерфейса с вкладками

Для начала нам нужно создать функцию, которая соберет вместе все кусочки нашей JavaScript-мозаики. Назовем ее styleExamples().

swapping/swapping.js

```
function styleExamples(){
  $("div.examples").each(function(){
    createTabs(this);
    activateTabs(this);
    displayTab($(this).children("ul.tabs").children().first());
  });
}
```

Мы находим все элементы `<div>` с классом `examples`: они будут использоваться как контейнеры для наших примеров текстов. Затем мы запускаем для каждого из них

функцию `createTabs()`, которая и создает интерфейс с вкладками. Эту функцию мы разберем прямо сейчас, а остальными займемся чуть позже.

swapping/swapping.js

```
function createTabs(container){
  $(container).prepend("<ul class='tabs'></ul>");
  $(container).children("div.example").each(function(){
    var exampleTitle = $(this).attr('class').replace('example','');
    $(container).children("ul.tabs").append(
      "<li class='tab "+exampleTitle+"'>"+exampleTitle+"</li>"
    );
  });
}
```

ВОПРОС/ОТВЕТ. А МОЖНО ДЛЯ ЭТОГО ИСПОЛЬЗОВАТЬ JQUERY UI TABS?

Да, безусловно. Но многое из того, что есть в UI Tabs (например, обработка событий), нам сейчас не нужно. К тому же, создавая вкладки своими руками и стараясь обходиться простыми конструкциями, мы лучше понимаем, как это все работает.

Для начала создадим неупорядоченный список, элементами которого станут наши вкладки. Для этого добавим соответствующий тег в начало содержимого того контейнера, в котором будут помещаться примеры текстов.

Затем пробежимся по всем примерам текстов внутри этого контейнера. У каждого такого примера есть два класса: название и класс `example`. Так как сейчас нам нужно только название, мы берем значение класса (с помощью `.attr('class')`) и заменяем в нем слово `example` на пустую строку. Так мы получаем название каждого примера, которое потом появится на соответствующей ему вкладке. Эти названия мы помещаем в теги `<li>`, которые, в свою очередь, добавляем в уже созданный список.

Если открыть эту страницу в браузере прямо сейчас, ничего не произойдет: пока не вызвана функция `styleExamples()`, JavaScript не будет работать. Давайте попробуем с этим разобраться.

Как переключаться между вкладками

Итак, наш контент стал чуть больше похож на интерфейс с вкладками, однако пользователь все еще не может переключаться между ними. Чтобы решить эту проблему, мы вызовем функцию `styleExamples()` при загрузке страницы. В результате элементы `<div>` с классом `example` превратятся в интерфейс с вкладками.

swapping/swapping.js

```
$(function(){
  styleExamples();
});
```

Если открыть нашу страницу в браузере теперь, мы увидим неупорядоченный список из двух элементов: english и latin. Но пока что это не очень большой шаг вперед. Давайте создадим функцию, которая будет переключаться между двумя примерами текстов. Сначала мы скроем оба примера, а затем отобразим тот, который хотим увидеть.

swapping/swapping.js

```
function displayTab(element){
  tabTitle = $(element)
    .attr('class')
    .replace('tab','')
    .replace('selected','').trim();

  container = $(element).parent().parent();
  container.children("div.example").hide();
  container.children("ul.tabs").children("li").removeClass("selected");

  container.children("div."+tabTitle).slideDown('fast');
  $(element).addClass("selected");
}
```

Мы обрезаем значение класса `tab selected english` так, чтобы осталось только `english`, и присваиваем его переменной `tabTitle`. Это понадобится, чтобы найти нужные нам элементы `<div>`. Сначала сделаем так, чтобы не было видно ни одной вкладки.

swapping/swapping.js

```
container = $(element).parent().parent();
container.children("div.example").hide();
container.children("ul.tabs").children("li").removeClass("selected");
```

Для этого мы берем контейнер и делаем скрытыми все его элементы `<div>` с классом `example`. Из каждого элемента `<li>` мы убираем класс `selected`. Хотя такого класса там может и не быть, гораздо легче пробежаться по всем элементам сразу, тем более что это сильно упрощает код. Теперь мы готовы к тому, чтобы на странице появился один из двух примеров.

swapping/swapping.js

```
container.children("div."+tabTitle).slideDown('fast');
$(element).addClass("selected");
```

Чтобы найти пример, который мы хотим отобразить, мы ищем `<div>` с таким же классом, как и класс переданного на входе элемента. Для него мы вызываем функцию `slideDown()`, которая раскрывает вкладку. Вместо нее можно использовать `.show()`, `.fadeIn()` или другие виды анимации из jQuery UI. Наконец, добавим класс `selected` к соответствующему элементу `<li>`, чтобы таблицам стилей было понятно, какая вкладка в данный момент открыта.

Теперь у нас есть функция `dispalyTab()`, но она нигде не используется. Понятно, что когда пользователь щелкает кнопкой мыши на названии одной из вкладок, мы должны открыть эту вкладку. Следовательно, вызов `displayTab()` должен происходить при щелчке мыши на элементе `<li>`.

Эта функция работает очень просто: она берет наш контейнер, находит в нем все элементы `<li>`, созданные функцией `createTabs()`, и заставляет их вызывать `displayTabs()` при щелчке мыши.

swapping/swapping.js

```
function activateTabs(element){
  $(element).children("ul.tabs").children("li").click(function(){
    displayTab(this);
  });
}
```

Оформление вкладок

Теперь вернемся к `styleExamples()` и посмотрим, как по мере загрузки страницы происходит вызов всех наших функций и как при этом меняется внешний вид страницы.

swapping/swapping.js

```
function styleExamples(){
  $("div.examples").each(function(){
    createTabs(this);
    activateTabs(this);
    displayTab($(this).children("ul.tabs").children().first());
  });
}
```

Последний вызов `displayTabs()` делает первую вкладку открытой по умолчанию. Это значит, что после загрузки страницы отображается именно она, тогда как все остальные оказываются скрытыми.

Таким образом мы завершаем работу над поведением элементов страницы и переходим к оформлению. Чтобы интерфейс выглядел так, как мы его себе представляли, добавим немного CSS.

swapping/swapping.js

```
li.tab {
  color: #333;
  cursor: pointer;
  float: left;
  list-style: none outside none;
  margin: 0;
  padding: 0;
  text-align: center;
  text-transform: uppercase;
  width: 80px;
  font-size: 120%;
  line-height: 1.5;
  background-color: #DDD;
}

li.tab.selected {
  background-color: #AAA;
}
```

```
ul.tabs {
  font-size: 12px;
  line-height: 1;
  list-style: none outside none;
  margin: 0;
  padding: 0;
  position: absolute;
  right: 20;
  top: 0;
}

div.example {
  font-family: "Helvetica", "san-serif";
  font-size: 16px;

}
div.examples {
  border: 5px solid #DDD;
  font-size: 14px;
  margin-bottom: 20px;
  padding: 10px;
  padding-top: 30px;
  position: relative;
  background-color: #000;
  color: #DDD;
}
```

Ну вот, все готово. Теперь у нас есть стандартный код, который пригодится нам при создании настоящего сайта. Там, где описания товаров даются на разных языках, пользователь сможет выбрать нужный язык с помощью удобного интерфейса.

На практике такое решение часто используется, когда необходимо сэкономить место на странице. Некоторые сайты помещают во вкладки достаточно разнородный контент: информацию о товаре, отзывы, ссылки. При этом если JavaScript отключен, информация все равно доступна пользователю, так как содержимое вкладок отображается линейно.

Дополнительные возможности

Что, если мы хотим, чтобы при загрузке страницы всегда открывалась какая-то определенная вкладка? Пусть, например, мы приводим примеры кода на Ruby, Python и Java, а пользователю нужен только Python. Было бы здорово, если бы ему не приходилось каждый раз открывать эту вкладку вручную. Попробуйте сами разобраться с этим вопросом.

Смотрите также

- Рецепт 8. Удобное раскрытие и сворачивание

Рецепт 8. Удобное раскрытие и сворачивание

Задача

Когда нам нужно перечислить большое количество элементов, сгруппированных по категориям, проще всего использовать вложенный неупорядоченный список. Однако такая верстка может не только затруднить навигацию по списку, но и вообще сделать его непонятным для пользователя. В таких ситуациях высоко ценится любое решение, способное облегчить жизнь пользователю. В качестве дополнения мы потребуем, чтобы список был доступен пользователям без JavaScript и программам экранного доступа.

Решение

Сравнительно простой способ построить вложенный список, не выделяя для каждой категории отдельную страницу, — это сделать список сворачивающимся. Это значит, что по усмотрению пользователя целые разделы списка могут отображаться или не отображаться.

Для примера мы создадим неупорядоченный список товаров, сгруппированных по категориям.

collapsiblelist/index.html

```
<h1>Categorized Products</h1>

<ul class='collapsible'>
  <li>
    Music Players

    <ul>
      <li>16 Gb MP3 player</li>
      <li>32 Gb MP3 player</li>
      <li>64 Gb MP3 player</li>
    </ul>
  </li>
  <li class='expanded'>
    Cameras & Camcorders
    <ul>
      <li>
        SLR
        <ul>
          <li>D2000</li>
          <li>D2100</li>
        </ul>
      </li>
      <li class='expanded'>
        Point and Shoot
        <ul>
          <li>G6</li>
```

```
          <li>G12</li>
          <li>CS240</li>
          <li>L120</li>
        </ul>
      </li>
      <li>
        Camcorders
        <ul>
          <li>HD Cam</li>
          <li>HDR-150</li>
          <li>Standard Def Cam</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

Возможно, мы захотим, чтобы при загрузке страницы некоторые узлы списка были свернуты, а некоторые — раскрыты. Но как это реализовать? Первое, что приходит в голову, — отметить свернутые узлы с помощью стиля `display:none`. Однако тогда содержимое этого узла станет недоступным для пользователей программ экранного доступа, так как последние игнорируют скрытый контент. Поэтому мы предлагаем другое решение: будем переключать режим отображения каждого узла в реальном времени с помощью JavaScript. А чтобы задать начальное состояние списка, мы воспользуемся CSS-классом `expanded`.

Если мы знаем, что пользователь хочет увидеть только список Point and Shoot Cameras, такой разметки будет недостаточно. На экране отобразится полный список, такой как на рис. 2.5. А если бы мы сделали список сворачивающимся, пользователь увидел бы только общие категории товаров (рис. 2.6) и смог бы сам посмотреть содержимое любых интересующих его категорий, оставаясь на одной и той же странице сайта.

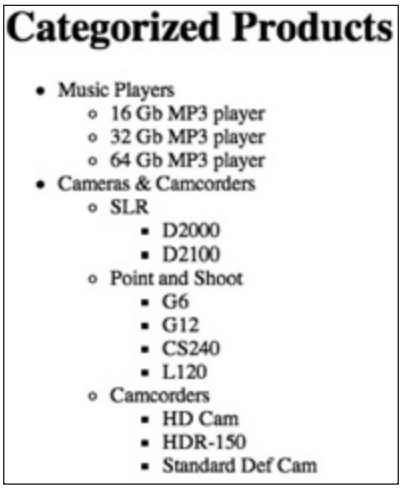

Рис. 2.5. Полный (несворачивающийся) список товаров, сгруппированных по категориям

Рис. 2.6. Свернутый список

Теперь давайте напишем JavaScript-код, который сделает список сворачивающимся, а также добавит вверху ссылки Expand all (Развернуть все) и Collapse all (Свернуть все). При этом важно, что для ссылок мы тоже используем JavaScript. Как и в случае со сворачиванием, мы не хотим добавлять в разметку лишние элементы, если мы не уверены, что они точно будут использоваться. К тому же так мы сможем применить эти свойства к любому другому списку, просто добавив класс `.collapsible` к элементу `<ul>`.

Для начала напишем функцию, которая будет переключаться между двумя состояниями узла: раскрытым и свернутым. Так как эта функция будет работать с объектом DOM, мы оформим ее как плагин jQuery. Это значит, что мы присвоим определение функции прототипу `jQuery.fn`. Потом мы сможем применить эту функцию ко всему элементу, для которого она будет вызвана. Определение функции нужно поместить внутрь самовызывающей функции, чтобы избежать конфликта с другими фреймворками, которые тоже могут использовать знак `$`. Наконец, чтобы наша функция поддерживала цепочки вызовов и вообще была полноправным жителем библиотеки jQuery, необходимо возвратить `this`. Следуя этим стандартным правилам написания плагинов jQuery, мы можем быть уверены, что *наша* функция будет работать точно так же, как и другие плагины jQuery.

collapsiblelist/javascript.js

```
(function($) {
  $.fn.toggleExpandCollapse = function(event) {
    event.stopPropagation();
    if (this.find('ul').length > 0) {
      event.preventDefault();
      this.toggleClass('collapsed').toggleClass('expanded').
        find('> ul').slideToggle('normal');
    }
    return this;
  }
})(jQuery);
```

Мы привяжем функцию `toggleExpandCollapse()` к событию щелчка мыши на всех элементах `<li>`, включая те, ниже которых ничего нет (они называются *листьями*). Дело в том, что у листьев очень ответственная роль: абсолютно ничего не делать. Здесь есть одна тонкость: необработанные события щелчка мыши «всплывают» по структуре DOM, поэтому если мы добавим обработчик событий только к элементам `<li>` с классами `.expanded` и `.collapsed`, событие щелчка мыши на листе «всплывет» и попадет к родительскому элементу `<li>`, который уже *является* сворачивающимся. Этот родительский элемент отреагирует на событие щелчка, то есть либо раскроется, либо, наоборот, свернется, чем нанесет неоправданный урон и без того хрупкой психике ошарашенного пользователя. Чтобы наш сайт не превратился в одну из машин Руба Голдберга, мы просто

запустим `event.stopPropagation()`. Добавляя обработчик событий к каждому элементу `<li>`, мы можем быть уверены в том, что ни одно событие щелчка мыши не «всплывет» наверх и с родительским элементом ничего не произойдет. Более подробно о том, как предотвратить «всплытие» событий, см. врезку «А почему бы просто не возвратить false?».

ВОПРОС/ОТВЕТ. А ПОЧЕМУ БЫ ПРОСТО НЕ ВОЗВРАТИТЬ FALSE?

В функциях jQuery код `return false` делает одновременно две вещи: запрещает событию «всплывать» по дереву DOM и предотвращает поведение элемента по умолчанию. Это хорошо работает для большинства событий, но иногда нам нужно разграничивать запрет на «всплытие» и отмену поведения по умолчанию. В какой-то ситуации нам может понадобиться всегда отменять действие по умолчанию, даже если функция работает неправильно. Поэтому иногда имеет смысл явно вызвать `event.stopPropagation()` или `event.preventDefault()`, не дожидаясь окончания работы функции, когда будет возвращено значение `false`[1].

Как мы уже говорили в начале главы, мы хотим поместить над списком специальные ссылки, с помощью которых можно будет свернуть или развернуть все узлы сразу. Эти ссылки можно создать с помощью jQuery, а затем добавить в начало нашего сворачивающегося списка. Так как для создания HTML с помощью jQuery может потребоваться много кода, мы не станем чрезмерно усложнять функции `prependToggleAllLinks()` и вынесем обработку событий в отдельные вспомогательные функции.

collapsiblelist/javascript.js

```
function prependToggleAllLinks() {
  var container = $('<div>').attr('class', 'expand_or_collapse_all');
  container.append(
      $('<a>').attr('href', '#').
      html('Expand all').click(handleExpandAll)
    ).
    append(' | ').
    append(
      $('<a>').attr('href', '#').
      html('Collapse all').click(handleCollapseAll)
    );
  $('ul.collapsible').prepend(container);
}

function handleExpandAll(event) {
  $('ul.collapsible li.collapsed').toggleExpandCollapse(event);
}

function handleCollapseAll(event) {
  $('ul.collapsible li.expanded').toggleExpandCollapse(event);
}
```

Чтобы создать объект DOM, просто заключим строку с нужным типом элемента (в нашем случае это тег `<a>`) в элемент jQuery. Затем с помощью API jQuery зададим его атрибуты и HTML-содержимое. Для простоты назовем наши ссылки Expand all

[1] http://api.jquery.com/category/events/event-object/

(Развернуть все) и Collapse all (Свернуть все) и поставим между ними вертикальную черту. В случае щелчка мышью на этих ссылках будут запускаться соответствующие вспомогательные функции.

Нам остается написать и инициализировать функцию, которая будет вызываться при загрузке страницы. Эта функция должна скрыть все узлы списка, не отмеченные как `.expanded`, и добавить класс `.collapsed` ко всем остальным элементам `<li>`.

collapsiblelist/javascript.js

```
function initializeCollapsibleList() {
  $('ul.collapsible li').click(function(event) {
    $(this).toggleExpandCollapse(event);
  });
  $('ul.collapsible li:not(.expanded) > ul').hide();
  $('ul.collapsible li ul').
    parent(':not(.expanded)').
    addClass('collapsed');
}
```

Событие щелчка мыши мы привязали ко всем элементам `<li>` из списка `.collapsible`. Кроме того, мы добавили классы `expanded` или `collapsed` ко всем элементам, кроме самих товаров. Эти классы понадобятся, когда наступит черед оформления списка.

Итак, когда DOM полностью готова, остается только собрать все кусочки в одно целое. Для этого мы инициализируем список и добавим ссылки Expand all | Collapse all (Развернуть все | Свернуть все).

collapsiblelist/javascript.js

```
$(document).ready(function() {
  initializeCollapsibleList();
  prependToggleAllLinks();
})
```

Так как это полноценный плагин jQuery, его можно применить и к любому другому неупорядоченному списку с помощью класса `.collapsible`. Такой код способен сделать любой длинный и громоздкий список понятным и удобным в использовании.

Дополнительные возможности

Если сначала создать прочный фундамент без использования JavaScript-кода, то потом на нем можно будет построить целое здание, добавив некоторые элементы поведения. При этом если мы напишем JavaScript-код и подключим его через CSS-классы, само здание и фундамент будут полностью отделены друг от друга. Благодаря этому наш сайт не будет слишком сильно зависеть от JavaScript: пользователи смогут работать с содержимым сайта, даже если JavaScript отключен. Мы называем этот подход *прогрессивным оформлением* и настоятельно рекомендуем ему следовать.

Так, создавая галерею изображений, попробуйте оформить каждый эскиз как ссылку на увеличенный вариант этого изображения, размещенный на отдельной странице. Затем с помощью JavaScript перехватите событие щелчка мыши на ссылке и откройте увеличенное изображение во всплывающем окне (эффект lightbox). Потом, как и в предыду-

щем примере, добавьте элементы управления, снова используя JavaScript. (Понятно, что они будут доступны только тем пользователям, у которых JavaScript включен.)

Если вы хотите создать форму, которая позволяет добавить сообщение и после этого обновляет страницу, начните с обычного запроса HTTP POST, а затем перехватите событие формы `submit`, используя JavaScript, и отправьте его с помощью Ajax. На первый взгляд это кажется слишком сложным, однако в конечном итоге вы экономите массу времени. Используйте все преимущества семантической разметки, а при подготовке данных формы не пренебрегайте разными хитрыми приемами: например, благодаря методу `serialize()` из jQuery вам не придется считывать каждое отдельное поле ввода и создавать свой собственный запрос POST.

Подобные методы пользуются хорошей поддержкой jQuery и ряда других библиотек, так как они реализуют простые решения, доступные для широкой аудитории.

Смотрите также

- Рецепт 9. Общение с веб-страницей с помощью горячих клавиш
- Рецепт 11. Отображение данных с автоматической дозагрузкой

Рецепт 9. Общение с веб-страницей с помощью горячих клавиш

Задача

Большинство пользователей привыкли «общаться» с сайтом с помощью мыши, хотя на самом деле это не самый удобный способ. В последнее время все более популярными — например, на Gmail и Tumblr — становятся сочетания клавиш, позволяющие быстро и удобно выполнять стандартные действия. Мы хотим, чтобы это стало возможным и на нашем сайте. При этом важно, чтобы не возникло никаких проблем со стандартным поведением приложения (как в случае с окном поиска).

Ингредиенты

- jQuery

Решение

С помощью JavaScript мы можем постоянно следить за тем, какие клавиши нажимает пользователь. Для этого нужно привязать специальную функцию к событию `keydown` нашего документа. После нажатия какой-либо клавиши мы проверяем, является ли она одной из наших горячих клавиш, и если да, то вызываем соответствующую ей функцию.

Представим, что на нашем сайте мы поместили огромное количество записей блога на совершенно разные темы. Проведя кое-какие тестирования, мы обнаружили следующее: когда пользователь решает, хочет ли он прочитать какую-то запись, он смотрит на ее заголовок и начало первого предложения. Если она ему не нравится, он прокручивает страницу вниз до следующей статьи. Так как среди статей встречаются и достаточно длинные, ясно, что пользователю это в конце концов надоедает. Поэтому мы хотим добавить несколько простых горячих клавиш для удобной навигации между записями и страницами, а также для работы с окном поиска. Наш интерфейс будет выглядеть примерно так, как показано на рис. 2.7.

С чего начать

Для начала попробуем сделать так, чтобы можно было легко переключаться между записями в пределах одной страницы. Мы создадим на нашей странице несколько элементов с классом `entry` и добавим горячие клавиши `j` и `k` для перехода к предыдущей и следующей записи соответственно. Эти клавиши используются для аналогичных целей во многих приложениях, включая Vim, о котором мы расскажем в рецепте 38 («Изменение файлов конфигурации сервера с помощью Vim»); в данном случае разумно следовать общепринятым обозначениям. После этого мы добавим навигацию между страницами с помощью стрелок вправо и влево, а затем создадим горячие клавиши для окна поиска.

Чтобы нам было на чем проверить, как работает навигация, мы создадим прототип с поисковой строкой и несколькими результатами поиска.

Рис. 2.7. Первоначальный вариант страницы с окном поиска и несколькими записями блога

keyboardnavigation/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
    </script>
    <script type="text/javascript"
      src='keyboard_navigation.js'></script>
  </head>
  <body>
    <p>Make this page longer so you can tell that we're scrolling!</p>
    <form>
      <input id="search" type="text"size="28" value="search">
    </form>
    <div id="entry_1" class="entry">
      <h2>This is the title</h2>
      <p>Lorem ipsum dolor sit amet...</p>
    </div>
    <div id="entry_2" class="entry">
      <h2>This is the title of the second one</h2>
      <p>In hac habitasse platea dictumst...</p>
    </div>
  </body>
</html>
```

Ради экономии места мы не стали приводить здесь длинный HTML-код. Чтобы прокрутка записей имела смысл, добавьте еще несколько разделов `<div id="entry_x" class="entry">`, так чтобы они не помещались на одном экране.

Захват нажатия клавиш

Чтобы настроить обработку событий при загрузке страницы, мы обратимся к jQuery. При нажатии одной из клавиш навигации нам нужно будет вызвать соответствующую функцию. Внутри метода `$(document).keydown()` мы с помощью оператора выбора определим, какая именно функция соответствует какой клавише. Для этого нам понадобятся специальные коды клавиш[1].

keyboardnavigation/keyboard_navigation.js

```
$(document).keydown(function(e) {
  if($(document.activeElement)[0] == $(document.body)[0]){
    switch(e.keyCode){
    // Навигация внутри страницы
    case 74: // j
      scrollToNext();
      break;
    case 75: // k
      scrollToPrevious();
      break;
    // Навигация между страницами
    case 39: // стрелка вправо
      loadNextPage();
      break;
    case 37: // стрелка влево
      loadPreviousPage();
      break;
    // Поиск
    case 191: // / (и ? при нажатой клавише Shift)
      if(e.shiftKey){
        $('#search').focus().val('');
        return false;
      }
      break;
    }
  }
});
```

Прежде чем проверить, какая из горячих клавиш была нажата, нужно убедиться, что она не использовалась для чего-то другого: например, для ввода текста в окне поиска или в обычном текстовом поле. Поэтому в первой строке нашей функции `if($( document.activeElement)[0] == $(document.body)[0])` мы проверяем, находится ли активный элемент в пределах тела страницы.

[1] Полный список кодов вы найдете на http://www.cambiaresearch.com/c4/702b8cd1-e5b0-42e6-83ac-25f0306e3e25/javascript-char-codes-key-codes.aspx

Навигация внутри страницы

Чтобы реализовать навигацию внутри страницы, нам понадобится список записей блога и номер текущей записи. Логично, что при первом переходе мы должны попадать на первую запись страницы.

keyboardnavigation/keyboard_navigation.js

```
$(function(){
  current_entry = -1;
});
```

При загрузке страницы мы задаем значение переменной `current_entry`, равное `-1`: это значит, что ни один переход еще не был выполнен. Чтобы понять, почему мы берем именно `-1`, давайте подумаем, как мы собираемся искать тот элемент, к которому хотим перейти. Проще всего составить массив из всех объектов с классом `.entry` и обращаться к его элементам по индексу. А так как в JavaScript элементы массива нумеруются с нуля, первая запись будет иметь индекс 0.

В разделе «Захват нажатия клавиш» мы определили, какие функции соответствуют каким клавишам. Так как мы хотим, чтобы при нажатии клавиши `j` пользователь переходил к следующей записи, мы должны вызвать функцию `scrollToNext()`.

keyboardnavigation/keyboard_navigation.js

```
function scrollToNext(){
  if($('.entry').size() > current_entry+1){
    current_entry++;
    scrollToEntry(current_entry);
  }
}
```

В начале функции `scrollToNext()` мы проверяем, существует ли вообще тот элемент, к которому мы хотим перейти. Для этого достаточно убедиться том, что при увеличении счетчика `current_entry` на единицу мы не получим число, превосходящее общее количество записей на странице. Если переход возможен, мы увеличиваем `current_entry` на единицу и вызываем `scrollToEntry()`.

keyboardnavigation/keyboard_navigation.js

```
function scrollToEntry(entry_index){
  $('html,body').animate(
    {scrollTop: $("#"+$('.entry')[entry_index].id).offset().top},'slow');
}
```

Чтобы перейти к нужной записи, функция `scrollToEntry()` использует библиотеки для создания анимации jQuery. Так как мы знаем номер этой записи (значение `current_entry`), мы можем определить ее идентификатор. Зная идентификатор, jQuery сможет отобразить следующую запись.

При нажатии клавиши `k` мы хотим выполнить аналогичную функцию под названием `scrollToPrevious()`, выполняющую переход к предыдущей записи.

keyboardnavigation/keyboard_navigation.js

```
function scrollToPrevious(){
  if(current_entry > 0){
    current_entry--;
    scrollToEntry(current_entry);
  }
}
```

Функция `scrollToPrevious()` проверяет, не пытаемся ли мы перейти к записи с номером меньше нуля, так как ноль соответствует первой записи. Если все в порядке, мы уменьшаем значение `current_entry` на единицу и снова вызываем `scrollToEntry()`.

Благодаря такому способу прокрутки записей пользователь может быстро просмотреть содержимое целой страницы. Но в один прекрасный момент он дойдет до ее конца и тогда захочет перейти на следующую страницу. Так что давайте решим и эту задачу.

Навигация между страницами

Можно по-разному реализовать навигацию между страницами. В нашем случае мы предполагаем, что номер страницы указан в строке запроса URL page=1; другие варианты — например, p=1, entries/2 и т. д. — потребуют лишь небольших корректировок.

Ради чистого и аккуратного кода давайте напишем отдельную функцию `getQueryString()`, которая будет доставать номер страницы из URL.

keyboardnavigation/keyboard_navigation.js

```
function getQueryString(name){
  var reg = new RegExp("(^|&)"+ name +"=([^&]*)(&|$)");
  var r = window.location.search.substr(1).match(reg);
  if (r!=null) return unescape(r[2]); return null;
}
```

Теперь создадим функцию `getCurrentPageNumber()`, которая с помощью `getQueryString()` будет проверять, есть ли в адресе номер страницы, заданный с помощью параметра page. Если да, то она переведет строковое значение номера в целочисленное и возвратит его на выходе. Если нет, это значит, что номер страницы еще не задан. Тогда мы считаем, что находимся на первой странице, и возвращаем `1`. Важно, чтобы функция возвращала не строку, а целочисленное значение, так как потом нам нужно будет произвести кое-какие вычисления.

keyboardnavigation/keyboard_navigation.js

```
function getCurrentPageNumber(){
  return (getQueryString('page') != null) ?
    parseInt(getQueryString('page')) : 1;
}
```

Функция, которая следит за нажатием горячих клавиш, реагирует на клавиши «стрелка влево» и «стрелка вправо». «Стрелка вправо» переводит нас на следующую страницу с помощью функции `loadNextPage()`, которая определяет номер текущей страницы и передает браузеру адрес следующей.

keyboardnavigation/keyboard_navigation.js

```
function loadNextPage(){
  page_number = getCurrentPageNumber()+1;
  url = window.location.href;
  if (url.indexOf('page=') != -1){
    window.location.href = replacePageNumber(page_number);
  } else if(url.indexOf('?') != -1){
    window.location.href += "&page="+page_number;
  } else {
    window.location.href += "?page="+page_number;
  }
}
```

Сначала мы определяем номер текущей страницы, а затем увеличиваем значение `page_number` на единицу (ведь мы хотим попасть на следующую страницу). Затем мы берем URL текущей страницы, чтобы немного его изменить и таким образом получить адрес следующей. Здесь мы сталкиваемся с определенными трудностями: дело в том, что URL может быть устроен по-разному. Сначала мы проверяем, есть ли в нем подстрока page=. Если да, как, например, в http://example.com?page=4, нам остается только заменить номер, используя регулярные выражения и функцию `replace()`. Замена номера нам понадобится и при переходе на предыдущую страницу, так что создадим для этого отдельную функцию `replacePageNumber()`. Если потом структура адреса изменится, нам придется менять код только в одном месте.

keyboardnavigation/keyboard_navigation.js

```
function replacePageNumber(page_number){
  return window.location.href.replace(/page=(\d)/,'page='+page_number);
}
```

Если в URL нет подстроки page=, нам придется добавлять этот параметр в строку запроса целиком. Далее мы проверяем, есть ли в строке запроса другие параметры. Если да, то они будут указаны после знака ?, так что будем искать ?. Если мы его найдем, как, например, в http://example.com?foo=bar, то мы добавим номер страницы в конец адреса. Если не найдем, то нам придется создавать строку запроса самим, что мы делаем в последнем блоке `else` конструкции `if...else`.

Нечто похожее мы делаем и для того, чтобы перейти к предыдущей странице. Однако здесь все немного проще: после того как мы определим номер текущей страницы и уменьшим его на единицу, нам останется лишь проверить, что получившееся число (`page_number`) не меньше 1, то есть больше 0. Если все хорошо, мы поставим это число в адрес после page=, и все будет готово.

keyboardnavigation/keyboard_navigation.js

```
function loadPreviousPage(){
  page_number = getCurrentPageNumber()-1;
  if(page_number > 0){
    window.location.href = replacePageNumber(page_number);
  }
}
```

Итак, на нашем сайте есть удобные средства навигации внутри страницы и между страницами. Давайте добавим еще одну возможность: быстрый доступ к окну поиска.

Переход к окну поиска

Логичнее всего выбрать в качестве горячей клавиши ?. Но так как фактически для этого требуется нажатие двух клавиш одновременно, нам придется немного усложнить процесс проверки: сначала мы будем искать код 191, соответствующий клавише /, а затем поверять значение свойства `shiftKey` нашего события, которое возвращает `true` при нажатой клавише Shift.

keyboardnavigation/keyboard_navigation.js

```
case 191: // / (и ? при нажатой клавише Shift)
  if(e.shiftKey){
    $('#search').focus().val('');
    return false;
  }
  break;
}
```

Если клавиша Shift нажата, мы находим окно поиска по DOM-идентификатору и вызываем функцию `focus()`, чтобы поместить курсор мыши в поле ввода. Затем мы удаляем все содержимое окна поиска с помощью `val('')`. И наконец, мы возвращаем `false`, чтобы символ ? не был добавлен в строку поиска.

Дополнительные возможности

Благодаря этим горячим клавишам пользователь может быстро перемещаться по сайту, не отрывая рук от клавиатуры.

Имея готовую схему, вам не составит труда добавить и другие горячие клавиши: например использовать пробел для эффекта lightbox, открывать во всплывающем окне консоль со списком задач или просматривать сообщение блога целиком.

Горячие клавиши можно использовать и во многих других примерах этой книги, в которых используется JavaScript, — например, для удобного просмотра изображений (Рецепт 4. Интерактивные слайд-шоу с помощью jQuery) или для просмотра и разворачивания узлов списка (Рецепт 8. Удобное раскрытие и сворачивание).

Смотрите также

- Рецепт 4. Интерактивные слайд-шоу с помощью jQuery
- Рецепт 8. Удобное раскрытие и сворачивание
- Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript
- Рецепт 38. Изменение файлов конфигурации сервера с помощью Vim

Рецепт 10. HTML с помощью Mustache

Задача

Чтобы создавать поистине потрясающие интерфейсы, нужно уметь работать с динамическим и асинхронным HTML. Благодаря Ajax- и JavaScript-библиотекам, таким как jQuery, мы можем обновлять пользовательский интерфейс без перезагрузки страницы, просто меняя HTML-код с помощью JavaScript. Для добавления на страницу новых элементов традиционно используется конкатенация строк кода, но это достаточно трудоемкое занятие, которое к тому же является потенциальным источником ошибок. Используя такое решение, мы сразу же попадаем в непроходимый лес одинарных и двойных кавычек, где на каждом шагу вызывается метод `append()`.

Ингредиенты

- jQuery
- Mustache.js

Решение

К счастью, существуют специальные инструменты, такие как Mustache, которые позволяют создавать реальный HTML-код, добавлять с его помощью настоящие данные и вставлять все это в документ. Mustache — это инструмент для создания шаблонов, реализованный на нескольких языках программирования. JavaScript-версия позволяет с помощью чистого HTML-кода создавать клиентское представление, отвлеченное от JavaScript-кода. В ней также возможны условные операторы и операторы цикла.

Mustache позволяет упростить процесс создания HTML-кода при добавлении нового контента. Работая с приложением для управления продуктом на основе JavaScript, мы познакомимся с синтаксисом Mustache.

Существующее на данный момент приложение позволяет добавлять в список товаров новые элементы. Поскольку обработка запросов выполняется с помощью JavaScript и Ajax, в примере используется наш стандартный сервер разработки. Когда пользователь хочет добавить новый товар и заполняет соответствующую форму, на сервер поступает запрос о сохранении этой информации, после чего товар появляется в списке. Но для этого нам необходима конкатенация строк, которая делает код громоздким и неудобным для чтения.

mustache/submit.html

```
newProduct.append('<span class="product-name">' +
  product.name + '</span>');
newProduct.append('<em class="product-price">' +
  product.price + '</em>');
newProduct.append('<div class="product-description">' +
  product.description + '</div>');

productsList.append(newProduct);
```

Использовать Mustache.js так же просто, как загружать скрипты. Одну из версий этого файла можно найти в архиве кодов данной книги; последнюю версию Mustache.js можно загрузить с сайта GitHub[1].

Как отобразить шаблон

Итак, мы хотим перестроить наше текущее приложение. Для начала нужно научиться отображать шаблоны с помощью Mustache. Самый простой способ это сделать — вызвать функцию `to_html()`.

```
Mustache.to_html(templateString, data);
```

Эта функция принимает два аргумента: первый — это строка-шаблон, по которой будет создаваться HTML-код, второй — данные, которые будут в нем использоваться. Переменная `data` представляет собой объект, поля которого становятся локальными переменными шаблона. Рассмотрим следующий код.

```
var artist = {name: "John Coltrane"};
var rendered = Mustache.to_html('<span class="artist name">{{ name }}</span>', artist);
$('body').append(rendered);
```

В переменной `rendered` содержится готовый HTML-код, который мы получили с помощью метода `to_html()`. Чтобы добавить в HTML-код свойство имени, Mustache использует теги и двойные фигурные скобки. Внутри фигурных скобок помещается название свойства. В последней строке полученный HTML-код добавляется в элемент `<body>`.

Это самый простой способ отобразить шаблон с помощью Mustache. И хотя в нашем примере будет больше кода (поскольку нам придется запрашивать данные у сервера), процесс создания шаблона в целом не изменится.

Замена старой системы на новую

Зная, как отобразить шаблон, мы можем удалить из приложения старый метод конкатенации строк. Давайте обратимся к нашему старому коду и посмотрим, какие фрагменты нужно удалить, а какие — заменить.

mustache/submit.html

```
function renderNewProduct() {
  var productsList = $('#products-list');

  var newProductForm = $('#new-product-form');

  var product = {};
  product.name = newProductForm.find('input[name*=name]').val();
  product.price = newProductForm.find('input[name*=price]').val();
  product.description =
    newProductForm.find('textarea[name*=description]').val();

  var newProduct = $('<li></li>');
  newProduct.append('<span class="product-name">' +
    product.name + '</span>');
```

[1] https://github.com/janl/mustache.js

```
  newProduct.append('<em class="product-price">' +
    product.price + '</em>');
  newProduct.append('<div class="product-description">' +
    product.description + '</div>');

  productsList.append(newProduct);

  productsList.find('input[type=text], textarea').each(function(input) {
    input.attr('value', '');
  });
}
```

Такой неаккуратный код не просто невозможно читать: с ним вообще очень трудно работать. Поэтому мы решили отказаться от метода jQuery `append()` и создать HTML-код, используя Mustache. Подумать только, с его помощью можно писать настоящий HTML-код и добавлять на страницу новые данные! Первым шагом к усмирению JavaScript будет построение специального шаблона. После этого одним простым движением мы добавим на страницу информацию о нашем товаре.

Если мы создадим элемент `<script>` и определим его контент как `text/template`, мы сможем поместить внутри код Mustache HTML и взять его для нашего шаблона. Чтобы можно было обращаться к этому элементу из JavaScript-кода с помощью jQuery, мы добавим к нему идентификатор.

```
<script type="text/template" id="product-template">
  <!-- шаблон HTML -->
</script>
```

Теперь напишем HTML-код для нашего шаблона. Раньше мы уже создавали для товара специальный объект, поэтому мы можем использовать для переменных шаблона те же имена, что и для свойств этого объекта.

```
<script type="text/template" id="product-template">
  <li>
    <span class="product-name">{{ name }}</span>
    <em class="product-price">{{ price }}</em>
    <div class="product-description">{{ description }}</div>
  </li>
</script>
```

После добавления шаблона мы можем вернуться к предыдущему коду, чтобы его переписать в соответствии с новым методом добавления HTML-кода. Ссылку на шаблон можно получить с помощью jQuery; для доступа к внутреннему контенту мы воспользуемся методом `html()`. После этого нам останется только передать Mustache HTML-код и данные.

```
var newProduct = Mustache.to_html( $('#product-template').html(), product);
```

Если мы посмотрим на результат, то увидим, что все выглядит довольно-таки неплохо. Единственный недостаток состоит в следующем: не стоит отображать поле с описанием, если описание не было получено с сервера; в таком случае нужно отменить добавление соответствующего элемента `<div>`. К счастью, Mustache допускает использова-

ние условных операторов: мы можем проверить, есть ли у данного товара описание, и отобразить нужный `<div>` только в случае положительного ответа.

```
{{#description}}
  <div class="product-description">{{ description }}</div>
{{/description}}
```

Этот оператор позволяет также работать с массивами: он автоматически проверяет, является ли свойство массивом, и если да, то проходит по всем его элементам.

ВОПРОС/ОТВЕТ. А МОЖНО ИСПОЛЬЗОВАТЬ ВНЕШНИЕ ШАБЛОНЫ?

Встраиваемые шаблоны очень удобны, однако если мы хотим отделить логику шаблонов от серверного представления, нам нужно будет создать на сервере специальную папку, в которой будут помещаться все файлы представления. Когда нам понадобится отобразить какой-то шаблон, мы с помощью jQuery отправим запрос GET и получим нужный шаблон.

```
$.get("http://mysite.com/js_views/external_template.html",
  function(template) {
    Mustache.to_html(template, data).appendTo("body");
  }
);
```

Так мы сможем работать с представлениями, не смешивая их с клиентскими представлениями.

Использование циклов

Раз уж нам удалось упростить код, отвечающий за добавление нового элемента в список товаров, мы решили пойти дальше и попытаться оптимизировать работу приложения: теперь мы хотим использовать JavaScript для отображения начальной страницы со списком товаров и комментариев к ним. Для этого мы создадим массив товаров по одному из свойств объекта данных; при этом у каждого элемента массива будет свойство `notes`. Последнее, в свою очередь, тоже будет массивом, и перебор его элементов будет выполняться внутри шаблона.

Для начала попробуем получить наши товары и отобразить их, предполагая, что сервер возвращает следующий JSON-массив.

mustache/index.html

```
$.getJSON('/products.json', function(products) {
  var data = {products: products};
  var rendered = Mustache.to_html($('#products-template').html(), data);
  $('body').append(rendered);
});
```

Теперь нам нужно создать шаблон, в соответствии с которым товары будут отображаться на странице. В случае Mustache пройти по элементам массива можно, передав этот массив с помощью оператора со знаком «решетки», например `{{#variable}}`. При этом вызываются свойства того объекта, который мы проходим в настоящий момент.

mustache/index.html

```
<script type="text/template" id="products-template">
  {{#products}}
     <li>
      <span class="product-name">{{ name }}</span>
      <em class="product-price">{{ price }}</em>
      <div class="product-description">{{ description }}</div>
      <ul class="product-notes">
        {{#notes}}
          <li>{{ text }}</li>
        {{/notes}}
      </ul>
    </li>
  {{/products}}
</script>
```

Теперь благодаря шаблонам и Mustache мы можем сделать так, чтобы начальная страница полностью генерировалась в браузере.

Шаблоны JavaScript — это прекрасный способ усовершенствовать структуру JavaScript-приложения. Теперь мы умеем отображать шаблоны, добавлять условия и перебирать элементы массива. Mustache.js — это простой способ избавиться от конкатенации строк и создать семантическую разметку на основе понятного HTML-кода.

Дополнительные возможности

Теперь мы знаем, что шаблоны Mustache позволяют создавать аккуратный код клиентской части приложения. Однако они также применимы и к языкам серверной части: существуют реализации Mustache на Ruby, Java, Python, ColdFusion и многих других. Более подробно об этом можно узнать на официальном сайте[1].

Таким образом, Mustache представляет собой инструмент для создания шаблонов как в серверной, так и в клиентской части приложения. Так, например, если представить ряд в HTML-таблице в виде шаблона Mustache и использовать его несколько раз (внутри цикла) для создания целой таблицы при загрузке начальной страницы, то потом его можно будет использовать снова, когда после успешного запроса Ajax вы захотите добавить в таблицу новый ряд.

Смотрите также

- Рецепт 11. Отображение данных с автоматической дозагрузкой
- Рецепт 13. Модные клиентские интерфейсы с помощью Knockout.js
- Рецепт 14. Как структурировать код с помощью Backbone.js
- Рецепт 20. Уведомление о состоянии сайта с помощью JavaScript и CouchDB

[1] http://mustache.github.com/

Рецепт 11. Отображение данных с автоматической дозагрузкой

Задача

Как для пользователя, так и для сервера важно, чтобы в каждый момент времени на экране не отображалось слишком много информации. Самое стандартное решение — деление контента на страницы (пагинация): при загрузке страницы пользователь будет видеть небольшой фрагмент контента, после чего сможет пролистывать содержимое по своему усмотрению. Таким образом, в каждый момент времени на экране будет отображаться лишь небольшая часть того, что на самом деле есть на вашем сайте.

На сегодняшний день известно, что пользователи тратят большую часть времени на последовательное пролистывание контента. В нашем примере пользователь наверняка захочет иметь возможность прокручивать весь список до тех пор, пока не найдет нужную информацию или не дойдет до конца списка. Мы попробуем реализовать это и на нашем сайте, стараясь минимизировать нагрузку на сервер.

Ингредиенты

- jQuery
- Mustache.js[1]
- QEDServer

Решение

Автоматическая дозагрузка позволяет размещать контент так, чтобы это было удобно пользователю, при эффективном использовании ресурсов. Теперь пользователю не нужно будет открывать следующую страницу с результатами поиска, тем самым перезагружая весь интерфейс; вместо этого мы загрузим следующую страницу с результатами в качестве фона и добавим эти результаты на страницу в тот момент, когда пользователь начнет прокручивать страницу дальше.

Мы хотим представить на нашем сайте всю линию продуктов компании, однако список слишком велик, чтобы сразу отображать его целиком. Поэтому нам, судя по всему, придется добавить пагинацию, так чтобы в каждый момент времени пользователь видел только десять товаров. Чтобы еще больше облегчить жизнь пользователям, мы собираемся ликвидировать кнопку Следующая страница (Next Page) и автоматически загружать следующую страницу, когда в этом будет необходимость. У пользователя должно сложиться впечатление, будто бы весь список товаров был загружен одновременно в тот момент, когда была открыта страница.

Чтобы создать рабочий прототип, мы воспользуемся QEDServer и его каталогом товаров. Весь наш код мы поместим в папку public в рабочей среде QEDServer. Для начала вам нужно будет создать файл products.html внутри папки public, которую QEDServer создаст сам. Более подробно о том, как работает QEDServer, рассказывается в разделе «QEDServer» (с. 17).

Чтобы наш код был чистым и аккуратным, мы обратимся к библиотеке Mustache Template, о которой говорилось ранее в рецепте 10. Для этого нам понадобится ее загрузить и поместить в папку public.

[1] http://github.com/documentcloud/underscore/blob/master/underscore.js

Мы начнем с создания простого HTML5-скелета в index.html, в котором подключим jQuery, библиотеку Mustache Template и файл endless_pagination.js. В последний будет помещаться код, реализующий разбиение на страницы.

endlesspagination/products.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>AwesomeCo Products</title>
    <link rel='stylesheet' href='endless_pagination.css'>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
    </script>
    <script type="text/javascript" src="mustache.js"></script>
    <script src="endless_pagination.js"></script>
  </head>
  <body>
    <div id="wrap">
      <header>
        <h1>Products</h1>
      </header>
    </div>
  </body>
</html>
```

В тело начальной страницы мы добавим заполнитель контента и изображение спиннера (рис. 2.8). Если пользователь все-таки дойдет до конца текущей страницы, спиннер подскажет ему, что контент в данный момент загружается (собственно, в этот момент он действительно должен загружаться).

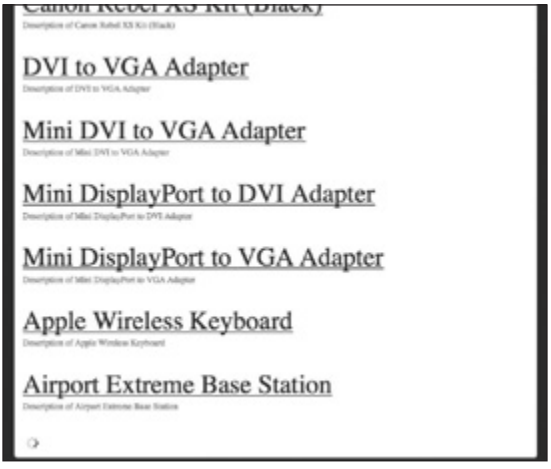

Рис. 2.8. Пользователь дошел до конца страницы

endlesspagination/products.html

```
<div id='content'>
  </div>
<img src='spinner.gif' id='next_page_spinner' />
```

API сервера QEDServer должен возвращать результат разбиения на страницы и отвечать на запросы JSON. Чтобы это проверить, зайдите на http://localhost:8080/products.json?page=2.

Зная, какие данные придут с сервера, мы можем перейти к написанию кода, с помощью которого будет обновляться наш интерфейс. Для этого мы создадим функцию, которая получает на входе массив JSON, размечает его с помощью шаблона Mustache и добавляет в конец страницы. Этот код мы поместим в файл с названием endless_pagination.js. Начнем с самых важных функций. Сначала напишем код, который будет превращать ответ JSON в HTML.

endlesspagination/endless_pagination.js

```
function loadData(data) {
  $('#content').append(Mustache.to_html("{{#products}} \
    <div class='product'> \
      <a href='/products/{{id}}'>{{name}}</a> \
      <br> \
      <span class='description'>{{description}}</span> \
    </div>{{/products}}", { products: data }));
}
```

Эта функция будет просматривать все товары и создавать отдельный `<div>` для каждого из них, чье имя является ссылкой. После этого новые элементы добавляются в конец списка и, таким образом, оказываются на странице.

Когда мы доходим до конца текущей страницы, мы хотим запросить содержимое следующей. Для этого нам нужно как-то узнать ее URL. Наше решение выглядит так: мы создадим глобальную переменную для адреса текущей страницы, и тогда в нужный момент, добавив соответствующие изменения, сможем получить адрес следующей.

endlesspagination/endless_pagination.js

```
var currentPage = 0;
function nextPageWithJSON() {
  currentPage += 1;
  var newURL = 'http://localhost:8080/products.json?page=' + currentPage;

  var splitHref = document.URL.split('?');
  var parameters = splitHref[1];
  if (parameters) {
    parameters = parameters.replace(/[?&]page=[^&]*/, '');
    newURL += '&' + parameters;
  }
  return newURL;
}
```

Функция `nextPageWithJSON()` увеличивает значение переменной `currentPage` на единицу и добавляет его в URL текущей страницы в качестве параметра page=. Кроме того, нам

нужно запомнить все остальные параметры текущего URL. При этом мы должны обязательно проверить, что предыдущее значение параметра page действительно было переопределено. Так мы сможем получить от сервера именно те данные, которые нам нужны.

Итак, у нас есть функции для отображения нового контента и получения URL следующей страницы. Теперь добавим функцию, которая будет отправлять запрос на сервер. По сути, это просто обращение к серверу через Ajax. Однако помимо этого нужно еще попытаться постараться предотвратить отправку лишних запросов. Для этого мы создадим глобальную переменную `loadingPage()` с начальным значением `0`. Будем увеличивать ее значение на единицу перед вызовом Ajax и присваивать начальное значение после этого. Таким образом мы создадим своего рода переключатель, или флажок. Если этого не сделать, наша функция может обращаться к серверу несколько десятков раз, и на каждый такой запрос сервер обязан будет ответить. Конечно же, это не то, что нам нужно.

endlesspagination/endless_pagination.js

```
var loadingPage = 0;
function getNextPage() {
  if (loadingPage != 0) return;

  loadingPage++;
  $.getJSON(nextPageWithJSON(), {}, updateContent).
    complete(function() { loadingPage-- });
}

function updateContent(response) {
  loadData(response);
}
```

После выполнения запроса Ajax мы передаем ответ функции `loadData()` (см. код, приведенный на с. 84). После добавления нового контента `loadData()` обновляет значение переменной `nextPage`, в которой хранится URL. После этого мы готовы к выполнению нового вызова Ajax.

Итак, у нас есть функция, которая запрашивает следующую страницу. Теперь нам нужно как-то определить, в какой момент это нужно сделать. В обычной ситуации сигналом для загрузки следующей страницы было бы нажатие ссылки Следующая страница (Next Page), но теперь нам нужно создать специальную функцию, которая будет возвращать значение `true`, когда на экране будет отображаться контент, находящийся на определенном расстоянии от нижнего конца страницы.

endlesspagination/endless_pagination.js

```
function readyForNextPage() {
  if (!$('#next_page_spinner').is(':visible')) return;

  var threshold = 200;
  var bottomPosition = $(window).scrollTop() + $(window).height();
  var distanceFromBottom = $(document).height() - bottomPosition;

  return distanceFromBottom <= threshold;
}
```

Наконец, мы добавим обработчик события прокрутки, который будет вызывать функцию `observeScroll()`. Каждый раз при прокрутке страницы мы будем вызывать дополнительную функцию `readyForNextPage()`, и как только она возвратит значение `true`, мы вызовем `getNextPage()`, которая и отправит запрос Ajax.

endlesspagination/endless_pagination.js

```
function observeScroll(event) {
  if (readyForNextPage()) getNextPage();
}

$(document).scroll(observeScroll);
```

КАК ЭТО РАБОТАЕТ В IE8

В таком виде этот код не будет работать в IE8. К сожалению, IE8 требует, чтобы заголовки запросов JSON имели специфический формат: например, «utf8» вместо «UTF-8». Запросы Ajax с неправильными заголовками не будут обработаны, и в результате на странице будет отображаться только спиннер. Об этом следует помнить, если вы используете JSON и хотите, чтобы сайт хорошо работал в IE.

Теперь мы наконец-то научились добавлять бесконечное количество контента. Однако на самом деле контент все же когда-то заканчивается. В этот момент нам нужно будет убрать спиннер, иначе пользователь может подумать, что либо интернет-соединение работает медленно, либо сайт работает неправильно. Для этого мы будем проверять, является ли список, полученный от сервера, пустым; если да, мы уберем спиннер.

endlesspagination/endless_pagination.js

```
function loadData(data) {
  $('#content').append(Mustache.to_html("{{#products}} \
  <div class='product'> \
    <a href='/products/{{id}}'>{{name}}</a> \
    <br> \
    <span class='description'>{{description}}</span> \
  </div>{{/products}}", { products: data }));
  if (data.length == 0) $('#next_page_spinner').hide();
}
```

Вот и все. Когда мы дойдем до конца списка, спиннера там уже не будет.

Дополнительные возможности

Этот прием превосходно подходит для оформления длинных списков; к тому же в последнее время он становится все более популярным как среди разработчиков, так и среди пользователей. Так как наша реализация составлена из нескольких отдельных функций, ее можно применять и в других ситуациях. Выбрав другое значение переменной `threshold`, мы можем изменить время загрузки контента; изменив функцию `loadData()`, мы можем отправлять HTML- и XML-запросы вместо запросов JSON. Но самое главное — мы можем спать спокойно, зная, что если jQuery не будет работать, контент никуда не потеряется. Это можно легко проверить, отключив JavaScript.

В следующем рецепте мы попробуем усовершенствовать этот код так, чтобы пользователю было удобнее работать на нашем сайте, — например, добавим возможность изменения URL и кнопку Назад (Back).

Смотрите также

- Рецепт 12. Ajax с контролем состояний
- Рецепт 10. HTML с помощью Mustache

Рецепт 12. Ajax с контролем состояний

Задача

Одно из потрясающих преимуществ Интернета — возможность обмениваться ссылками. Однако с появлением сайтов, основанных на Ajax, ситуация изменилась в худшую сторону: теперь нажатие на ссылке Ajax совсем не обязательно приводит к обновлению URL браузера. Это не только осложняет обмен ссылками, но и делает бесполезной кнопку Назад (Back). Такие сайты не являются «добропорядочными жителями Глобальной сети»: например, после завершения сессии невозможно определить, какую страницу вы посетили последней.

К сожалению, та реализация автоматической дозагрузки, над которой мы работали в предыдущем разделе (см. «Рецепт 11. Отображение данных с автоматической дозагрузкой»), тоже не является «добропорядочным жителем Глобальной сети». Когда при прокрутке страницы мы запрашиваем новый контент через Ajax, URL браузера не меняется. Но не нужно забывать, что при этом состояние страницы меняется, так как после дозагрузки на ней отображается не тот контент, который был при первоначальной загрузке. Предположим, например, что нам понравился товар со страницы 5 и мы хотим отправить своему другу ссылку на него. Проблема в том, что когда наш друг откроет эту ссылку, он увидит другой список товаров и не сможет понять, какой из них понравился нам.

Но это еще не все. Рассмотрим следующую ситуацию: пользователь находится на сайте, полностью основанном на Ajax, и щелкает мышью на кнопке браузера Назад (Back). К сожалению, пользователь попадает не туда, куда он хотел, а на какую-то другую страницу, в которой есть ссылка на эту. Он начинает нервничать и щелкает на кнопке Вперед (Forward), в результате чего полностью перестает понимать, где он находится. Но, к счастью, в нашем арсенале есть прекрасное решение этой проблемы.

Ингредиенты

- jQuery
- Mustache.js[1]
- QEDServer

Решение

Мы решили вернуться к задаче из рецепта 11 («Отображение данных с автоматической дозагрузкой») и усовершенствовать ее решение. Реализация, предложенная в предыдущем разделе, не дает возможности обмениваться ссылками. Чтобы наша карма в среде веб-разработчиков не упала слишком низко и чтобы пользователи чувствовали себя комфортно, мы хотим реализовать на нашей странице со списком контроль состояний: если во время прокрутки мы будем переходить на новую страницу, URL будет меняться. В спецификации HTML5 существует JavaScript-функция `pushState()`, которая позволяет менять URL, оставаясь на одной и той же странице, в большинстве браузеров. Надо сказать, это прекрасная новость для веб-разработчиков! Веб-приложения, полностью основанные на Ajax, больше не должны работать по схеме запрос/загрузка. Одновременно с этим мы получаем ряд важных преимуществ. В частности, при переходе

[1] https://github.com/documentcloud/underscore/blob/master/underscore.js

к следующей странице не нужно заново загружать HTML-код заголовка и футера, лишний раз запрашивать изображения, таблицы стилей и JavaScript-файлы. Пользователи, в свою очередь, смогут легко обмениваться URL и обновлять страницу без негативных последствий. Но самое важное — кнопка Назад (Back) наконец-то будет работать.

Использование функции pushState

Функция `pushState()` пока еще находится в состоянии разработки. Как правило, старые версии браузеров ее не поддерживают, однако для них существуют специальное решение, использующее ту часть URL, которая идет после знака решетки. Это решение хоть и работает, но выглядит совершенно ужасно. И дело не только в том, какие URL получаются в результате. Интернет обладает превосходной долговременной памятью. Он создавался не только для того, чтобы вы могли отправлять своей бабушке ссылки на смешных говорящих котят, но и для того, чтобы можно было находить контент по ссылке, сохраненной несколько лет назад, даже если сервер уже давно переехал (при условии что создатели этого контента тоже были «добропорядочными жителями Глобальной сети» и сделали так, чтобы старый URL возвращал нужный код состояния HTTP 301). Если мы будем использовать URL со знаком решетки в качестве временной меры, мы замучаемся до скончания времен поддерживать эти устаревшие ссылки в рабочем состоянии[1]. Поскольку URL с решетками никогда не передаются на сервер, наше приложение вынуждено будет перенаправлять трафик, когда функция `pushState()` будет официально принята.

Теперь давайте посмотрим, что нужно сделать, чтобы добавить к нашей странице с бесконечным списком контроль состояний.

Какие параметры необходимо учитывать

Так как мы не знаем, какую страницу пользователь захочет загрузить при первом же запросе, мы должны хранить в памяти начальную страницу и текущую страницу. Если, например, пользователь сразу открыл страницу 3, то нам нужно обеспечить возможность вернуться к этой странице позже. Если он воспользовался прокруткой и перешел со страницы 3, скажем, на страницу 7, загрузив при этом все промежуточные страницы, это нам тоже нужно запомнить. Чтобы при обновлении страницы пользователю не пришлось снова выполнять прокрутку, мы должны хранить в памяти первую и последнюю страницы.

Далее нам нужно придумать, как отправлять информацию о первой и последней страницах. Наиболее простой способ — добавить эти параметры в URL во время запроса get. При первой загрузке страницы мы зададим значение параметра page в URL, равное номеру текущей страницы, и будем считать, что пользователь хочет посмотреть только эту страницу. Если вдобавок клиент передаст значение параметра start_page, мы поймем, что пользователь хочет просмотреть несколько страниц, начиная от start_page и заканчивая page. Таким образом, возвращаясь к старому примеру, если мы находимся на странице 7, но пришли туда со страницы 3, наш URL будет выглядеть так: http://localhost:8080/products?start_page=3&page=7.

Такого набора параметров должно быть достаточно для того, чтобы воссоздать список товаров и открыть в точности ту страницу, которую пользователь видел при первом обращении по этому URL.

---

[1] http://danwebb.net/2011/5/28/it-is-about-the-hashbangs

statefulpagination/stateful_pagination.js

```
function getParameterByName(name) {
  var match = RegExp('[?&]' + name + '=([^&]*)')
    .exec(window.location.search);

  return match && decodeURIComponent(match[1].replace(/\+/g, ' '));
}

var currentPage = 0;
var startPage = 0;

$(function() {
  startPage = parseInt(getParameterByName('start_page'));
  if (isNaN(startPage)) {
    startPage = parseInt(getParameterByName('page'));
  }
  if (isNaN(startPage)) {
    startPage = 1;
  }
  currentPage = startPage - 1;

  if (getParameterByName('page')) {
    endPage = parseInt(getParameterByName('page'));
    for (i = currentPage; i < endPage; i++) {
      getNextPage(true);
    }
  }

  observeScroll();
});
```

В этом фрагменте кода не происходит ничего необычного: мы просто определяем значения start_page и current_page и запрашиваем их у сервера. При этом мы используем функцию, очень похожую на `getNextPage()` из предыдущего раздела; единственное отличие состоит в том, что здесь нет запрета на отправку сразу нескольких запросов. Раньше это было необходимо для того, чтобы во время прокрутки избежать многократного обращения к одному и тому же контенту. Но теперь эта проблема уже не актуальна, так как мы точно знаем, какие страницы хотим загрузить.

Подобно тому, как мы следили за изменением значения `currentPage` (см. код, приведенный на с. 84), теперь мы будем делать то же самое с переменной `startPage`. Этот параметр мы вытащим из URL, после чего сможем обращаться к тем страницам, которые еще не загружались. И хотя его значение останется неизменным, мы все же хотим проследить за тем, чтобы оно было обязательно добавлено в URL и оставалось там при каждом обращении к новой странице.

Обновление URL браузера

Чтобы обновить URL, мы напишем функцию `updateBrowserUrl()`, которая будет вызывать `pushState()` и задавать значения параметров start_page и page. При этом важно

помнить, что не все браузеры поддерживают `pushState()`; это нужно проверить перед вызовом функции. И хотя в таких браузерах это решение работать не будет, никто не мешает нам предусмотреть возможность появления поддержки `pushState()` в будущем.

statefulpagination/stateful_pagination.js

```
function updateBrowserUrl() {
  if (window.history.pushState == undefined) return;

  var newURL = '?start_page=' + startPage + '&page=' + currentPage;
  window.history.pushState({}, '', newURL);
}
```

Функция `pushState()` содержит три параметра. Первый — объект состояния; обычно это объект JSON. Так как во время прокрутки мы получаем JSON от сервера, эту информацию можно было бы хранить именно здесь. Но в нашем случае эта стратегия неуместна, поскольку данные проще получить с сервера. На данном этапе мы передадим в качестве параметра пустой список. Второй аргумент — это строка, с помощью которой будет обновляться заголовок браузера. Пока эта возможность почти нигде не реализована, но даже в противном случае мы не стали бы ее использовать: нам не нужно обновлять заголовок браузера. Так что здесь мы снова оставляем аргумент пустым, а именно передаем пустую строку.

А теперь перейдем к самой сути функции `pushState()`. С помощью третьего параметра мы указываем, как должен измениться URL. Здесь можно задать либо абсолютный путь, либо параметры URL, значения которых необходимо обновить. По соображениям безопасности мы не можем менять домен URL, но зато можем легко менять все, что находится после домена верхнего уровня. Поскольку нас интересуют только параметры URL, в начале третьего аргумента `pushState()` мы ставим знак `?`. Наконец, мы задаем параметры start_page и page; если они уже заданы, функция `pushState()`обновит их сама.

statefulpagination/stateful_pagination.js

```
function updateContent(response) {
  loadData(response);
  updateBrowserUrl();
}
```

Напоследок мы добавим в функцию `updateContent()` вызов `updateBrowserUrl()`. Теперь код, реализующий автоматическую дозагрузку контента, будет осуществлять контроль состояния. Это значит, что если пользователь перейдет на другую страницу с помощью кнопки Назад (Back), он сможет легко вернуться обратно с помощью кнопки Вперед (Forward). Кроме того, теперь можно не бояться обновлять страницу, но что самое важное — делиться ссылками. Таким образом, мы превратили нашу страничку в добропорядочного жителя Глобальной сети, причем благодаря усилиям разработчиков современных браузеров это не потребовало больших усилий.

Дополнительные возможности

Создавая веб-страницы с помощью JavaScript и Ajax, мы должны внимательно следить за тем, как работают наши интерфейсы. Метод `pushState()` и History API позволяют

легко наладить работу стандартных элементов управления браузера, к которым пользователи уже успели привыкнуть. С уровнями абстракции наподобие History.js[1] это становится еще проще, так как в них предусмотрены изящные обходные пути для работы с браузерами, не поддерживающими History API.

Идеи, о которых мы говорили в этом разделе, активно проникают и в другие JavaScript-фреймворки, такие как Backbone.js. В итоге это приведет к тому, что кнопку Назад (Back) можно будет использовать и в более сложных одностраничных приложениях.

Смотрите также

- Рецепт 10. HTML с помощью Mustache
- Рецепт 12. Ajax с контролем состояний
- Рецепт 14. Как структурировать код с помощью Backbone.js

[1] http://archive.plugins.jquery.com/plugin-tags/pushstate

Рецепт 13. Модные клиентские интерфейсы с помощью Knockout.js

Задача

Когда в результате каких-либо действий пользователя возникает необходимость изменить контент, современные приложения, как правило, предпочитают обновлять не всю страницу, а только ее часть: во-первых, обращения к серверу часто бывают слишком затратными, а во-вторых, после обновления всей страницы пользователь может просто потеряться в ее содержимом.

К сожалению, в такой ситуации JavaScript может сделать ваш код неудобным в использовании. Если большую часть времени вам нужно просто следить за несколькими событиями, то в один прекрасный момент вам может понадобиться обновить сразу несколько фрагментов страницы, что тут же превратится в настоящий кошмар.

Knockuot — это удобный шаблон, который позволяет связывать элементы интерфейса с объектами и таким образом автоматически обновлять одни элементы интерфейса при изменении других без использования вложенных обработчиков событий.

Ингредиенты

- Knockout.js[1]

Решение

Шаблон Knockout.js использует *модели представления*, которые включают в себя существенную часть логики представления, относящейся к изменениям интерфейса. Свойства этих моделей соотносятся с элементами самого интерфейса.

Мы хотим, чтобы наши клиенты могли изменять количество товаров в корзине и обновлять общую сумму в режиме реального времени. Модели представления Knockout и их связи с данными понадобятся нам для обновления корзины. В ней будет отдельная строка для каждого товара, поле для обновления общей суммы и кнопка для удаления товара из корзины. При любых изменениях списка товаров мы будем обновлять как промежуточную сумму в каждой строке, так и общую сумму. В результате мы получим нечто похожее на интерфейс, представленный на рис. 2.9.

Product	Price	Quantity	Total	
Macbook Pro 15 inch	1699	1	1699	Remove
Mini Display Port to VGA Adapter	29	1	29	Remove
Magic Trackpad	69	1	69	Remove
Apple Wireless Keyboard	69	1	69	Remove
Total			1866	

Рис. 2.9. Как выглядит корзина

[1] http://knockoutjs.com/

Общие сведения о Knockout

«Модели представления» Knockout — это самые обычные JavaScript-объекты со свойствами и методами, а также несколькими специальными ключевыми словами. Для примера рассмотрим простой объект Person (Персона) с методами, возвращающими имя, фамилию и полное имя.

knockout/binding.html

```
var Person = function(){
  this.firstname = ko.observable("John");
  this.lastname = ko.observable("Smith");
  this.fullname = ko.dependentObservable(function(){
    return(
       this.firstname() + " " + this.lastname()
    );
  }, this);
};

ko.applyBindings( new Person );
```

Чтобы привязать методы и логику объекта к элементу интерфейса, мы используем HTML5 атрибуты `data-`.

knockout/binding.html

```
<p>First name: <input type="text" data-bind="value: firstname"></p>
<p>Last name: <input type="text" data-bind="value: lastname"></p>
<p>Full name:
  <span aria-live="polite" data-bind="text: fullname"></span>
</p>
```

Здесь при изменении имени или фамилии в соответствующем текстовом окне полное имя обновляется автоматически. Однако динамическое обновление может вызвать некоторые проблемы у пользователей программ экранного доступа. Чтобы сообщить таким программам о том, что данный элемент интерфейса меняется динамически, мы добавляем к нему атрибут `aria-live`.

Пока это все довольно-таки просто. Теперь мы немного усложним задачу и оформим с помощью Knockout одну строку корзины так, чтобы при изменении количества товаров менялась общая сумма. Потом на ее основе можно будет создать полноценную корзину. Начнем с модели данных.

Строку корзины мы представим в виде простого JavaScript-объекта `LineItem` с полями `name` (название) и `price` (стоимость). Создайте новую HTML-страницу и подключите библиотеку Knockout.js внутри раздела `<head>`.

knockout/item.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Update Quantities</title>
    <script type="text/javascript" src="knockout-1.3.0.js"></script>
```

```
  </head>

  <body>
  </body>

</html>
```

Добавьте блок `<script>` внизу страницы перед закрывающим тегом `<body>` и поместите туда следующий код.

knockout/item.html

```
var LineItem = function(product_name, product_price){
  this.name = product_name;
  this.price = product_price;
};
```

В JavaScript функции являются конструкторами объектов, поэтому с помощью функции мы можем сымитировать создание класса. В нашем случае конструктор класса присвоит значения `name` и `price` в момент создания нового экземпляра `LineItem`.

Далее нам нужно сообщить Knockout, что мы хотим использовать `LineItem` в качестве модели представления, и тогда к ее свойствам можно будет обращаться в HTML-разметке. Для этого добавим в блок `script` еще один вызов.

knockout/item.html

```
var item = new LineItem("Macbook Pro 15", 1699.00);
ko.applyBindings(item);
```

Мы создаем новый экземпляр `LineItem` и передаем его методу `applyBindings()`; кроме того, мы задаем название и стоимость товара. Этот процесс станет динамическим позже, а пока мы будем считать эти значения заданными.

Итак, у нас есть готовый объект, и теперь мы можем создать наш интерфейс, используя данные из этого объекта. Разметка нашей корзины будет основана на HTML-таблице; также нам потребуются элементы `<thead>` и `<tbody>`.

knockout/item.html

```
<div role="application">
  <table>
    <thead>
      <tr>
        <th>Product</th>
        <th>Price</th>
        <th>Quantity</th>
        <th>Total</th>
      </tr>
    </thead>
    <tbody>
      <tr aria-live="polite">
        <td data-bind="text: name"></td>
        <td data-bind="text: price"></td>
```

продолжение ⇨

```
      </tr>
    </tbody>
  </table>
</div>
```

Так как строки нашей таблицы будут обновляться в результате определенных действий пользователя, мы добавим к ним атрибут `aria-live`: теперь программы экранного доступа будут знать, что этот контент может меняться. Чтобы сообщить таким программам о том, что данное приложение является интерактивным, мы поместили всю корзину в дополнительный `<div>`, для которого задали ARIA-роль HTML5 `application`. Более подробно об этом можно узнать в спецификации HTML5[1].

Обратите внимание на следующие две строки.

knockout/item.html

```
<td data-bind="text: name"></td>
<td data-bind="text: price"></td>
```

Теперь наш экземпляр `LineItem` является глобальным отображаемым объектом страницы, а следовательно, его свойства `name` и `price` также являются отображаемыми. Поэтому в этих двух строках кода мы можем присвоить параметру `text` элемента значение нужного нам свойства.

Загрузив эту страницу в браузере, мы можем увидеть, что строка таблицы начинает обретать форму и в ней уже заполнены название и стоимость.

Теперь добавим в таблицу текстовое поле, в котором пользователь сможет изменить число товаров.

knockout/item.html

```
<td><input type="text" name="quantity"
      data-bind='value: quantity, valueUpdate: "keyup"'>
</td>
```

В Knockout мы обращаемся к полям данных обычных HTML-элементов с помощью text, однако у некоторых HTML-форм, таких как `input`, есть атрибуты `value`. Поэтому здесь мы устанавливаем связь между атрибутом `value` и свойством модели представления `quantity`, которое нам еще предстоит определить.

Свойство quantity нужно не только для того, чтобы отображать данные, но и для того, чтобы их задавать; все события будут запускаться именно в момент задания данных. Чтобы задать значение свойства `quantity` в нашем классе, мы будем использовать функцию Knockout `ko.observable()`.

knockout/item.html

```
this.quantity = ko.observable(1);
```

Чтобы наше текстовое поле имело какое-то значение `quantity` при первой загрузке страницы, мы передаем функции `ko.observable()` значение по умолчанию.

Теперь можно вводить количество товаров. Но мы хотели еще показать промежуточную сумму. Чтобы это сделать, добавим в таблицу еще один столбец.

[1] http://www.w3.org/TR/html5-author/wai-aria.html

knockout/item.html

```
<td data-bind="text: subtotal "></td>
```

Как и в случае со столбцами `name` и `price`, в качестве текста для ячейки таблицы мы берем значение свойства `subtotal` из модели представления.

Наконец мы подошли к одной из самых потрясающих возможностей Knockout — методу `dependentObservable()`. Свойство `quantity` мы определили как «наблюдаемое» (observable); это значит, что если в нем что-то изменится, другие элементы это заметят. Теперь мы объявляем функцию `dependentObservable()`, которая запускает определенный код в случае, если «наблюдаемое» поле меняется. Значение этой функции мы прикрепляем к свойству нашего объекта, чтобы связать его с пользовательским интерфейсом.

knockout/item.html

```
this.subtotal = ko.dependentObservable(function() {
  return(
    this.price * parseInt("0"+this.quantity(), 10)
  ); //<label id="code.subtotal" />
}, this);
```

Но откуда функция `dependentObservable()` знает, за какими полями нужно наблюдать? На самом деле она просто смотрит на то, к каким «наблюдаемым» свойствам мы обращаемся в функции, которую сами определили. Так как мы производим определенные операции со стоимостью и количеством, Knockout будет следить за обоими этими параметрами и запускать этот код каждый раз, когда какой-то из них меняется.

У функции `dependentObservable()` есть второй параметр: он задает контекст для тех свойств, к которым мы обращаемся внутри функции. Это связано с устройством функций и объектов JavaScript; более подробно об этом можно прочитать в документации к Knockout.js.

Теперь один ряд готов. При изменении количества товаров стоимость обновляется в режиме реального времени. Теперь самое время собрать все, чему мы научились, и создать полноценную корзину с промежуточными суммами в каждой строке и одной общей суммой.

Связывание потока управления

Мы уже поняли, что связывание объектов с HTML — очень удобный механизм. Тем не менее вполне вероятно, что в нашей корзине окажется более одного товара, и тогда нам придется не только много раз переписывать один и тот же код, но и связывать элементы интерфейса с несколькими объектами `LineItem`. Поскольку это может оказаться достаточно трудным и утомительным, мы попробуем перестроить наш интерфейс так, чтобы этого избежать.

Мы предлагаем отказаться от использования `LineItem` в качестве модели представления и создать новый объект для всей корзины. В новом объекте `Cart` мы будем хранить все наши объекты `LineItem`. Мы уже достаточно знаем о `dependentObsevables`, чтобы понять, что к этому объекту можно добавить свойство, вычисляющее общую сумму каждый раз, когда какой-то из элементов списка меняется.

А как будет выглядеть HTML-разметка для одной строки? Чтобы не писать один и тот же код несколько раз, мы воспользуемся связыванием потока управления и заставим Knockout использовать нашу HTML-разметку отдельной строки для каждого товара в корзине. Давайте начнем.

Для начала определим массив товаров, которые наполнят нашу корзину.

knockout/update_cart.html

```
var products = [
  {name: "Macbook Pro 15 inch", price: 1699.00},
  {name: "Mini Display Port to VGA Adapter", price: 29.00},
  {name: "Magic Trackpad", price: 69.00},
  {name: "Apple Wireless Keyboard", price: 69.00}
];
```

В реальной ситуации мы бы получили эти данные с помощью веб-службы или вызова Ajax либо сгенерировали бы этот массив в серверной части.

Теперь давайте создадим объект `Cart` и поместим в него все товары. Здесь мы будем использовать те же операции, что и при создании `LineItem`.

knockout/update_cart.html

```
var Cart = function(items){
  this.items = ko.observableArray();

  for(var i in items){
    var item = new LineItem(items[i].name, items[i].price);
    this.items.push(item);
  }
}
```

Еще нам нужно поменять объект связывания: теперь это будет класс `Cart`, а не класс `LineItem`.

knockout/update_cart.html

```
var cartViewModel = new Cart(products);
ko.applyBindings(cartViewModel);
```

Чтобы объединить все товары в одной корзине, мы будем использовать `observableArray()` — аналог `observable()` со свойствами массива. При создании нового экземпляра корзины мы передали массив данных. Затем объект пройдет по всем элементам этого массива и создаст для каждого из них новый экземпляр `LineItem`, который будет добавлен в массив товаров. Так как этот массив является «наблюдаемым», пользовательский интерфейс будет реагировать на любые изменения его содержимого. Поскольку теперь в нашей корзине не один элемент, а несколько, нам придется изменить соответствующий код.

Чтобы добавить в таблицу несколько однотипных строк, мы воспользуемся возможностями Knockout, а именно вызовем `data-bind` для тега `<tbody>`.

knockout/update_cart.html

```
<tbody data-bind="foreach: items">
  <tr aria-live="polite">
    <td data-bind="text: name"></td>
    <td data-bind="text: price"></td>
```

```
    <td><input type="text" name="quantity" data-bind='value: quantity'></td>
    <td data-bind="text: subtotal "></td>
  </tr>
</tbody>
```

ВОПРОС/ОТВЕТ. ЧТО МОЖНО СКАЗАТЬ О ДОСТУПНОСТИ KNOCKOUT?

И хотя многие JavaScript-интерфейсы, услышав слово «доступность», тут же начинают бить тревогу, само по себе использование JavaScript не мешает людям с ограниченными возможностями обращаться к контенту сайта.

В этом рецепте мы попытались сделать наше приложение более понятным для программ экранного доступа с помощью ARIA-ролей и атрибутов HTML5. Однако понятие «доступность» далеко не ограничивается такими программами: мы должны стремиться к тому, чтобы с нашим приложением могла работать максимально широкая аудитория пользователей.

С помощью Knockout мы реализуем решения, основанные на JavaScript; таким образом, они будут работать, только если JavaScript подключен; никогда не следует об этом забывать. Поэтому мы рекомендуем сначала создавать приложение, работающее без JavaScript, а потом уже заниматься разного рода усовершенствованиями. В нашем примере мы отображаем содержимое корзины с помощью Knockout. Но если бы мы использовали серверный фреймворк, мы могли бы с его помощью создать HTML-код корзины, а потом уже поверх него добавить связывание Knockout. Таким образом, доступность сайта зависит не от библиотек и технологий, а скорее от практической реализации.

Мы сообщаем Knockout, что он должен отобразить на странице отдельный элемент `<tbody>` для каждого товара в массиве `items`. Кроме этого в строке менять ничего не нужно.

Итак, у нас есть таблица из нескольких строк, в каждой из которых автоматически вычисляется промежуточная сумма. Теперь нам нужно научиться вычислять общую сумму и удалять товары из корзины.

Общая сумма

Мы уже видели метод `dependentObservable()` в работе: с его помощью мы вычисляли промежуточную сумму в каждой строке. Точно так же, добавив `dependentObservable()` к самому объекту `Cart`, мы можем вычислить и общую сумму.

knockout/update_cart.html

```
this.total = ko.dependentObservable(function(){
  var total = 0;
  for (item in this.items()){
    total += this.items()[item].subtotal();
  }
  return total;
}, this);
```

Этот код будет запускаться каждый раз, когда в каком-то из элементов массива будут происходить изменения. В таблице для общей суммы потребуется отдельный ряд. Но поскольку эта сумма относится не к отдельному ряду, а ко всей таблице, мы не станем

помещать ее в тег `<tbody>`, а воспользуемся специальным тегом `<tfoot>`, который, в свою очередь, поместим непосредственно перед закрывающим тегом `<thead>`. Когда футер расположен *выше* тела таблицы, браузерам и вспомогательным устройствам удается быстрее и лучше понять структуру таблицы.

knockout/update_cart.html

```
<tfoot>
  <tr>
    <td colspan="4">Total</td>
    <td aria-live="polite" data-bind="text: total()"></td>
  </tr>
</tfoot>
```

Теперь мы можем менять количество товаров в корзине, и при этом будут одновременно обновляться два параметра: соответствующая промежуточная сумма и общая сумма. Далее мы займемся кнопкой Удалить (Remove).

Удаление товаров из корзины

Для полноты картины нам остается добавить в конце каждой строки кнопку Удалить (Remove), которая будет убирать товар из корзины. К счастью, на данном этапе эта задача сводится всего лишь к нескольким элементарным действиям. Для начала нам придется изменить таблицу, добавив в нее соответствующие кнопки.

knockout/update_cart.html

```
<td>
  <button
    data-bind="click: function() { cartViewModel.remove(this) }">Remove
  </button>
</td>
```

НЕ ЗАБУДЬТЕ О СИНХРОНИЗАЦИИ С СЕРВЕРОМ

В последнее время веб-разработчики предпочитают создавать корзины, обновляющиеся в реальном времени, не выходя за рамки клиентской части приложения. В некоторых случаях отправка Ajax-запросов туда и обратно после каждого действия пользователя оказывается невозможной.

Если вы решили придерживаться такого подхода, вам понадобится синхронизировать данные корзины в клиентской части приложения с данными сервера. Вы же не хотите, чтобы кто-то менял за вас цены!

Когда пользователь соглашается оплатить товары, отправьте на сервер данные об их количестве и выполните повторное вычисление их общей стоимости; только после этого можно приступать к оплате.

На этот раз мы связываем не данные и интерфейс, а событие и функцию. В этом случае мы передаем товар (`this`) методу `remove()`, вызванному для нашего объекта `cartViewModel`. Однако пока наша кнопка работать не будет, так как мы еще не определили метод `remove()`. Давайте добавим этот метод в объект `Cart`.

knockout/update_cart.html

```
this.remove = function(item){ this.items.remove(item); }
```

Ну вот и все. Так как массив `items` относится к типу `observableArray`, весь интерфейс будет обновляться при любых его изменениях.

Дополнительные возможности

Knockout прекрасно подходит в ситуациях, когда нам нужно создать динамический интерфейс, умещающийся на одной странице. А поскольку он не требует привязки к специальным веб-фреймворкам, его можно применять практически всегда.

Однако гораздо важнее другое: модели представления Knockout используют самый обычный JavaScript, и, следовательно, с его помощью мы можем реализовать многие из стандартных возможностей пользовательского интерфейса: например, живой поиск на основе Ajax, элементы управления для правки на месте с передачей данных обратно на сервер или даже обновление контента одного раскрывающегося поля при изменении данных другого поля.

Смотрите также

- Рецепт 14. Как структурировать код с помощью Backbone.js

Рецепт 14. Как структурировать код с помощью Backbone.js

Задача

Потребность в более надежных и интерактивных клиентских интерфейсах побуждает разработчиков создавать новые и потрясающие JavaScript-библиотеки. Но по мере того как приложения расширяются и усложняются, код клиентской части становится похожим на ящик для мелочей, в котором беспорядочно валяются библиотеки вперемешку со связями событий, запросами jQuery Ajax и функциями разбора JSON.

Нам необходим способ разработки клиентских приложений, основанный на том подходе, который уже много лет применяется при написании кода для серверной части, — а именно фреймворк. Надежный JavaScript-фреймворк позволит нам структурировать код и избежать копирования его фрагментов. Кроме того, это большой шаг на пути к стандартизации: благодаря фреймворку наш код станет более понятным для других разработчиков.

Поскольку Backbone представляет собой достаточно сложную библиотеку, этот рецепт будет гораздо длиннее и, вполне возможно, вызовет некоторые трудности.

Ингредиенты

- Backbone.js[1]
- Underscore.js[2]
- JSON2.js[3]
- Mustache[4]
- jQuery
- QEDServer

Решение

Для решения этой задачи подходит несколько разных фреймворков. Из них мы выбрали именно Backbone.js благодаря его гибкости, надежности и качеству кода, при том что на момент написания этой книги он был еще сравнительно новым. Аналог связывания событий, о котором мы говорили в разделе «Рецепт 13. Модные клиентские интерфейсы с помощью Knockout.js», есть и в Backbone, однако здесь мы устанавливаем связь между моделью и сервером, а за изменениями URL следит система маршрутизации запросов. Таким образом, в нашем распоряжении есть мощный фреймворк, который прекрасно подойдет для сложных клиент-серверных приложений, однако может оказаться слишком громоздким для более простых приложений.

С помощью Backbone мы хотим усовершенствовать интерфейс нашего интернет-магазина, сделав его более чувствительным к изменениям. Как показывают данные сетевых

[1] http://documentcloud.github.com/backbone/
[2] http://documentcloud.github.com/underscore/
[3] https://github.com/douglascrockford/JSON-js
[4] http://mustache.github.com/

журналов и результаты изучения аудитории пользователей, обновление страницы занимает слишком много времени, а многое из того, с чем мы обращаемся к серверу, можно было бы сделать, не выходя за рамки клиентской части приложения. Менеджер предложил нам взять интерфейс для управления продуктом и сделать из него одностраничный интерфейс, в котором можно было бы добавлять и удалять товары, не обновляя всю страницу.

Однако перед тем как перейти к созданию такого интерфейса, мы хотим разобраться в том, что такое Backbone и как с его помощью можно решить нашу задачу.

Общие сведения о Backbone

Backbone — это клиентская реализация схемы Model–View–Controller (Модель–представление–поведение); существенное влияние на нее оказали и различные серверные фреймворки, такие как ASP.NET MVC и Ruby on Rails. Backbone включает несколько компонентов, которые позволяют наладить удобное взаимодействие с серверным кодом.

Модели используются для представления данных и осуществляют взаимодействие с серверной частью приложения с помощью Ajax. Также в них можно разместить бизнес-логику и средства проверки данных.

Представления Backbone несколько отличаются от представлений других фреймворков. Здесь они не составляют уровень представления, а скорее являются «контроллерами представления». В обычном клиентском интерфейсе может быть много разных объектов; код, который эти объекты запускают, располагается как раз в этих самых представлениях. С их помощью можно отображать данные по шаблону и изменять пользовательский интерфейс.

Маршрутизаторы, следящие за изменениями URL, могут устанавливать связи между моделями и представлениями. Когда мы хотим использовать в одном интерфейсе несколько «страниц» или вкладок, мы можем заставить маршрутизатор обрабатывать запросы и отображать соответствующие представления. Также Backbone обеспечивает поддержку кнопки Назад (Back) в браузере.

Наконец, в Backbone можно создавать коллекции, которые сильно упрощают работу с многочисленными экземплярами моделей. На рис. 2.10 показано, как эти компоненты взаимодействуют друг с другом и как с их помощью можно создать интерфейс управления продукцией.

По умолчанию модели Backbone используют метод jQuery `ajax()` для обмена данными с серверным RESTful-приложением посредством JSON. Для этого серверная часть должна уметь принимать запросы GET, POST, PUT и DELETE, а также находить JSON в теле запроса. Однако так происходит только по умолчанию — в документации Backbone рассказывается о том, как изменить код клиентской части в зависимости от конфигурации серверной части.

Та серверная часть, с которой мы будем работать, не требует никаких изменений. Поэтому мы сможем просто вызывать для наших моделей разные методы, тогда как Backbone будет аккуратно выполнять сериализацию и десериализацию данных о товаре.

И последнее: как мы уже говорили в примечании «Что можно сказать о доступности Knockout?», такие фреймворки лучше использовать *поверх* уже готового сайта, то есть в качестве дополнения к интерфейсу. Если код клиентской части строится на прочном фундаменте, гораздо проще реализовать решение, не основанное на JavaScript. Работая над этой задачей, мы будем считать, что у нас уже есть интерфейс, работающий без использования JavaScript.

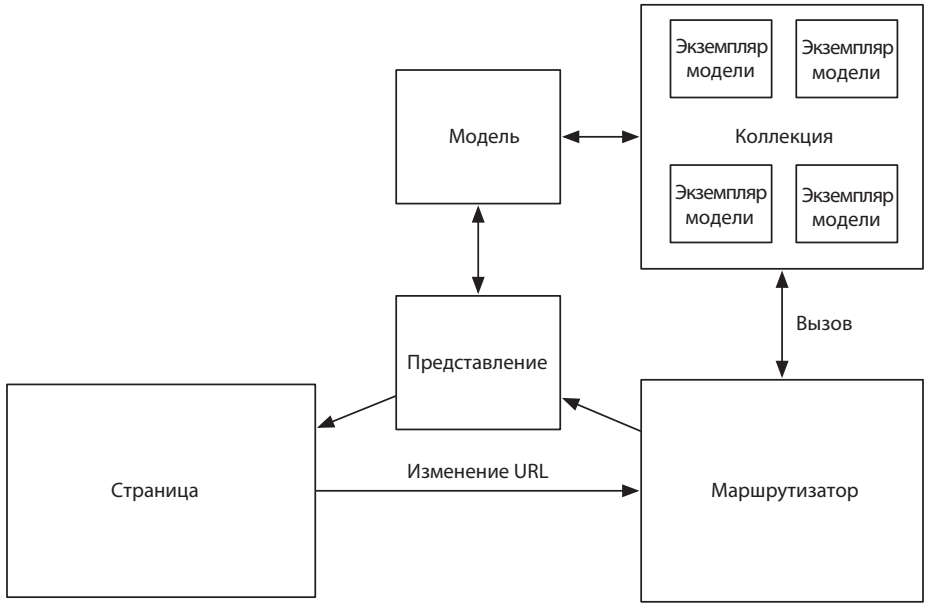

Рис. 2.10. Компоненты Backbone

Создание интерфейса

Мы хотим построить простой одностраничный интерфейс, с помощью которого можно будет легко работать со списком товаров интернет-магазина (рис. 2.11). В верхней части страницы будет располагаться форма для добавления нового товара, а под ней — список товаров. Backbone нам понадобится для того, чтобы наладить связь с серверной частью приложения, а именно реализовать поиск в списке товаров и добавить возможность изменения данных. Для этого мы будем использовать интерфейс в стиле REST.

- Запрос GET() к http://example.com/products.json находит нужный элемент в списке товаров.
- Запрос GET к /products/1.json находит JSON-представление товара с идентификатором 1.
- Запрос POST к /products.json с JSON-представлением товара в теле запроса создает новый товар.
- Запрос PUT к http://example.com/products/1.json с JSON-представлением товара в теле запроса обновляет товар с идентификатором 1.
- Запрос DELETE к /products/1.json удаляет товар с идентификатором 1.

Поскольку запросы Ajax должны выполняться в пределах одного и того же домена, мы выберем в качестве сервера разработки QEDServer; мы также воспользуемся его API для управления продуктом. Чтобы сервер разработки нашел все нужные файлы, мы поместим их в папку public, которую QEDServer создаст в рабочей области.

Для создания интерфейса нам потребуется модель, представляющая один товар, и коллекция, объединяющая в себе множество моделей товаров. Нам также понадобится маршрутизатор, который будет обрабатывать запросы на отображение списка товаров и формы добавления нового товара. Кроме того, нам нужно будет создать представления для списка товаров и формы добавления товара.

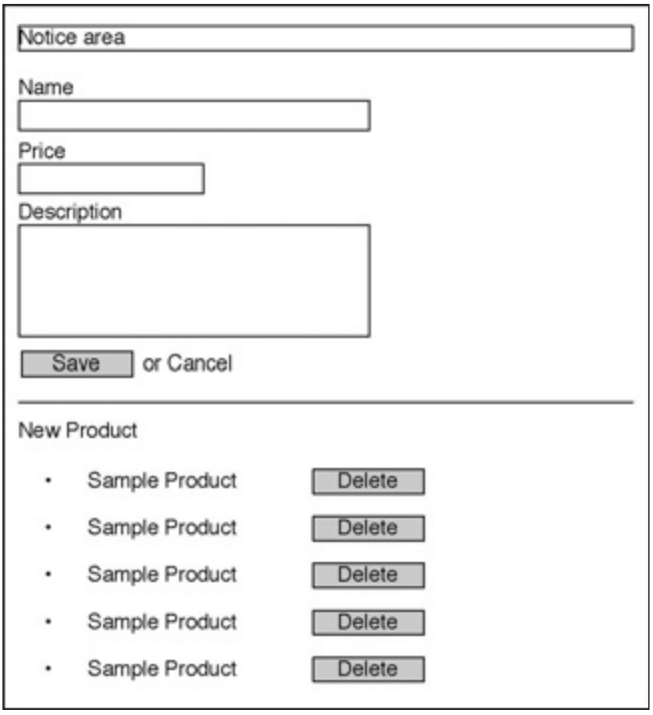

Рис. 2.11. Интерфейс для работы с товарами интернет-магазина

Начнем с создания папки lib, в которую мы поместим библиотеку Backbone и связанные с ней файлы.

```
$ mkdir javascripts
$ mkdir javascripts/lib
```

Далее нам нужно загрузить Backbone.js и его компоненты с сайта Backbone.js[1]. В этом рецепте мы используем версию Backbone 0.5.3. Нам также потребуется библиотека Underscore.js, так как Backbone использует некоторые из ее JavaScript-функций, существенно облегчающих написание кода. Кроме того, мы подключим библиотеку JSON2, благодаря которой разбор JSON поддерживается в большем числе браузеров. А так как мы уже знакомы с шаблонами Mustache, нам понадобится соответствующая библиотека: на ее основе мы и будем создавать шаблоны[2]. Загрузите эти файлы и поместите их в папку javascripts/lib.

Наконец, создадим в папке javascripts файл app.js. В нем будут храниться все наши компоненты Backbone, а также наш собственный код. Идея поместить их в отдельные файлы кажется разумной только на первый взгляд: в итоге это потребует дополнительного обращения к серверу для каждого файла.

Теперь у нас есть все что нужно, и мы можем приступить к созданию основного HTML-кода в файле index.html. В нем мы зададим элементы пользовательского интерфейса

[1] http://documentcloud.github.com/backbone/

[2] Чтобы сэкономить ваше время, мы поместили все эти файлы вместе с исходными кодами для этой книги.

и подключим все остальные файлы. Мы начнем с самых стандартных объявлений, а затем добавим несколько пустых элементов `<div>` (которые будут содержать сообщения для пользователя), форму и элемент `<ul>` для списка товаров.

backbone/public/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Product Management</title>
  </head>
  <body role="application">
    <h1>Products</h1>
    <div aria-live="polite" id="notice">
    </div>
    <div aria-live="polite" id="form">
    </div>
    <p><a href="#new">New Product</a></p>

    <ul aria-live="polite" id="list">
    </ul>
  </body>
</html>
```

Некоторые области будут обновляться без обновления всей страницы, поэтому мы добавили к ним атрибуты HTML5 ARIA. Это нужно для того, чтобы программы экранного доступа могли правильно обрабатывать такие события[1].

Под этими областями и непосредственно *перед* закрывающим тегом `<body>` мы подключим jQuery, библиотеку Backbone и то, что она будет использовать, а также наш файл app.js.

backbone/public/index.html

```
<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
</script>
<script type="text/javascript"
        src="javascripts/lib/json2.js"></script>
<script type="text/javascript"
        src="javascripts/lib/underscore-min.js"></script>
<script type="text/javascript"
        src="javascripts/lib/backbone-min.js"></script>
<script type="text/javascript"
        src="javascripts/lib/mustache.js"></script>
<script type="text/javascript"
        src="javascripts/app.js"></script>
```

А теперь давайте займемся списком товаров.

[1] http://www.w3.org/TR/html5-author/wai-aria.html

Построение списка товаров

Для составления списка мы будем обращаться за каждым товаром к серверной части, основанной на Ajax. Для этого нам понадобится модель и коллекция: модель будет представлять один товар, а коллекция — набор товаров. При создании и удалении товара мы будем работать непосредственно с моделью, а когда нам понадобится получить от сервера весь список, мы сможем обращаться к записям нашей коллекции и получить в свое распоряжение сразу несколько моделей Backbone.

Начнем с создания модели. В файле javascripts/app.js мы определим `Product` следующим образом.

backbone/public/javascripts/app.js

```
var Product = Backbone.Model.extend({

  defaults: {
    name: "",
    description: "",
    price: ""
  },
  url : function() {
    return(this.isNew() ? "/products.json" : "/products/" + this.id + ".json");
  }
});
```

Значения по умолчанию нужны в тех случаях, когда данные еще не заданы, — например, при создании нового экземпляра. После задания этих значений мы сообщаем модели, где она должна взять данные. Backbone вычисляет это с помощью метода `url()`; нам же придется расписать все полностью.

После того как мы определили модель, мы можем перейти к созданию коллекции. С ее помощью мы соберем вместе все товары, которые нужно отобразить в списке.

backbone/public/javascripts/app.js

```
var ProductsCollection = Backbone.Collection.extend({
  model: Product,
  url: '/products.json'
});
```

У коллекции, как и у модели, должен быть метод `url()`, но так как мы будем обращаться ко всем товарам сразу, достаточно будет задать постоянное значение URL — /products.json.

Так как приложение будет обращаться к коллекции в нескольких местах, нам понадобится экземпляр коллекции товаров. Этот объект мы создадим в самом начале файла javascripts/app.js.

backbone/public/javascripts/app.js

```
$(function(){
  window.products = new ProductsCollection();
```

Объект коллекции товаров мы присоединяем к объекту `window`. Благодаря этому нам будет проще обращаться к нашей коллекции из разных представлений.

Итак, мы определили модель и коллекцию и теперь можем переключить свое внимание на представление.

Шаблон для списка и представление

Представления Backbone включают в себя всю логику, относящуюся к изменениям интерфейса в ответ на различные события. Для списка товаров нам понадобится два представления. Первое будет представлять отдельные товары; оно будет использовать шаблон Mustache и обрабатывать все события, касающиеся данного товара. Второе будет перебирать все элементы коллекции товаров и для каждого из них отображать на странице первое представление. Так мы сможем точнее контролировать каждый отдельный компонент.

Для начала создадим простой шаблон Mustache, с помощью которого представления Backbone будут перебирать товары из коллекции. Этот шаблон нужно добавить в index.html перед теми тегами `<script>`, которые подключают библиотеки.

backbone/public/index.html

```
<script type="text/html" id="product_template">
  <li>
    <h3>
      {{name}} - {{price}}
      <button class="delete">Delete</button>
    </h3>
    <p>{{description}}</p>
  </li>
</script>
```

Мы добавили название, стоимость и описание товара, а также специальную кнопку для его удаления.

Далее мы создадим новое представление `ProductView`, которое будет расширением класса Backbone `View`. В нем есть несколько важных моментов.

backbone/public/javascripts/app.js

```
ProductView = Backbone.View.extend({
  template: $("#product_template"),
  initialize: function(){
    this.render();
  },
  render: function(){
  }
});
```

Сначала, используя jQuery, мы достаем с начальной страницы шаблон Mustache по его идентификатору и записываем в свойство `template`. Иначе нам пришлось бы делать это каждый раз, когда требуется отобразить товар.

Затем мы определяем функцию `initialize()`. Она будет запускаться в момент создания нового экземпляра ListView и вызывать функцию `render()` нашего представления.

У каждого представления по умолчанию есть функция `render()`. Но чтобы она на самом деле что-то делала, нам придется ее переопределить. Мы хотим, чтобы она отображала шаблон Mustache, который уже находится в переменной `template`. Однако там он хранится в виде объекта jQuery. Поэтому чтобы получить его содержимое, нам понадобится вызвать функцию `html()`.

backbone/public/javascripts/app.js

```
render: function(){
  var html = Mustache.to_html(this.template.html(), this.model.toJSON() );
  $(this.el).html(html);
  return this;
}
```

В этом методе мы обращаемся к `this.model`, так как там содержатся товары из нашего будущего списка. Когда мы будем создавать новый экземпляр представления, мы можем передать ему модель или коллекцию, и тогда к этой модели или коллекции можно будет легко обращаться в методах представления — точно так же, как это было с шаблоном Mustache. Чтобы шаблон имел доступ к данным модели, мы вызываем от этой модели метод `toJson()` и передаем результат шаблону.

Метод `render()` записывает HTML из шаблона Mustache в свойство представления `el`, а затем возвращает данный экземпляр `ProductView`. Во время вызова этого метода мы возьмем значение этого свойства и добавим его на страницу.

Чтобы это сделать, нам потребуется создать еще одно представление — `ListView`. Его структура будет аналогична структуре `ProductView`, но вместо того чтобы отображать шаблон Mustache, оно будет перебирать все элементы коллекции товаров и отображать `ProductView` для каждого из них.

backbone/public/javascripts/app.js

```
ListView = Backbone.View.extend({
  el: $("#list"),

  initialize: function() {
    this.render();
  },

  renderProduct: function(product){
    var productView = new ProductView({model: product});
    this.el.append(productView.render().el);
  },

  render: function() {
    if(this.collection.length > 0) {
       this.collection.each(this.renderProduct, this);
    } else {
      $("#notice").html("There are no products to display.");
    }
  }
});
```

Список товаров нам нужен для того, чтобы обновить содержимое области `list`. Ссылка на эту область хранится в свойстве `el`, поэтому мы можем обратиться к нему прямо из метода `render()` (то же самое мы делали с шаблоном Mustache в `ProductView`).

Чтобы максимально упростить работу с коллекциями, Backbone использует дополнительные функции из Underscore.js. Внутри метода `render()` мы с помощью метода `each()` проходим по всем элементам коллекции и вызываем метод `renderProduct`. При этом метод `each()` передает товар автоматически. В качестве второго параметра мы передаем `this`, так как мы хотим, чтобы `renderProduct()` выполнялся в рамках этого представления. Иначе метод `each()` стал бы искать `renderProduct()` в коллекции и в итоге просто не работал.

На данном этапе у нас есть модель, коллекция и пара представлений; кроме того, мы добавили шаблон, однако нам все еще нечего в нем показать. В момент загрузки страницы в браузере все это должно как-то соединиться вместе. Для этого нам потребуется маршрутизатор.

Обработка изменений URL с помощью маршрутизаторов

Во время первого запуска страницы нам нужно будет вызвать специальный код, который достанет коллекцию товаров из Ajax API. Чтобы товары можно было отобразить на странице, эту коллекцию нужно будет передать новому экземпляру `ListView`. Маршрутизаторы Backbone позволяют следить за изменениями URL и реагировать на эти изменения вызовом специальных функций.

Давайте создадим новый маршрутизатор `ProductsRouter`, который будет расширением маршрутизатора Backbone. В нем нам нужно задать *маршрут*, то есть установить соответствие между частью URL, расположенной после знака решетки, и вызываемой функцией маршрутизатора. Для тех случаев, когда в URL не будет знака решетки, мы определим пустой маршрут, который будет вызывать функцию `index()`. Этот маршрут будет запускаться при загрузке страницы index.html.

backbone/public/javascripts/app.js

```
ProductsRouter = Backbone.Router.extend({
    routes: {
      "": "index"
    },
    index: function() {
    }
});
```

Чтобы получить данные с сервера, функция `index()` берет нашу коллекцию товаров и вызывает для нее метод `fetch()`.

backbone/public/javascripts/app.js

```
index: function() {
  window.products.fetch({
    success: function(){
      new ListView({ collection: window.products });
    },
    error: function(){
```

```
      $("#notice").html("Could not load the products.");
    }
  });
}
```

Аргументами метода `fetch()` являются функции `success` и `error`. Если при попытке получить данные с сервера происходит ошибка, мы сообщаем об этом пользователю, помещая соответствующее уведомление на странице в специально отведенной для этого области `notice`. Если же нам удается получить данные с сервера, запускается функция `success()`, которая создает новый экземпляр представления. Поскольку метод `initialize()` нашего представления автоматически создает представление списка, нам остается только создать новый экземпляр маршрутизатора, тем самым запустив весь процесс.

Код для создания нового экземпляра маршрутизатора мы поместим в файле javascripts/app.js сразу после определения `window.productCollection`. После этого Backbone начнет следить за изменением URL.

backbone/public/javascripts/app.js

```
window.products = new ProductsCollection();
// START_HIGHLIGHTING
$.ajaxSetup({ cache: false });
window.router = new ProductsRouter();
Backbone.history.start();
// END_HIGHLIGHTING
```

В строке `Backbone.history.start();` мы сообщаем Backbone о том, что пора начать следить за изменением URL. Без этого маршрутизатор не будет работать и на странице ничего не будет происходить.

В следующей строке мы позаботились о том, чтобы браузеры не перехватывали ответы Ajax, которые будут приходить с сервера.

```
$.ajaxSetup({ cache: false });
```

Теперь на http://localhost:8080/index.html можно наконец-то увидеть наш список товаров.

Итак, давайте посмотрим, чего мы добились на данном этапе. У нас есть маршрутизатор, который следит за изменением URL и вызывает специальный метод, который получает от веб-службы необходимые модели, используя нашу коллекцию товаров. Далее эта коллекция передается представлению, которое создает шаблон и добавляет его в пользовательский интерфейс (см. схему, изображенную на рис. 2.12). На первый взгляд может показаться, что с помощью такой сложной комбинации действий и длинного кода мы на самом деле делаем что-то очень простое. Сейчас это действительно так, но зато потом, когда придет время усовершенствовать наше приложение, мы сэкономим массу времени. Нам удалось заложить основу добавления, обновления и удаления товаров, так что теперь не нужно будет беспокоиться о том, как это будет происходить. Далее мы подробно рассмотрим процедуру добавления товаров.

Создание нового товара

Чтобы создать новый товар, пользователь должен будет щелкнуть на ссылке Новый товар (New Product) и заполнить открывшуюся форму. После этого мы отправим на сервер данные из этой формы и обновим список товаров.

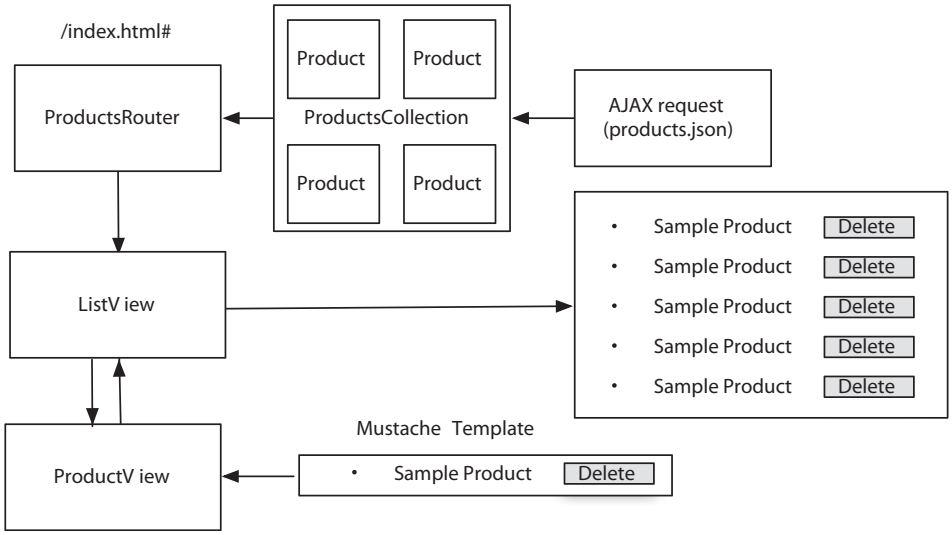

Рис. 2.12. Получение списка товаров с помощью Backbone

Для формы нам потребуется еще один шаблон Mustache, который мы добавим в index.html *сразу после* шаблона для товара, но перед тегами `<script>`, в которых мы подключаем библиотеки.

backbone/public/index.html

```
<script type="text/html" id="product_form_template">
  <form>
    <div class="row">
      <label>Name<br>
        <input id="product_name" type="text" name="name"
               value="{{name}}">
      </label>
    </div>
    <div class="row">
      <label>Description<br>
        <textarea id="product_description"
                  name="description">{{description}}</textarea>
      </label>
    </div>
    <div class="row">
      <label>Price<br>
        <input id="product_price" type="text" name="price"
               value="{{price}}"></label>
    </div>
    <button>Save</button>
  </form>
  <p><a id="cancel" href="#">Cancel</a></p>
</script>
```

Теги шаблона Mustache поместят значения из модели в поля формы. Именно поэтому, создавая модель Backbone, мы задали значения по умолчанию. Позже, при редактировании записей, мы сможем использовать этот шаблон еще раз.

Теперь нам нужно представление, которое будет отображать этот шаблон на основе модели. Поэтому в файле javascripts/app.js мы создадим объект `FormView`. В целом он будет аналогичен представлению для списка товаров, однако на этот раз в качестве значения переменной `el` мы возьмем область `form`, а функция `render()` будет брать шаблон формы и отображать результат в этой области.

backbone/public/javascripts/app.js

```
FormView = Backbone.View.extend({
  el: $("#form"),
  template: $("#product_form_template"),
  initialize: function(){
    this.render();
  },
  render: function(){
    var html = Mustache.to_html(this.template.html(), this.model.toJSON() );
    this.el.html(html);
  }
});
```

Эта форма должна появляться на странице только после того, как пользователь щелкнет на ссылке Новый товар (New Product). Поскольку нажатие на этой ссылке приводит к изменению URL — точнее, к добавлению строки #new, — мы можем поручить ответные действия маршрутизатору. Для этого нам нужно сначала изменить раздел `routes`, добавив в него новый маршрут для #new (то есть того адреса, на который указывает наша ссылка).

backbone/public/javascripts/app.js

```
routes: {
  "new": "newProduct",
  "": "index"
},
```

Затем мы определим функцию, которая будет создавать новую модель и передавать ее новому экземпляру представления формы, с тем чтобы потом представление можно было отобразить на странице. Этот метод мы поместим перед методом `index()`; поскольку объявления этих двух методов на самом деле являются свойствами объекта `this`, нужно обязательно поставить между ними запятую.

backbone/public/javascripts/app.js

```
newProduct: function() {
  new FormView( {model: new Product()});
},
```

Когда мы обновляем страницу и нажимаем на ссылке Новый товар (New Product), на экране отображается наша форма. Благодаря Backbone History мы можем пользоваться кнопкой Назад (Back), и при этом URL будет меняться. Однако пока мы не можем сохранять записи, поэтому следующим шагом будет добавление соответствующей логики.

Как представления могут реагировать на события

Только что мы использовали маршрутизатор для того, чтобы отобразить нашу форму. Однако маршрутизаторы могут реагировать только на изменение URL, а нам нужно научиться обрабатывать события щелчка мыши на кнопке Сохранить (Save) и ссылке Отмена (Cancel). Поэтому мы предлагаем сделать это в нашем представлении формы.

Сначала определим события, которые наше представление будет отслеживать. После функции `initialize()` мы добавим следующий код.

backbone/public/javascripts/app.js

```
events: {
   "click .delete": "destroy"
 },
events: {
    "click #cancel": "close",
    "submit form": "save",
},
```

Этот синтаксис немного отличается о того, который обычно используется в JavaScript для обработки событий. В ключе задается событие, которое мы хотим отслеживать и за которым следует CSS-селектор наблюдаемого элемента. Значение задает функцию представления, которую нужно будет запустить. В нашем случае мы будем отслеживать событие щелчка мыши на кнопке отмены и событие отправки формы.

backbone/public/javascripts/app.js

```
close: function(){
  this.el.unbind();
  this.el.empty();
},
```

Метод `save()` устроен несколько сложнее. Сначала мы отменяем отправку формы, а затем берем значения всех полей формы и помещаем их в новый массив. Далее мы задаем атрибуты нашей модели и вызываем ее метод `save()`.

backbone/public/javascripts/app.js

```
save: function(e){
  e.preventDefault();
  data = {
    name: $("#product_name").val(),
    description: $("#product_description").val(),
    price: $("#product_price").val()
  };
  var self = this;
  this.model.save(data, {
    success: function(model, resp) {
      $("#notice").html("Product saved.");
      window.products.add(self.model);
      window.router.navigate("#");
```

```
      self.close();
    },
    error: function(model, resp){
      $("#notice").html("Errors prevented the product from being created.");
    }
  });
},
```

Метод `save()` предполагает использование того же принципа, что и метод `fetch()`, который мы описывали при создании коллекции товаров. Это значит, что нам нужно определить его поведение в случае успеха и неудачи (при возникновении ошибки). Так как эти две функции обратного вызова имеют разную область видимости, нам придется хранить текущую область видимости во временной переменной `self`, и тогда к ней можно будет обращаться в функции `success`. В отличие от метода `each()`, который мы использовали при создании списка товаров, в Backbone функциям обратного вызова нельзя передать область видимости[1].

Если товар был успешно сохранен, мы добавляем в коллекцию новую модель и меняем URL с помощью маршрутизатора. Правда, при этом не происходит запуска соответствующей функции маршрутизатора, то есть товар не появляется в списке. К счастью, благодаря связыванию событий Backbone это можно легко исправить.

Когда мы добавляем в коллекцию новую модель, коллекция выбрасывает событие add, которое также можно отслеживать. Вы помните метод `renderProduct()` из представления списка? Мы можем сделать так, чтобы представление запускало этот метод всякий раз после добавления в коллекцию новой модели. Для этого нужно всего лишь добавить в метод `initialize()` объекта `ListView` следующую строку.

backbone/public/javascripts/app.js

```
this.collection.bind("add", this.renderProduct, this);
```

Метод `bind()` позволяет обрабатывать специфические события. Для этого ему нужно передать событие, функцию и область видимости. Здесь в качестве третьего параметра мы передаем `this`, так как хотим, чтобы областью видимости было *представление*, а не коллекция. Мы уже сталкивались с аналогичной ситуацией при описании метода `render()` в представлении списка: тогда это относилось к `collection.each`.

Так как при добавлении новой записи мы используем функцию `renderProduct()`, запись будет добавляться в конец списка. Если вы хотите, чтобы она появлялась в начале списка, вы можете создать новую функцию `addProduct()` и использовать в ней метод jQuery `prepend()`.

Теперь на нашей страничке можно создавать новые товары, после чего список товаров автоматически обновляется. Далее мы займемся удалением товаров и увидим, что все наши труды не пропали даром.

Удаление товара

Чтобы реализовать удаление товара, мы должны вспомнить все, что мы делали при создании `FormView`, и добавить в `ProductView` функцию `destroy()`, которая будет запускаться при нажатии кнопки Удалить (Delete).

[1] По крайней мере, на момент написания этой книги.

Для начала мы определим еще одно событие, которое нужно будет отслеживать, а именно событие щелчка мыши на кнопке с классом `delete`.

backbone/public/javascripts/app.js

```
events: {
  "click .delete": "destroy"
 },
events: {
    "click #cancel": "close",
    "submit form": "save",
},
```

Далее мы определим метод `destroy()`, который вызывает наше событие. Внутри этого метода мы вызовем метод `destroy()` от нашей модели, связанной с этим представлением. В этом внутреннем методе будет использоваться уже знакомая нам стратегия с функциями обратного вызова `success` и `error`. Чтобы обойти проблему с областями видимости, вспомним прием с переменной `self`, который мы удачно использовали при сохранении записей в представлении формы.

backbone/public/javascripts/app.js

```
destroy: function(){
  var self = this;
  this.model.destroy({
  success: function(){
    self.remove();
  },
  error: function(){
      $("#notice").html("There was a problem deleting the product.");
    }
  });
},
```

Если сервер успешно удалил модель, запускается функция `success`, которая, в свою очередь, вызывает для данного представления метод `remove()`, и запись исчезает с экрана. Если что-то пошло не так, на экране отображается сообщение об ошибке.

Дополнительные возможности

Что ж, на этом мы остановимся. Для начала получилось очень неплохо. Но есть еще несколько интересных возможностей, которые мы оставили вам для самостоятельного изучения.

Во-первых, код, обновляющий `notice`, мы используем в нескольких местах.

```
$("#notice").html("Product saved.");
```

Вместо этого можно создать функцию-обертку, чтобы отделить его от разметки, или же добавить еще одно представление Backbone и еще один шаблон Mustache.

При сохранении записей мы достаем значения из формы, ориентируясь на селекторы jQuery. Вместо этого можно использовать события onchange на полях формы и таким образом добавлять данные сразу в модель.

Следующий шаг после добавления и удаления записей — это их редактирование. Отображение формы по-прежнему останется делом маршрутизатора, а в качестве представления формы можно взять уже готовое представление, которое мы использовали при создании товара.

Backbone дает нам превосходную систему для работы с серверными данными, однако это еще далеко не все. Совсем не обязательно использовать Backbone для работы с Ajax-сервером: точно так же можно передавать данные клиентскому механизму хранения данных HTML5.

Для более прочной интеграции с серверными приложениями Backbone поддерживает метод `pushState()` из HTML5 History. Это значит, что мы можем использовать не только URL со знаком решетки, но и обычные адреса. Благодаря этому мы можем создавать изящные решения, которые будут нормально работать при отключенном JavaScript, а в противном случае будут использовать Backbone.

Благодаря несметному количеству возможностей и превосходной поддержке Ajax-серверов Backbone является гибким фреймворком, удобным в тех ситуациях, когда нам необходим хорошо структурированный код.

Смотрите также

- Рецепт 10. HTML с помощью Mustache
- Рецепт 13. Модные клиентские интерфейсы с помощью Knockout.js

3

Данные

Веб-разработчикам приходится обрабатывать данные самыми разными способами. Иногда требуется встроить виджет, предоставляемый другой службой, а иногда — получить данные непосредственно от пользователя. В рецептах этой главы рассматриваются вопросы обработки, обслуживания и представления данных.

Рецепт 15. Встроенная карта Google

Задача

Если пользователю нужно будет добраться в незнакомое место, он, скорее всего, захочет посмотреть этот адрес на карте. При этом большое значение имеют быстрота и доступность. Найти нужное место можно и просто по адресу или словесному описанию, но все же самый простой способ — посмотреть карту, запомнить название улицы, взять ключи и сесть в машину. Когда мы добавляем на наш сайт карту, мы даем пользователю наиболее полное представление о том, где находится наша организация и как до нее добраться.

Ингредиенты

- Google Maps API

Решение

С помощью программного интерфейса Google Maps мы можем использовать все возможности карт Google прямо на нашем сайте. В нашем распоряжении оказывается два типа карт: статические и интерактивные. Статическая карта — это самая обычная картинка, тогда как на интерактивной карте существует возможность прокрутки и изменения масштаба. Программный интерфейс Google Maps поддерживает все те языки программирования, которые умеют отправлять запросы на серверы Google. В документации можно найти много примеров на JavaScript[1], что очень удобно, так как мы будем использовать именно этот язык.

Используя Google Maps API на нашем сайте, мы можем реализовать весь список возможностей, доступных в полной версии приложения.

Помимо отображения самих карт, Google Maps API позволяет добавлять в них специальные элементы. Например, мы можем расставить на карте маркеры и связать их с событиями мыши; можно также создавать всплывающие окна, которые будут сообщать нам информацию об объектах прямо на карте; можно добавить виды улиц, определять местоположение пользователя, создавать маршруты и их описания, рисовать на карте свои собственные модели. Пока пределом возможностей можно считать только небо — по крайней мере, до тех пор пока Google не запустит свою космическую программу и тем самым потеснит NASA[2].

Мы работаем над созданием карты на веб-странице одного из местных университетов. Приемная комиссия хочет показать гостям и абитуриентам университета, где они могут поесть и припарковать машину. Поэтому мы обратимся к Google Maps API и создадим с его помощью интерактивную карту, добавив в нее маркеры и дополнительную информацию.

Начнем с создания базовой HTML-разметки. Следуя рекомендации Google, мы объявим тип документа HTML5; однако следует помнить, что использование `<DOCTYPE html>` не является строго обязательным.

---

[1] https://developers.google.com/maps/documentation/javascript/reference?hl=ru-RU

[2] http://www.google.com/space/

googlemaps/map_example.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Freshman Landing Page</title>
    <style>
    </style>
    <script type="text/javascript">
    </script>
  </head>
  <body>
  </body>
</html>
```

Далее добавим в наш документ Google Maps JavaScript API. При этом нам нужно указать, собираемся ли мы определять местоположение пользователя с помощью сенсора. Поскольку в нашу задачу это не входит, мы сделаем это значение равным false.

googlemaps/map_example.html

```
<script type="text/javascript"
  src="http://maps.google.com/maps/api/js?sensor=false">
</script>
```

Согласно требованиям этого API, мы должны поместить карту в контейнер `<div>`.

googlemaps/map_example.html

```
<div id="map_canvas"></div>
```

Поскольку карта займет весь контейнер, мы зададим его размеры с помощью CSS. Для этого в новый раздел `<style>` внутри области `<head>` мы добавим такой код.

googlemaps/map_example.html

```
#map_canvas {
  width: 600px;
  height: 400px;
}
```

Теперь этот контейнер готов к тому, чтобы вместить карту размером 600×400 пикселов. Далее мы займемся данными.

Как загрузить карту с помощью JavaScript

В самом низу области `<head>` мы добавим блок `<script>`, в который и поместим код для инициализации карты. Мы создадим специальную функцию `loadMap()`, которая будет загружать карту по широте и долготе; это будет происходить во время загрузки окна браузера. Если в вашем проекте используется какой-либо фреймворк (например, jQuery), загрузку карты можно выполнить внутри вызова DOM, но мы здесь обойдемся самым обычным JavaScript.

googlemaps/map_example.html

```
window.onload = loadMap;
```

Далее напишем функцию `loadMap()`. Так как мы уже отказались от сенсора, широта и долгота будут константами. Эти координаты задают центр карты. Получить их можно разными способами. Например, можно открыть карту Google, найти на ней место, которое вы хотите сделать центром, щелкнуть на соответствующей метке правой кнопкой мыши и выбрать Что тут находится? (What's here?). После этого в поле поиска появятся координаты этого объекта. Кроме того, можно воспользоваться Google Maps Lat/Long Popup[1]. Это приложение позволяет получать нужные значения, просто щелкнув мышью на нужной точке.

googlemaps/map_example.html

```
function loadMap() {
  var latLong = new google.maps.LatLng(44.798609, -91.504912);

  var mapOptions = {
    zoom: 15,
    mapTypeId: google.maps.MapTypeId.ROADMAP,
    center: latLong
  };

  var map = new google.maps.Map(document.getElementById("map_canvas"),
            mapOptions);
}
```

Внутри этой функции мы создаем объект, в котором будут храниться некоторые параметры карты. При этом можно задать тип карты, масштаб и кое-что еще. Чтобы правильно определить масштаб, вам придется немного поэкспериментировать. Чем больше значение, тем подробнее будет карта. Если вы хотите показать улицы, можно выбрать значение 15.

С помощью `mapTypeId` можно изменить способ появления карты. Однако здесь нужно обратить внимание на то, что при изменении типа карты устанавливаются новые значения масштаба и пределов его изменения. Список типов карт можно найти в документации к Google Maps API[2].

Наконец, мы создаем саму карту. Конструктору `Map` мы должны передать элемент DOM, в котором будет храниться карта, и объект с параметрами. Теперь, если открыть эту страницу в браузере, можно увидеть на ней карту с центром в заданной точке (рис. 3.1).

Добавление маркеров

Чтобы показать будущим первокурсникам, где можно перекусить или провести свободное время, мы добавим на карту несколько маркеров. В Google Maps маркеры представляют собой один из возможных слоев, а слои способны реагировать на события щелчка мыши. Благодаря этому мы сможем добавить информационные окна, которые будут открываться при щелчке мыши на маркере.

[1] http://www.gorissen.info/Pierre/maps/googleMapLocationv3.php

[2] https://developers.google.com/maps/documentation/javascript/reference?hl=ru-RU#MapTypeId

Рис. 3.1. Первоначальный вариант карты

Поскольку сама карта у нас уже есть, добавить маркер будет очень просто: нужно запустить конструктор и передать ему некоторые параметры.

googlemaps/map_example.html

```
mogiesLatLong = new google.maps.LatLng(44.802293, -91.509376);
var marker = new google.maps.Marker({
  position: mogiesLatLong,
  map: map,
  title: "Mogies Pub"
});
```

Чтобы создать маркер, нам потребуются его координаты, карта, к которой он будет относиться, и название, которое будет появляться при наведении на него курсора мыши.

Далее нам нужно создать информационное окно, которое будет появляться при щелчке мыши на маркере. Для этого нужно вызвать конструктор.

googlemaps/map_example.html

```
var mogiesDescription = "<h4>Mogies Pub</h4>" + "<p>Excellent local
  restaurant with top of the line burgers and sandwiches.</p>";
var infoPopup = new google.maps.InfoWindow({
  content: mogiesDescription
});
```

Наконец, добавим к маркеру обработчик событий. С помощью объекта `event` Google Maps мы можем добавить слушатель событий, который и откроет информационное окно.

googlemaps/map_example.html

```
google.maps.event.addListener(marker, "click", function() {
  infoPopup.open(map,marker);
});
```

Когда пользователь щелкнет на этом маркере, на экране появится окно с информацией об этом месте (рис. 3.2).

Рис. 3.2. Что произойдет, если пользователь щелкнет на маркере кнопкой мыши

В это окно можно добавить сколько угодно HTML-кода, то есть, по сути, сколько угодно информации. Действуя аналогичным образом, мы можем отметить и другие интересующие нас места и, таким образом, завершить создание карты.

Дополнительные возможности

Это лишь малая часть того, что можно сделать помощью Google Maps API. Помимо маркеров существуют и другие уровни взаимодействия, благодаря которым пользователь может легко и быстро получать необходимую информацию. Так же можно создавать маршруты и их описания, определять местоположение пользователя и даже показывать изображения улиц. Обо всех этих возможностях рассказывается в документации к Google Maps API[1]; там же представлены работающие примеры, которые можно брать за основу при создании ваших собственных веб-страниц.

Помимо Google Maps у Google существует много других программных интерфейсов. Полный список таких интерфейсов можно найти на странице Google APIs and Developer Products[2].

Смотрите также

- Рецепт 17. Простая контактная форма
- Рецепт 18. Межсайтовый доступ с помощью JSONP
- Рецепт 19. Виджет, встраиваемый на внешние сайты

[1] http://code.google.com/apis/maps/documentation/javascript/reference.html
[2] http://code.google.com/more/table

Рецепт 16. Диаграммы и графики с помощью Highcharts

Задача

Как говорится, лучше один раз увидеть, чем сто раз услышать. Конечно же, это выражение применимо и к диаграммам. Представление данных с помощью графиков и диаграмм выигрывает не только с точки зрения содержания, но зачастую и с эстетической точки зрения.

Существует множество способов создавать таблицы — начиная с генерирования изображений на сервере и заканчивая системами, основанными на Adobe Flash. Мы хотим найти простой и эффективный способ создания диаграмм и графиков, который не использовал бы Flash, так как он не работает на устройствах iOS; при этом мы хотим иметь возможность генерировать изображения на сервере.

Ингредиенты

- jQuery
- Highcharts[1]
- QEDServer

Решение

JavaScript-библиотека Highcharts позволяет легко создавать понятные интерактивные диаграммы и графики. Она подходит для большинства платформ и не требует особой конфигурации сервера, так как работает на машине клиента. Встроенный интерфейс Highlights обладает массой интерактивных возможностей и настроек; благодаря этому мы можем представлять наши данные самым разным образом. В этом рецепте мы хотим сначала создать и настроить простую диаграмму, а затем усовершенствовать ее, подключив удаленные данные.

Группа сбыта разработала партнерскую программу для нашего сайта продаж. Нас попросили разработать интерфейс для партнеров, и мы решили, что лучше отобразить их данные с помощью диаграмм и графиков. Для этого мы будем использовать Highcharts. Но сначала мы решили разобраться в том, как создать на странице самую простую диаграмму.

Простая диаграмма

Чтобы познакомиться с возможностями Highcharts, мы создадим простую секторную диаграмму. Для начала нам нужна HTML-основа, в которой мы подключим все необходимые JavaScript-файлы: highcharts.js, который можно получить на сайте Highcharts, и библиотеку jQuery, которую использует Highcharts. Обычно мы используем версию jQuery 1.7, но здесь, следуя требованиям Highcharts, мы подключим версию 1.6.2.

[1] http://www.highcharts.com/

highcharts/example_chart.html

```
<script type="text/javascript"
  src="/jquery.js">
</script>
<script type="text/javascript" src="/highcharts.js"></script>
```

Теперь можно перейти к созданию диаграммы. Для этого Highcharts потребуется отдельный элемент `<div>`, который мы поместим в элемент `<body>`. Чтобы иметь доступ к этому элементу из JavaScript-кода, добавим к нему `id`.

highcharts/example_chart.html

```
<body>
  <div id="pie-chart"></div>
</body>
```

Волшебство произойдет в тот момент, когда мы создадим новый экземпляр класса `Highcharts.Chart` и передадим ему некоторые параметры. В Highcharts существует тонкая система настроек, однако если пользоваться ими всеми, код быстро станет очень громоздким. Для простоты мы создадим переменную `chartOptions` и с ее помощью зададим несколько параметров, которые будут понятны Highcharts.

highcharts/example_chart.html

```
$(function() {
  var chartOptions = {};

  chartOptions.chart = {
    renderTo: "pie-chart"
  };
  chartOptions.title = {text: "A sample pie chart"};
  chartOptions.series = [{
    type: "pie",
    name: "Sample chart",
    data: [
      ["Section 1", 30],
      ["Section 2", 50],
      ["Section 3", 20]
    ]
  }];
  var chart = new Highcharts.Chart(chartOptions);
});
```

Первое значение — это одно из свойств диаграммы, которое содержит информацию о ней; именно сюда мы передаем идентификатор элемента `<div>`, который перед этим создали. Далее мы задаем заголовок диаграммы, выбрав в качестве значения произвольный текст. Наконец, свойство `series` представляет собой массив объектов, соответствующих диаграммам. Если задать несколько таких объектов, они будут отображаться поверх друг друга. Каждый объект имеет тип, название и набор данных, причем формат данных зависит от типа диаграммы. Для секторной диаграммы это двумерный массив, причем внутренние массивы представляют собой пары X и Y.

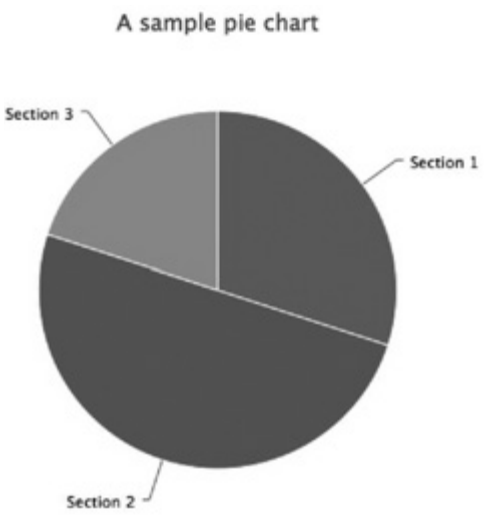

Рис. 3.3. Простая секторная диаграмма

Таким образом, для простой диаграммы (рис. 3.3) нам потребовалось всего несколько строк кода. Теперь давайте посмотрим, какие еще параметры есть в Highcharts.

Внешнее оформление диаграммы

С помощью Highcharts можно создавать секторные, линейные и точечные диаграммы, а также двумерные гистограммы; возможность корректировки этих типов позволяет создать бесчисленное множество интереснейших графиков.

Возьмем переменную `chartOptions` из предыдущего примера. Для нее можно задать свойство `plotOptions` — объект, в котором будут содержаться настройки способа прорисовки диаграммы. Давайте сделаем это для нашей секторной диаграммы.

Параметры, общие для всех диаграмм, можно задать внутри свойства `series` объекта `chartOptions`. Но можно ограничиться только одним типом и задать эти параметры только для него. В нашей секторной таблице мы хотим изменить внешний вид подписей секций диаграммы.

highcharts/example_chart.html

```
var pieChartOptions = {
  dataLabels: {
    style: {
      fontSize: 20
    },
    connectorWidth: 3,
    formatter: function() {
      var label = this.point.name + " : " + this.percentage + "%";
      return label;
    }
  }
};
```

продолжение ⇩

```
chartOptions.plotOptions = {
  pie: pieChartOptions
};
```

Чтобы подписи было лучше видно, мы изменили размер шрифта; поэтому нам пришлось увеличить и толщину линии, соединяющей подпись с соответствующим сектором. Далее мы добавили функцию, которая возвращает новый вариант подписи. По умолчанию подпись содержит только номер секции, но мы добавили к ней еще и процентное значение. В итоге мы получили нечто похожее на диаграмму, изображенную на рис. 3.4.

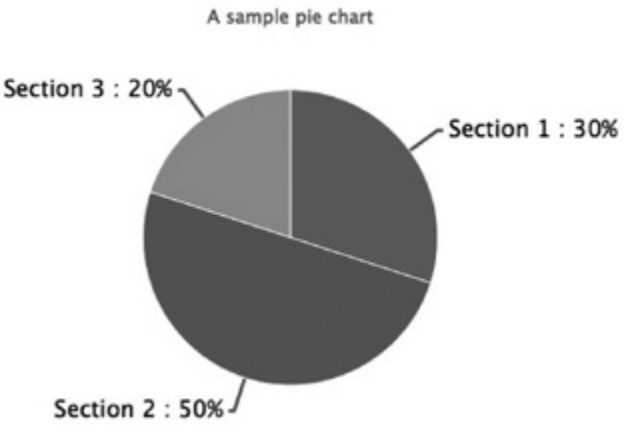

Рис. 3.4. Готовая секторная диаграмма

У свойства `plotOptions` есть масса параметров; их полный список можно найти в документации Highcharts в разделе, посвященном `plotOptions`[1].

Итак, мы научились создавать простые диаграммы и изменять их настройки. Теперь мы можем заняться данными партнерской программы.

Данные партнерской программы

Наша партнерская программа работает со сравнительно большим количеством данных, и в большинстве случаев их удобнее всего представить в виде графиков самых разных типов. Чтобы отобразить информацию о клиентах — их имена, местоположение, возраст, — нам потребуется другой тип графиков. Эти сведения позволят составить общую характеристику клиентов и на ее основе разработать стратегию сбыта продукции. Наша задача — представить данные в виде такого графика, который позволил бы маркетологам быстро проанализировать ситуацию на рынке, не обращаясь к исходным данным.

Итак, мы хотим, чтобы график мог наиболее наглядно охарактеризовать аудиторию наших клиентов. Для этого мы решили выбрать гистограмму, на которой можно будет легко увидеть среднее и наиболее частотное значение. В итоге должно получиться нечто похожее на график, изображенный на рис. 3.5.

Для начала нам понадобится новый HTML-документ с подключенными jQuery и Highcharts. Поскольку мы собираемся работать с данными в формате JSON и Ajax-запросами, этот HTML-файл нужно поместить в папку public, которую создаст QEDServer.

[1] http://www.highcharts.com/ref/#plotOptions

highcharts/affiliates.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Affiliate Customer Data</title>
  <script type="text/javascript"
    src="/jquery.js">
  </script>
  <script type="text/javascript" src="/highcharts.js"></script>
</head>
<body>
  <div id="customer-data"></div>
</body>

</html>
```

Рис. 3.5. Гистограмма с информацией о клиентах

Сюда же мы добавим специальный элемент `<script>`, в котором создадим новый экземпляр класса Highcharts.Chart. Мы хотим задать некоторые параметры, в частности название графика и тот элемент страницы, в который он будет добавлен.

highcharts/affiliates.html

```
var options = {
  chart: {
    renderTo: "customer-data"
  },
  title: {
    text: "Customer Data"
  },
  credits: {
    enabled: false
  }
};
```

Теперь у нас есть готовый HTML-документ, и мы можем заняться нашими данными.

Как отобразить информацию о клиентах

В обычной ситуации сведения о клиентах приходят с сервера, однако здесь мы будем использовать данные, специально созданные для этого примера. Их можно найти в исходных кодах, размещенных на сайте этой книги.

Как вы наверняка помните, по соображениям безопасности браузеры запрещают загружать обычные данные в формате JSON с произвольного удаленного сервера; для этого необходимо, чтобы index.html и файл с данными хранились на одном и том же сервере. Поэтому наши данные следует поместить в папку sample_data, которая в свою очередь должна находиться в папке public, созданной QEDServer. В таком случае QEDServer сможет использовать данные с http://localhost:8080/sample_data/customer_data.json, а вебстраница сможет правильно их обрабатывать.

Чтобы показать на гистограмме возраст клиентов, нам нужно сопоставить каждое значение возраста с тем, сколько раз встречается это значение. Пока у нас есть только список возрастов, поэтому для получения частот нам потребуется JavaScript. Мы создадим запрос для получения информации о клиентах и поместим все эти вычисления в функцию `success`, которая будет запускаться при успешном выполнении запроса Ajax.

highcharts/affiliates.html

```
$.getJSON('/sample_data/customer_data.json', function(data) {

  var ages = [];

  $.each(data.customers, function(i, customer) {
    if (typeof ages[customer.age] === "undefined") {
      ages[customer.age] = 1;
    } else {
      ages[customer.age] += 1;
    }
  });

  var age_data = [];

  $.each(ages, function(i, e) {
    if (typeof e !== "undefined") {
      age_data.push([i, e]);
    }
  });
});
```

В процессе вычисления мы использовали временный массив `ages`. В нем каждому индексу соответствует число значений возраста, равных этому индексу. Далее мы просматриваем этот массив и выбираем из него только те значения возрастов, которые встретились хотя бы один раз, и, следуя требованиям Highcharts, помещаем их в двумерный массив. После перевода данных в правильный формат можно создать гистограмму.

highcharts/affiliates.html

```
options.series = [{
  type: "column",
  name: "Customer Ages",
  data: age_data
}];

var chart = new Highcharts.Chart(options);
```

Теперь можно наконец-то посмотреть на график и понять, к каким возрастным группам относятся наши клиенты.

Дополнительные возможности

Highcharts обладает широким кругом возможностей. И хотя в этом разделе мы прошли путь от простых диаграмм к более сложным, мы коснулись лишь малой части того, на что способна эта JavaScript-библиотека. Чтобы более полно представить себе все ее возможности, откройте справочный раздел сайта Highcharts[1]. Также имеет смысл познакомиться с документацией Highcharts и отметить для себя те возможности, которые могут вам понадобиться в будущем. Кроме того, в документации вы найдете ссылку на JSFiddle.net[2], где приведены JavaScript-примеры использования большинства этих возможностей.

Смотрите также

- Рецепт 18. Межсайтовый доступ с помощью JSONP
- Рецепт 15. Встроенная карта Google
- Рецепт 9. Общение с веб-страницей с помощью горячих клавиш
- Рецепт 23. Drag and Drop на мобильных устройствах

[1] http://www.highcharts.com/ref/
[2] http://jsfiddle.net/

Рецепт 17. Простая контактная форма

Задача

На любом, пусть даже статическом, сайте пользователь должен иметь возможность связаться с его владельцем. Однако просто разместить на странице электронный адрес недостаточно: нужно постараться заинтересовать пользователя, предложив ему простой и понятный способ контакта. С другой стороны, владельцу будет гораздо легче отсортировать сообщения, отправленные прямо с сайта.

Ингредиенты

- ❑ PHP-сервер

Решение

Отправка электронного письма через контактную форму требует гораздо меньшего числа операций, что, безусловно, увеличивает вероятность контакта с пользователем. С помощью HTML-формы мы сможем обработать данные, введенные пользователем, создать специальные скрипты для отправки писем и сообщить пользователю об успешной отправке или возникших ошибках.

В текущей версии нашего сайта пользователь не может связаться с владельцем, что, по всей вероятности, лишает нас потенциальных клиентов. Поэтому менеджер компании хочет, чтобы мы создали простую форму для отправки электронных писем.

Из множества возможных серверных языков в нашей ситуации лучше всего использовать PHP: благодаря его простому синтаксису мы сможем без труда написать скрипт для обработки данных контактной формы. Кроме того, PHP изначально доступен на большинстве серверов; а противном случае его можно легко установить. Это удобный инструмент для написания простых серверных функций, в которых разработчикам хочется обойтись без использования более сложных фреймворков.

При создании контактной формы мы будем одновременно работать над двумя компонентами: HTML и PHP. С помощью HTML мы создадим саму форму, которая будет предлагать пользователю ввести данные, а затем с помощью PHP мы обработаем эти данные и отправим электронное письмо. Мы также хотим добавить некоторые элементы интерфейса — например, сообщения об ошибках. Для тестирования мы обратимся к виртуальной машине. Если вы еще не создали свой PHP-сервер разработки, сделайте это сейчас, следуя указаниям из рецепта 37 («Как установить виртуальную машину»).

Создание HTML

Для начала займемся HTML-разметкой. Форма будет предлагать пользователю выполнить четыре действия: ввести имя, электронный адрес, тему и текст сообщения. Мы будем считать электронный адрес обязательным полем, так как иначе обратная связь с пользователем будет невозможна. Для темы сообщения мы зададим значение по умолчанию. Все эти типы данных мы опишем в файле contact.php и, таким образом, создадим нашу форму.

contact/contact.php

```
<form id="contact-form" action="contact.php" method="post">

  <label for="name">Name</label>
  <input class="full-width" type="text" name="name">

  <label for="email">Your Email</label>
  <input class="full-width" type="text" name="email">

  <label for="subject">Subject</label>
  <input class="full-width" type="text" name="subject"
         value="Web Consulting Inquiry">

  <label for="body">Body</label>
  <textarea class="full-width" name="body"></textarea>

  <input type="submit" name="send" value="Send">
</form>
```

Обратите внимание на то, что атрибут `action` отсылает к файлу, в котором находится сама форма. Это дает нам возможность поместить на одной странице и контактную форму, и скрипты, отвечающие за ее отправку. Мы добавили все необходимые поля и кнопку отправки. Теперь у нас есть форма, но пока она выглядит как беспорядочное нагромождение кнопок и строк ввода. Чтобы расставить по местам все наши метки и поля ввода, добавим немного оформления.

contact/contact.php

```
body {
  font-size: 12px;
  font-family: Verdana;
}

#contact-form {
  width: 320px;
}

#contact-form label {
  display: block;
  margin: 10px 0px;
}

#contact-form input, #contact-form textarea {
  padding: 4px;
}

#contact-form .full-width {
  width: 100%;
}
```

продолжение ⇩

```
#contact-form textarea {
  height: 100px;
}
```

Мы поменяли шрифты, добавили нужное количество отступов и полей, а также передвинули некоторые элементы. Теперь наша форма выглядит более понятной и удобной (рис. 3.6), и мы можем заняться ее функциональными возможностями и перейти к написанию серверной части кода.

Рис. 3.6. Форма с элементами оформления

Отправка электронного сообщения

Мы хотим, чтобы во время обработки страницы PHP перехватывал запросы POST и отправлял электронные сообщения. Поскольку наша страница отправляет сообщения сама себе, нам необходимо всего лишь добавить PHP-код в ее верхней части. Когда пользователь нажмет на кнопку отправки, мы должны будем взять данные из переменной `$_POST`, проверить их, а затем отправить, используя PHP-функцию `mail()`. Код, выполняющий предварительную обработку, мы поместим в PHP-блок перед тегом `<html>`.

contact/contact.php

```
<?php
if (isset($_POST["send"])) {
}
?>
```

Обработку нужно выполнять только в том случае, если пользователь нажал кнопку Отправить (Send). Так как в HTML-коде у этой кнопки есть атрибут `name`, мы можем проверить, есть ли это значение с массиве `$_POST`. После этого мы можем обратиться к данным, которые ввел пользователь. Здесь мы будем также использовать массив `$_POST` и для удобства создадим специальные переменные.

contact/contact.php

```
$name = $_POST["name"];
$email = $_POST["email"];
$subject = $_POST["subject"];
$body = $_POST["body"];
```

Теперь наши данные хранятся в специальных переменных. Далее мы хотим проверить адрес электронной почты пользователя. Для этого нам потребуется регулярное выражение. Если адрес окажется неправильным, нам нужно сообщить об этом пользователю.

contact/contact.php

```
$errors = array();

$email_matcher = "/^[_a-z0-9-]+(\.[_a-z0-9-]+)*" .
"@" .
"[a-z0-9-]+" .
"(\.[a-z0-9-]+)*(\.[a-z]{2,3})$/";

if (preg_match($email_matcher, $email) == 0) {
  array_push($errors, "You did not enter a valid email address");
}
```

Все ошибки мы помещаем в специальный массив, чтобы потом вывести их на экран одновременно. Этот массив мы определили прямо здесь, так что он будет доступен в любом место страницы.

Пора отправлять сообщение. Давайте вызовем PHP-функцию `mail()`. Здесь нам нужно будет указать несколько аргументов: адрес, на который отправляется письмо, тему, текст сообщения и любое количество заголовков. Для всего этого нам потребуются дополнительные переменные. Давайте создадим их и выполним вызов `mail()`.

contact/contact.php

```
if (count($errors) == 0) {
  $to = "joe@awesomeco.com"; // ваш адрес
  $subject = "[Generated from awesomeco.com] " . $subject;

  $from = $name . " <" . $email . ">";
  $headers = "From: " . $from;

  if (!mail($to, $subject, $body, $headers)) {
    array_push($errors, "Mail failed to send.");
  }
}
```

Необходимо убедиться в том, что во время отправки сообщения не произошло ошибок. В качестве флага можно использовать значение функции `mail()`: в случае успешной отправки она возвращает `true`. Если что-то пойдет не так, мы добавим в массив `$errors` еще одну строку и таким образом сообщим об ошибке пользователю. Теперь можно проверить, как это все работает.

Тестирование контактной формы

Чтобы протестировать форму, нам потребуется папка с поддержкой PHP, размещенная на сервере разработки. В данном случае мы будем использовать виртуальную машину, которая работает в нашей сети на http://192.168.1.100/. Если у вас еще нет своей виртуальной машины, загляните в рецепт 37 («Как установить виртуальную машину»): в нем рассказывается о том, как настроить сервер для тестирования.

Давайте отправим на сервер разработки тот файл, над которым мы работали. Для этого можно использовать команду `scp` или одну из SFTP-программ — например, FileZilla для Windows.

```
$ scp contact.php webdev@192.168.1.100:/var/www/
```

Открыв http://192.168.1.100/contact.php, мы можем ввести данные и нажать на кнопку отправки. После этого проверьте, пришло ли сообщение на ваш адрес. Сообщение должно выглядеть примерно так, как показано на рис. 3.7.

Рис. 3.7. Отправленное электронное сообщение

Сообщения об ошибках

Как вы помните, в PHP-коде мы проверяли правильность введенного адреса. Однако в случае ошибки наша страница никак на нее не отреагирует. Это нужно исправить, так что вернемся к нашему HTML-коду.

contact/contact.php

```
<?php if (count($errors) > 0) : ?>
  <h3>There were errors that prevented the email from sending</h3>

  <ul class="errors">
    <?php foreach($errors as $error) : ?>
      <li><?php echo $error; ?></li>
    <?php endforeach; ?>
  </ul>
<?php endif; ?>
```

В самом начале кода нашей формы мы проверим, есть ли что-нибудь в массиве `$errors`. Если да, мы пройдем по всем его элементам и выведем сообщения на экран. Здесь мы обращаемся к альтернативному синтаксису, позволяющему использовать блоки `if` и `for` и, таким образом, избежать нагромождения одинарных и двойных кавычек и операторов `echo`. Далее можно заняться оформлением списка ошибок. Чтобы привлечь внимание пользователя, мы сделаем текст этих сообщений красным.

contact/contact.php

```
.errors h3, .errors li {
  color: #FF0000;
}

.errors li {
  margin: 5px 0px;
}
```

Вывод сообщений об ошибках — важный шаг навстречу пользователю. Однако нам нужно исправить один неприятный недостаток этой системы: дело в том, что при возникновении ошибки из формы исчезает вся ранее введенная информация. К счастью, это легко исправить, так как эти данные хранятся в специальных переменных. Для этого нам нужно добавить в каждое поле `<input>` свойство `value`, а в область `<textarea>` — соответствующий текст. Теперь поля формы будут выглядеть так.

```
<label for="name">Name</label>
<input class="full-width" type="text" name="name"
       value="<?php echo $name; ?>" />

<label for="email">Your Email</label>
<input class="full-width" type="text" name="email"
       value="<?php echo $email; ?>" />

<label for="subject">Subject</label>
<input class="full-width" type="text" name="subject"
       value="<?php echo isset($subject) ?
         $subject : 'Web Consulting Inquiry'; ?>" />

<label for="body">Body</label>
<textarea class="full-width" name="body"><?php echo $body; ?></textarea>
```

На этом мы завершаем работу над системой оповещения об ошибках. Теперь, если пользователь вводит неправильные данные, он получает уведомление об ошибке, а введенная им информация сохраняется в полях формы. На рис. 3.8 можно увидеть, что произойдет, если пользователь введет неправильный адрес.

Рис. 3.8. Сообщение об ошибке

Когда на нашем сайте появится такая форма, пользователи будут чаще отправлять нам электронные сообщения, а это не может не повлиять на успешность нашей компании.

Дополнительные возможности

Здесь мы рассмотрели лишь один пример того, что можно сделать с помощью формы на основе PHP. Например, на сайте компании, занимающейся веб-консалтингом, можно создать такую форму, которая будет помогать пользователям узнавать стоимость услуг. Кроме того, можно задаться вопросом совместимости этой формы с другими платформами. В спецификации HTML5 появились дополнительные типы данных, вводимых пользователем, такие как `email`. Это может решить проблему с iOS, Android и другими

мобильными платформами. Подробнее о подобных нововведениях в HTML5 см. в книге «HTML5 и CSS3. Веб-разработка по стандартам нового поколения» («HTML5 and CSS3: Develop with Tomorrow's Standards Today») [Hog10].

Смотрите также

- Рецепт 19. Виджет, встраиваемый на внешние сайты
- Рецепт 27. Простой блог с помощью Jekyll
- Рецепт 37. Как установить виртуальную машину
- Рецепт 36. Как разместить статический сайт на Dropbox
- Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake

Рецепт 18. Межсайтовый доступ с помощью JSONP

Задача

Мы хотим обратиться к данным, хранящимся на другом домене, однако серверный язык не позволяет нам этого сделать либо из-за ограничений веб-сервера, либо потому, что мы хотим, чтобы загрузка происходила в браузере пользователя. Обычные API-вызовы, обращенные к внешним сайтам, не решают эту проблему из-за правила ограничения домена[1]: согласно этому правилу, клиентские языки программирования, такие как JavaScript, не имеют доступа к веб-страницам, размещенным на других доменах.

Ингредиенты

- jQuery
- Удаленный сервер, возвращающий данные в формате JSONP
- Flickr API Key[2]

Решение

Чтобы загрузить удаленные данные с сервера, расположенного на другом домене, можно использовать JSONP или «JSON с подкладкой». Это значит, что мы будем попрежнему иметь дело с данными в формате JSON, но они будут представлены в виде вызова функции. При загрузке скрипта с удаленного сервера браузер будет пытаться запустить эту функцию в том случае, если она есть на странице; при этом данные в формате JSON будут переданы в качестве переменной. Таким образом, чтобы получить данные с удаленного сайта, нам нужно всего лишь создать такую функцию и описать в ней, как следует обработать JSON.

Мы хотим загрузить на нашу страницу двенадцать самых лучших фотографий, используя Flickr API. Хотя некоторые API позволяют задавать имя функции, в которую помещается загружаемый контент, Flickr API всегда использует для этого функцию `jsonFlickrApi()`. Поэтому если мы хотим загружать данные с Flickr, нам нужно будет создать на странице именно такую функцию.

Начнем с создания пустой страницы, то есть страницы без какого-либо контента в теге `<body>`. Все, что должно появиться на странице, будет загружаться на нее динамически.

Для начала напишем функцию, которая будет загружать наши фотографии с Flickr. Внутри `loadPhotos()` мы зададим ключ API, метод Flickr, который мы будем использовать, а также количество фотографий, которые мы хотим загрузить. Полный список методов Flickr можно найти в документации этого API[3].

---

[1] https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript
[2] http://www.flickr.com/services/api/keys/
[3] http://www.flickr.com/services/api/

jsonp/index.html

```
function loadPhotos(){
  var apiKey = '98956b44cd9ee04132c7f3595b2fa59e';
  var flickrMethod = 'flickr.interestingness.getList';
  var photoCount = '12';
  var extras = 'url_s';
  $.ajax({
    url:'http://www.flickr.com/services/rest/?method='+flickrMethod+
      '&format=json&api_key='+apiKey+
      '&extras='+extras+
      '&per_page='+photoCount,
    dataType: "jsonp"
  });
}
```

Эти переменные нужны для того, чтобы нам было проще изменять запрашиваемые данные, не разбирая URL по частям в каждом отдельном случае. Также мы добавили к запросу дополнительный атрибут `url_s`, чтобы в возвращаемых данных содержался URL уменьшенной версии фотографий. Далее функция jQuery `ajax()` вызывает Flickr. Так как данные будут запрашиваться с другого домена, мы задаем значение `dataType`, равное `"jsonp"`.

Теперь напишем функцию, которая будет загружать данные, возвращенные с Flickr API. Ответ содержит массу разнообразной информации, включая общее число доступных страниц, на случай, если мы захотим загрузить еще несколько фотографий; пока мы ограничимся двенадцатью.

jsonp/flickr_response.html

```
jsonFlickrApi({
  "photos": {
    "page": 1,
    "pages": 250,
    "perpage": 2,
    "total": 500,
    "photo": [
      {
        "id": "5889925003",
        "owner": "12386438@N04",
        "secret": "51c74e7c3e",
        "server": "6034",
        "farm": 7,
        "title": "",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0,
        "url_s": "http:\/\/farm7.static.flickr.com\/1\/image_m.jpg",
        "height_s": "160",
        "width_s": "240"
      },
      ...
```

```
    ]
  },
  "stat": "ok"
})
```

Фотографии, полученные с Flickr, помещены в массив с говорящим названием `photos`. Следовательно, нам нужно пройти по всем элементам этого массива и для каждого из них создать тег `<img>`, который мы добавим на страницу.

jsonp/index.html

```
function jsonFlickrApi(data){
  $.each(data.photos.photo, function(i,photo){
   var imageTag = $('<img>');
    imageTag.attr('src', photo.url_s);
    $('body').append(imageTag);
  });
}
```

Для этого мы вызовем `$.each(data.photos.photo, function(i,photo){...}`. На каждом шаге цикла мы будем оформлять новую фотографию, создавая для нее тег `<img>` и задавая в качестве значения атрибута `src` URL уменьшенной фотографии (который мы получили, указав в строке запроса значение параметра `extras`, равное `url_s`). Готовый элемент `<img>` мы добавляем в тело страницы и в результате получаем готовую галерею самых лучших фотографий с Flickr.

Когда будет готова объектная модель документа, нам останется только вызвать `loadPhotos()`, и мы получим целую страницу фотографий.

jsonp/index.html

```
$(function(){
  loadPhotos();
});
```

Таким образом, JSONP позволяет достаточно легко загружать динамический контент с внешних сайтов, не обращаясь к другим серверным языкам.

Дополнительные возможности

Предположим, что некоторые из возможностей нашей страницы полностью зависят от какого-то внешнего сайта, причем информацию о статусе системы этого сайта можно получить с помощью JSONP. Тогда мы могли бы проверять этот статус через фиксированный промежуток времени, например 60 секунд, и в случае его изменения обновлять нашу страницу.

Поскольку все, о чем мы здесь говорили, происходит в клиентской части приложения, нам не придется беспокоиться о дополнительной нагрузке на сервер. Тем не менее иногда приходится об этом думать. Чтобы избежать лишних запросов к серверу, мы рекомендуем добавить на страницу флажок, при установке которого данные будут регулярно обновляться, а при снятии — не будут обновляться.

Смотрите также

- Рецепт 19. Виджет, встраиваемый на внешние сайты
- Рецепт 14. Как структурировать код с помощью Backbone.js

Рецепт 19. Виджет, встраиваемый на внешние сайты

Задача

Виджет — это такое сочетание HTML, JavaScript и CSS, которое позволяет владельцам одного сайта добавлять на свою страницу код, отвечающий за добавление содержимого с другого сайта. Это может быть как общая информация о вашем сайте, так и данные, связанные с конкретным пользователем: тем самым виджеты расширяют границы нашего сайта и позволяют пользователям совместно использовать одно и то же содержимое. При всей простоте замысла создание виджета требует от нас нескольких нетривиальных действий. В частности, необходимо позаботиться о том, чтобы наш JavaScript-код не конфликтовал с JavaScript-кодом сайта пользователя. Кроме того, необходимо реализовать загрузку контента. Если правильно оформить код, наши функции не будут приводить к необратимым изменениям существующего кода или кода других виджетов. Иначе добавление нашего виджета может превратить работающую страницу в неработающую.

Ингредиенты

- jQuery
- JSONP

Решение

Виджеты представляют собой небольшие фрагменты кода, которые пользователи могут добавлять на свои веб-страницы; в результате эти фрагменты кода будут загружать контент с другого сайта. С помощью JavaScript и CSS мы можем реализовать загрузку содержимого с нашего сервера и добавление его на страницу, и тогда пользователю останется только загрузить с нашего сервера соответствующий JavaScript-код. Кроме того, поскольку код будет храниться не на сервере пользователя, мы сможем вносить в него любые изменения, и их результат будет виден на веб-страницах пользователей.

Мы хотим создать виджет, с помощью которого пользователи смогут добавить на свой сайт журнал коммитов из официального репозитория сайта Ruby on Rails[1]. Чтобы избежать конфликтов с JavaScript-кодом страницы пользователя, мы решили создать анонимную JavaScript-функцию. Нам также необходимо будет проверить, подключена ли библиотека jQuery, так как мы собираемся использовать некоторые ее методы и возможность краткой записи. Если она не подключена или требуется другая версия, мы загрузим свой вариант библиотеки. Создание и запуск виджета будет осуществляться посредством удаленной загрузки данных с помощью JSONP. JSONP позволяет получать данные с удаленного сервера с использованием JavaScript даже в тех случаях, когда эти данные размещены на другом домене. Загрузив необходимый контент, мы сгенерируем HTML-код и добавим его на страницу (рис. 3.9).

Процесс добавления виджета должен быть максимально простым. В нашем случае пользователю понадобится всего лишь две строки кода: ссылка на JavaScript-код и элемент `<div>`, в котором будет размещаться загруженный контент.

[1] https://github.com/rails/rails

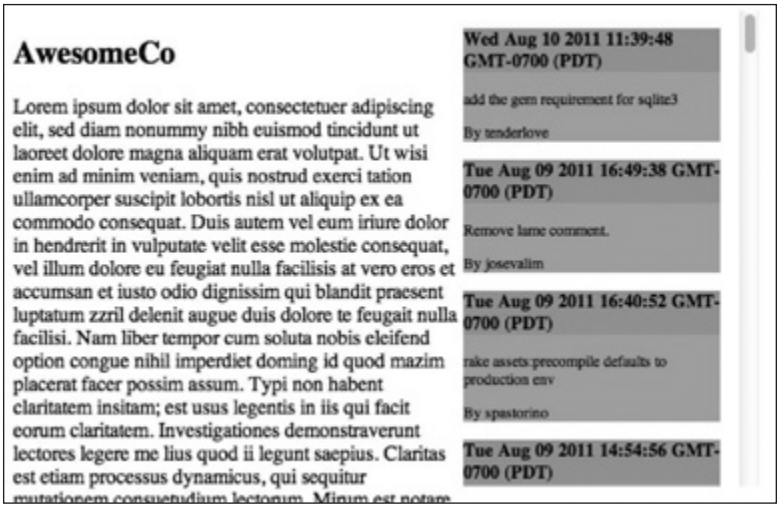

Рис. 3.9. Как наш виджет выглядит на самой обыкновенной странице

widget/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Widget Examples</title>
  </head>
  <body>
    <div style="width:350px; float:left;">
      <h2>AwesomeCo</h2>
      <p>
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam
        nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat
        volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation
        ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.
        Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse
        molestie consequat, vel illum dolore eu feugiat nulla facilisis at
        vero eros et accumsan et iusto odio dignissim qui blandit praesent
        luptatum zzril delenit augue duis dolore te feugait nulla facilisi.
      </p>
    </div>
    <script src="widget.js"></script>
    <div id="widget"></div>
  </body>
</html>
```

Сначала мы создадим анонимную функцию, которая защитит наш код от кода пользователя; это достаточно распространенный прием, который настоятельно рекомендуется использовать в подобных ситуациях. С одной стороны, важно, чтобы наш виджет не нарушил работу кода пользователя, а с другой — чтобы код пользователя не нарушил

работу нашего виджета. Эта функция будет автоматически запускаться сразу после загрузки скрипта, в результате чего на странице появится наш виджет.

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

Так как мы собираемся использовать библиотеку jQuery, необходимо убедиться, что ее возможности распространяются только на наш виджет, делая его изолированным от остального клиентского кода.

widget/widget.js

```
var jQuery;
if (window.jQuery === undefined || window.jQuery.fn.jquery !== '1.7') {
    var jquery_script = document.createElement('script');
    jquery_script.setAttribute("src",
        "http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js");
    jquery_script.setAttribute("type","text/javascript");
    jquery_script.onload = loadjQuery; // Загрузка во всех браузерах, кроме IE
    jquery_script.onreadystatechange = function () { // Загрузка в IE
        if (this.readyState == 'complete' || this.readyState == 'loaded') {
            loadjQuery();
        }
    };
    // Добавление jQuery в тег <head> страницы или в documentElement
    (document.getElementsByTagName("head")[0] ||
      document.documentElement).appendChild(jquery_script);
} else {
    // Версия jQuery — именно та, которая нам нужна
    jQuery = window.jQuery;
    widget();
}

function loadjQuery() {
    // Загрузка jQuery в режиме noConflict во избежание конфликтов
    // с другими библиотеками
    jQuery = window.jQuery.noConflict(true);
    widget();
}
```

Загруженную библиотеку jQuery мы записываем в переменную, которая видна только внутри нашей функции. Как для этой, так и для других переменных мы используем `var`, тем самым гарантируя, что названия этих переменных не вступят в конфликт с кодом страницы пользователя. Если нужная нам версия jQuery уже загружена, мы будем использовать ее. В противном случае мы добавляем в наш документ тег `<script>`. При загрузке локальной версии jQuery мы вызываем метод noConflict(), который позволяет избежать конфликтов с другими версиями jQuery или другими библиотеками, например Prototype, в которой `$()` используется для обозначения функции верхнего уровня.

Итак, мы добавили jQuery. Теперь можно загрузить данные с помощью JSONP и добавить их на страницу. Для загрузки самых последних коммитов с Rails мы воспользуемся API GitHub.

widget/widget.js

```
function widget() {
  jQuery(document).ready(function($) {
    // Загрузка данных
    var account = 'rails';
    var project = 'rails';
    var branch = 'master';

    $.ajax({
      url: 'http://github.com/api/v2/json/commits/list/'+
        account+
        '/'+project+
        '/'+branch,
      dataType: "jsonp",
      success: function(data){
        $.each(data.commits, function(i,commit){
          var commit_div = document.createElement('div');
          commit_div.setAttribute("class", "commit");
          commit_div.setAttribute("id","commit_"+commit.id);
          $('#widget').append(commit_div);
          $('#commit_'+commit.id).append("<h3>"+
            new Date(commit.committed_date)+
            "</h3><p>"+commit.message+"</p>"+
            "<p>By "+commit.committer.login+"</p>");
        });
      }
    });


    var css = $("<link>", {
      rel: "stylesheet",
      type: "text/css",
      href: "widget.css"
    });
    css.appendTo('head');
  });
}
```

Сначала функция `widget()` загружает данные с помощью JSONP и выполняет действия, необходимые для их добавления на страницу. Запрос выполняется с помощью функции jQuery `ajax()`. После этого выполняется вызов `success`, в результате которого для каждого коммита создается отдельный элемент `<div>`, состоящий из даты, автора и собственно сообщения. Этот элемент `<div>` мы добавляем в `<div> #widget`, который пользователь должен был добавить рядом с тегом `<script>`.

После загрузки данных можно перейти к созданию HTML-кода виджета и добавить его в тег `<div>`, который пользователь должен был поместить рядом с тегом `<script>`. Кроме того, нам нужно загрузить таблицу стилей и применить ее к нашему виджету.

widget/widget.css

```
#widget {
  width:230px;
  display:block;
  font-size: 12px;
  height: 370px;
  overflow-y: scroll;
}

.commit {
  background-color: #95B4D9;
  width:200px;
}

.commit h3 {
  display:block;
  background-color: #7DA7D9;
}
```

В этой таблице стилей мы задали высоту и ширину элемента и присвоили атрибуту `overflow-y` значение `scroll`. Это позволит нам не перегружать страницу пользователя нашими данными, даже если их объем будет достаточно велик.

Таким образом, нам удалось создать простой фрагмент кода, с помощью которого можно добавлять информацию с нашего сайта на другие веб-страницы. Вне зависимости от типа встраиваемых данных — сведений общего характера или персональных сведений — виджеты расширяют сферу распространения нашего контента и способствуют улучшению взаимодействия пользователей с нашим сайтом.

Дополнительные возможности

Наш виджет загружает контент только один раз — во время загрузки страницы пользователя. Однако он не содержит никакой информации, относящейся к данному конкретному пользователю или его аккаунту. А как сделать так, чтобы виджет мог опознавать пользователя и отправлять на удаленный сервер более точный запрос? Можно, например, использовать дополнительную переменную в URL тега `<script>`, с помощью которого мы динамически генерируем на сервере JavaScript-код. А можно создавать для каждого пользователя отдельный JavaScript-файл.

Возможны и интерактивные виджеты, которые не ограничиваются JSON- или XML-контентом. Попробуйте, к примеру, реализовать пролистывание записей с помощью мыши, а не полосы прокрутки; для этого удобно использовать jQuery. Данные можно загружать в момент загрузки страницы, а можно запрашивать с удаленного сервера по мере необходимости. Кроме того, можно сделать такой виждет, который будет автоматически обновляться каждые 60 секунд.

Еще более интерактивные виджеты могут запрашивать данные у пользователя и затем отправлять их нам по электронной почте или на сайт.

Виджеты могут быть очень разнообразными. Если какая-то информация может оказаться популярной среди пользователей или же вам необходим эффективный способ сбора данных для вашего сайта, мы рекомендуем вам не пренебрегать возможностью создать специальный виджет.

Рецепт 20. Уведомление о состоянии сайта с помощью JavaScript и CouchDB

Задача

Приложение, управляемое базой данных, может быть устроено достаточно сложным образом. Как правило, такое приложение включает в себя HTML, JavaScript, SQL-запросы и какой-либо серверный язык программирования, а также сервер базы данных. Чтобы соединить все эти компоненты в одно целое, веб-разработчик должен быть хорошо знаком с каждым из них. Именно поэтому нам необходимо простое альтернативное решение, в котором мы могли бы применить уже имеющиеся у нас знания, оставляя возможность для дальнейших усовершенствований по мере усложнения задачи.

Ингредиенты

- CouchDB[1]
- Аккаунт на Cloudant.com[2]
- CouchApp[3]
- jQuery
- Mustache[4]

Решение

CouchDB — это мощный и при этом компактный пакет, включающий одновременно и документальную базу данных, и веб-сервер. Таким образом, чтобы создать приложение, управляемое базой данных, достаточно написать HTML- и JavaScript-код и затем загрузить его на сервер CouchDB. Этот сервер и будет предоставлять наш контент пользователю. Запросы можно также отправлять с помощью JavaScript, и тогда нам не придется обращаться к другим языкам программирования. Таким образом, получается, что у нас есть уважительная причина для использования CouchDB.

Несмотря на все усилия, наши веб-серверы в последнее время работают нестабильно. Поэтому было бы неплохо иметь возможность сообщать пользователям о приостановке его работы, смягчив тем самым волну возмущения. С помощью CouchDB мы хотим создать и разместить в сети простой сайт, который будет оповещать пользователей о возникновении неполадок в нашей сети. Поскольку в момент оповещения наша сеть не будет работать, этот сайт необходимо разместить в другой сети. Поэтому вместо того чтобы создавать свой собственный сервер CouchDB, мы воспользуемся специальной службой под названием Cloudant, с помощью которой можно бесплатно создать свой собственный маленький сервер CouchDB и использовать его для тестирования.

Чтобы сэкономить время, мы решили использовать фреймворк CouchApp, предназначенный для создания и внедрения HTML- и JavaScript-приложений для CouchDB.

---

[1] http://couchdb.apache.org/
[2] https://cloudant.com/
[3] http://couchapp.org/page/index
[4] http://mustache.github.com/

CouchApp включает в себя инструменты для создания проектов и добавления файлов в базу данных CouchDB. Однако прежде чем перейти к созданию самого сайта, мы расскажем о том, как работает CouchDB.

Что такое CouchDB

CouchDB — это «документальная база данных». Вместо «таблиц» со «строками» в ней используются «коллекции» с «документами». Она отличается от реляционных баз данных наподобие MySQL и Oracle. Последние используют *реляционную модель*, в которой данные раскладываются на сущности, а между элементами устанавливаются определенные связи, что позволяет избежать дублирования данных. Чтобы получить из этих данных то, с чем можно в дальнейшем работать, мы обычно используем запросы. В реляционной модели «человек» и его адрес хранятся в разных таблицах. Это хорошее и надежное решение, однако оно подходит не для всех случаев.

Основная идея документальной базы данных состоит в том, чтобы хранить данные в виде документов. В таком случае их можно будет использовать несколько раз. При этом нам не важно, как один документ соотносится с другим. И хотя некоторые склонны противопоставлять реляционные базы данных документальным, на самом деле они скорее дополняют друг друга, так как используются в основном для разных целей.

В нашей системе каждое обновление состояния будет представлено в виде отдельного документа CouchDB. Эти документы будут отображаться с помощью простого интерфейса, который нам также необходимо будет создать. Начнем с создания базы данных и документа с данными о состоянии сайта.

Создание базы данных

В Cloudant существует специальный веб-интерфейс, предназначенный для создания новой базы данных. Когда мы в первый раз зайдем на Cloudant, используя новую учетную запись, система предложит нам это сделать. Свою базу данных мы назовем statuses.

С помощью интерфейса Cloudant можно также создавать документы. После выбора базы данных на экране появляется список ее документов. Чтобы добавить новый документ, достаточно щелкнуть на кнопке New Document (Новый документ). Так мы сможем создать несколько сообщений о состоянии сайта.

Документы представляют собой коллекцию ключей и значений в формате JSON. Так как каждое уведомление о состоянии сайта должно иметь заголовок и описание, оно будет представлено следующим образом.

```
{
  "title": "Unplanned Downtime",
  // Непредвиденный сбой
  "description": "Someone tripped over the power cord!"
  // Кто-то запутался в проводах
}
```

Добавлять новые поля можно с помощью мастера или же прямо в исходный код, который открывается кнопкой View Source (Просмотр исходного кода). Можно также использовать cURL (см. врезку «Использование cURL при работе с CouchDB» на с. 149).

Чтобы нам было что отображать на странице, добавим несколько документов, используя графический пользовательский интерфейс. Сначала мы создадим новый документ, заполним поля для заголовка и описания; поле `_id` мы оставим пустым (рис. 3.10).

Рис. 3.10. Добавление нового документа с помощью мастера Cloudant

ИСПОЛЬЗОВАНИЕ CURL ПРИ РАБОТЕ С COUCHDB

Поскольку CouchDB использует JSON RESTful API, мы можем создавать базы данных, обновлять документы и отправлять запросы из командной строки, не прибегая к инструментам графического пользовательского интерфейса. Все необходимые HTTP-запросы можно создавать с помощью специального инструмента командной строки cURL. Программа cURL устанавливается в большинстве операционных систем, включая OS X и Linux.

Так, вместо того чтобы создавать нашу базу данных statuses с помощью графического пользовательского интерфейса, мы могли бы отправить такой запрос PUT, используя cURL:

```
curl -X PUT http://awesomeco:****@awesomeco.cloudant.com/statuses
```

При этом данные можно было бы добавить так.

```
curl -X POST http://awesomeco:****@awesomeco.cloudant.com/statuses \
  -H "Content-Type: application/json" \
  -d '{"title":"Unplanned Downtime","description":"Someone tripped over
      the cord."}'
```

Флаг `-H` задает тип контента, а с помощью флага `-d` мы передаем строку с данными, которые необходимо отправить.

Таким образом, с помощью cURL настроить и заполнить базу данных можно гораздо быстрее, чем через веб-консоль. Более того, одни и те же действия можно выполнять несколько раз, объединяя фрагменты кода в отдельные скрипты.

Итак, в нашей базе данных появились некие данные. Теперь можно создать интерфейс, который отобразит их на странице.

Создание простого CouchApp

Приложения CouchApp можно размещать на самом сервере CouchDB. Приложение командной строки CouchApp включает себя инструменты для создания и настройки таких приложений. Кроме того, с его помощью можно добавлять файлы прямо в нашу базу данных.

Приложение CouchApp написано на языке Python, однако его можно установить в системах Windows и OS X, не устанавливая сам Python. Для этого достаточно зайти на страницу установки, найти пакет, подходящий для вашей системы, и установить его[1].

После установки CouchApp мы можем создать наше первое приложение.

```
$ couchapp generate app statuses
```

Эта строка создает новую папку statuses и помещает в нее новое приложение CouchApp. Оно содержит несколько подпапок, каждая из которых нужна для определенных целей.

В папку _attachments мы поместим HTML- и JavaScript-код, необходимый для работы нашего интерфейса. При добавлении этого CouchApp на сервер CouchDB содержимое этой папки будет рассматриваться как дизайн-документ.

В папке views хранятся «представления» CouchDB, которые соответствуют различным способам оформления наших документов. Так, например, из тридцати полей документа мы можем включить в представление только два или три в зависимости от конкретной задачи. Представления используются во многих типах баз данных, в том числе реляционных.

Далее из командной строки мы добавим это приложение в базу данных CouchDB.

```
$ couchapp push statuses \
http://awesomeco:****@awesomeco.cloudant.com/statuses

2011-07-20 14:24:28 [INFO] Visit your CouchApp here:
http://awesomeco.cloudant.com/statuses/_design/statuses/index.html
```

Эта строка добавляет папку statuses, в которой содержится все наше приложение, в базу данных statuses. Там она будет храниться в качестве «дизайн-документа». Чтобы посмотреть на наше приложение в браузере, можно открыть http://awesomeco.cloudant.com/statuses/_design/statuses/index.html, однако пока мы увидим там лишь самую обычную начальную страницу. Раз мы уже знаем, как добавлять данные на сервер, самое время перейти непосредственно к созданию приложения для оповещения пользователей о состоянии сайта.

Представление для даты запроса

В CouchDB представления нужны в первую очередь не для того, чтобы обращаться к документам напрямую, а для того, чтобы возвращаемые данные выглядели наилучшим образом. Когда мы обращаемся к представлению, CouchDB вызывает JavaScript-функцию, которая представляет данные в удобном для нас виде.

С помощью команды `couchapp` мы можем создавать файлы для представления. Поскольку мы хотим отображать на странице сообщения о состоянии сайта, давайте создадим представление под названием `messages`:

```
$ couchapp generate view statuses messages
```

Эта строка создает новую папку views/messages, состоящую из двух файлов: map.js и reduce.js. В файле map.js мы задаем список полей, которые будут отображаться на странице.

[1] http://couchapp.org/page/installing

Каждое сообщение имеет заголовок и описание, а также уникальный идентификатор и номер. Для этой страницы нам понадобятся только первые два, поэтому файл map.js будут выглядеть следующим образом:

couchapps/statuses/views/messages/map.js

```
function(doc) {
  emit( "messages", {
  title: doc.title,
  description: doc.description
  } )
}
```

Файл reduce.js нужен для того, чтобы упростить или переформулировать результаты запроса. Поскольку в данном случае это не нужно, мы просто удалим файл reduce.js.

Чтобы проверить, правильно ли работает такое представление, мы еще раз отправим наше приложение на удаленный сервер CouchDB на Cloudant.

```
$ couchapp push statuses \
http://awesomeco:****@awesomeco.cloudant.com/statuses
```

После этого откроем в браузере http://awesomeco.cloudant.com/statuses/_design/statuses/_view/messages. На экране должно появиться примерно следующее.

```
{"total_rows":2,"offset":0,"rows":[
  {"id":"02abeecc98362b3a26f85ea047bfaf5d","key":"messages","value":
    {"title":"Unscheduled Downtime",
    "description":"Someone tripped over the power cord!"}
  }
]}
```

Теперь у нас есть готовое представление, и мы можем приступить к оформлению страницы. Для этого напишем несложный HTML- и jQuery-код.

Отображение сообщений

Для того чтобы создать незамысловатый интерфейс, мы возьмем страницу по умолчанию _attachments/index.html и заменим большую часть ее кода следующим фрагментом.

couchapps/statuses/_attachments/index.html

```
  <body>
    <h1>AwesomeCo Status updates</h1>

    <div id="statuses">
      <p>Waiting...</p>
    </div>

    <script src="vendor/couchapp/loader.js"></script>
  </body>
</html>
```

Потом мы будем обновлять содержимое области `statuses` по мере поступления сведений из нашей базы данных.

В рецепте 10 («HTML с помощью Mustache») мы говорили о том, что при добавлении HTML-кода на страницу удобно использовать шаблоны. Наша страница использует JavaScript-файл loader.js, который загружает JavaScript-библиотеки, необходимые для CouchApp, в том числе библиотеки jQuery и jQuery Couch. Поэтому чтобы подключить Mustache, мы поместим копию файла mustache.js в vendor/couchapps/_attachments и добавим его в список скриптов.

couchapps/statuses/vendor/couchapp/_attachments/loader.js

```
couchapp_load([
  "/_utils/script/sha1.js",
  "/_utils/script/json2.js",
  "/_utils/script/jquery.js",
  "vendor/couchapp/mustache.js",
  "/_utils/script/jquery.couch.js",
  "vendor/couchapp/jquery.couch.app.js",
  "vendor/couchapp/jquery.couch.app.util.js",
  "vendor/couchapp/jquery.mustache.js",
  "vendor/couchapp/jquery.evently.js"
]);
```

Теперь можно добавить в index.html простой шаблон Mustache. Плагин jQuery CouchDB будет возвращать данные в следующем виде:

```
data = {
  rows: [
    {
      id: "9e227166d51569f2713728da59ff9d6b",
      key: "messages",
      value: {
        title: "Unplanned Downtime",
        description: "Someone tripped over the power cord."
      }
    }
  ]
};
```

Чтобы добавить в наш шаблон заголовок и описание каждого сообщения, мы перебираем все элементы массива `rows`. При этом к каждому полю мы добавляем префикс `value`, поскольку внутри объекта они хранятся именно с таким ключом. Теперь добавим этот шаблон в файл index.html.

couchapps/statuses/_attachments/index.html

```
<script type="text/html" id="template">
  {{#rows}}
  <div class="status">
    <h2>{{ value.title }}</h2>
    <p>{{ value.description }}</p>
  </div>
  {{/rows}}
</script>
```

Далее нам нужно обратиться к CouchDB, чтобы получить сообщения о состоянии сайта и добавить их в наш шаблон. Для этого мы создадим специальную функцию, которая будет располагаться в новом блоке `<script>` на странице index.html.

couchapps/statuses/_attachments/index.html

```
$db = $.couch.db("statuses");
var loadStatusMessages = function(){
  $db.view("statuses/messages",{
    success: function( data ) {
      var template = Mustache.to_html(
        $("#template").html(), data
      );
      $("#statuses").html(template);
    }
  });
}
```

Здесь используется тот же принцип, что и в рецепте 14 («Как структурировать код с помощью Backbone.js»), когда мы вызывали функцию `success` при успешной загрузке данных. Можно было определить и функцию обратного вызова `error`, однако в плагине CouchDB сообщение об ошибке предусмотрено по умолчанию.

Теперь нам нужно просто вызвать следующую функцию при загрузке страницы.

couchapps/statuses/_attachments/index.html

```
$(function(){
  loadStatusMessages();
});
```

Нам осталось только добавить наш CouchApp в базу данных в последний раз. Теперь, когда мы снова откроем эту страницу в браузере, она будет выглядеть так, как показано на рис. 3.11. После этого мы можем заняться разного рода доработками; после внесения изменений в код нам нужно будет добавить его на сервер.

Рис. 3.11. Страница с уведомлением о состоянии сайта

Дополнительные возможности

В этом разделе нам удалось создать очень простое, но превосходно работающее веб-приложение, использующее только HTML и JavaScript и основанное на базе данных CouchDB. Однако это далеко не предел наших возможностей. По мере усложнения кода можно работать над его структурой, используя различные JavaScript-фреймворки, например Backbone. В CouchApp есть фреймворк Evently, который позволяет решать проблемы, связанные с делегированием событий, — это может понадобиться при

создании более сложных пользовательских интерфейсов[1]. В нашем простом примере в этом не было необходимости, однако вполне возможно, что Evently окажется полезным в одном из ваших проектов.

У нашего приложения получился длинный и страшный URL. К счастью, CouchDB позволяет переписывать URL. Так, например, можно было бы сделать http://awesomeco.cloudant.com/statuses/_de-sign/statuses/index.html менее громоздким, сократив его до http://status.awesomeco.com/.

Следует помнить, что CouchDB — не просто хранилище данных клиентской части: его можно использовать и в серверных приложениях. Это надежное хранилище для документов, с которым очень удобно работать и которое можно легко расширять. Конечно, CouchDB способен решить не любую проблему, однако он все же занимает определенное место в области веб-разработки и особенно хорош при работе с данными, для которых наличие связей не является обязательным требованием.

Смотрите также

- Рецепт 10. HTML с помощью Mustache
- Рецепт 13. Модные клиентские интерфейсы с помощью Knockout.js
- Рецепт 14. Как структурировать код с помощью Backbone.js

[1] http://couchapp.org/page/evently

4

Мобильные устройства

При создании сайтов и приложений современный веб-разработчик вынужден учитывать интересы пользователей мобильных устройств. Ограничения пропускной способности, маленький размер экрана и новые способы взаимодействия с контентом ставят перед нами ряд интереснейших задач. В этой главе мы расскажем, как снизить потребление пропускной способности с помощью спрайтов, а также научим вас обрабатывать события мультитач и создавать интерфейсы с переходами.

Рецепт 21. Специальное оформление для мобильных устройств

Задача

При создании сайта веб-разработчик обязан учитывать множество различных факторов. Специфика браузера и разрешение экрана всегда сказывались на том, как выглядит наш контент. Чтобы сайт выглядел приемлемо и на ноутбуке с экраном 13", и на домашнем компьютере с экраном 30", разработчикам приходится прикладывать массу усилий. Еще несколько лет назад нам бы пришлось ломать голову над тем, как наш сайт будет выглядеть на экране карманного компьютера. Но после недавнего бума на рынке смартфонов и планшетов мы вынуждены заботиться не только о размере экрана, но и о том, что при повороте его ориентация может меняться.

Ингредиенты

- jQuery
- CSS Media Queries

Решение

Здесь мы можем использовать технологию CSS Media Queries, которая позволяет загружать таблицы стилей, специфичные для данного конкретного браузера. Media Queries стала использоваться еще во времена HTML4 и CSS2, однако с появлением CSS3 она была существенно расширена — в частности, были добавлены атрибуты `device-width` и `device-height`. Так у нас появилась потрясающая возможность создавать отдельные таблицы стилей для устройств с разной шириной и высотой экрана.

Рецепт 8 («Удобное раскрытие и сворачивание») был посвящен созданию списка товаров, узлы которого могут раскрываться и сворачиваться. Но недавно наша команда аналитиков обратила внимание на увеличение трафика со стороны пользователей мобильных устройств, 90 % которых составляют устройства iPhone. На данный момент наш сайт выглядит так, как показано на рис. 4.1. Из-за маленького размера шрифта пользоваться им с мобильного устройства крайне неудобно.

В качестве основы мы возьмем тот код, который мы создали в рецепте 8 («Удобное раскрытие и сворачивание»). Так как теперь большая часть трафика поступает от пользователей iPhone, мы начнем именно с этих устройств. Внутри раздела `<head>` мы с помощью нескольких тегов загрузим таблицы стилей, относящиеся к iPhone. Эти таблицы стилей будут храниться в файле iPhone.css в той же папке, что и style.css (см. «Рецепт 8. Удобное раскрытие и сворачивание»).

При подключении iPhone.css мы используем обычную ссылку на таблицу стилей, в которую добавлен дополнительный атрибут `media`. Чтобы эта таблица стилей использовалась только для устройств с максимальной шириной экрана 480 пикселов, мы задаем для этого атрибута значение `"only screen and (max-device-width: 480px)"`. Браузеры настольных компьютеров просто проигнорируют этот код, тогда как мобильные устройства с соответствующим разрешением будут использовать именно его.

targeting_mobile/index.html

```
<link rel="stylesheet" type="text/css" href="iPhone.css"
media="only screen and (max-device-width: 480px)">
    <meta name="viewport"
      content="width=device-width;
               height=device-height;
               maximum-scale=1.4;
               initial-scale=1.0;
               user-scalable=yes"/>
```

Рис. 4.1. Текущая версия списка товаров

Чтобы ограничить возможности просмотра контента в браузерах мобильных устройств, мы добавили мета-тег `viewport`. Дело в том, что такие браузеры не умеют автоматически размещать контент в зависимости от размера экрана; как правило, они отображают его так же, как он выглядел бы на экране настольного компьютера. Браузер стремится уместить на экране сразу весь контент, и в результате сайт выглядит мелким и пользователю приходится изменять масштаб. Добавив мета-тег `viewport`, мы заставим браузер автоматически подгонять контент под ширину экрана.

Теперь давайте посмотрим, как можно улучшить оформление списка, учитывая особенности устройства iPhone. Первым делом мы изменим насыщенность шрифта, сделав его полужирным; для этого зададим соответствующее значение `font-weight` в теге `<body>`, и в результате текст будет гораздо удобнее читать.

targeting_mobile/iPhone.css

```
body {
  font-weight: bold;
}
```

Далее мы сделаем так, чтобы элемент <ul> занимал большую часть экрана, но не выходил за его пределы. Чтобы эффективнее использовать рабочее пространство экрана, мы решили занять небольшую часть левого поля.

targeting_mobile/iPhone.css

```
ul.collapsible {
  width:430px;
  margin-left:-10px;
}
```

Ширина тега <ul> не должна превышать 430 пикселов — иначе он не поместится на экране. Чтобы немного передвинуть список влево, мы воспользуемся свойством margin-left.

Специфика сайтов для мобильных устройств заключается не только в визуальном оформлении, но и в способе взаимодействия пользователя с контентом. Например, чтобы выбрать объект на экране iPhone, достаточно прикосновения пальца; таким образом, в зону прикосновения попадает не один пиксел, как в случае с мышью. Поэтому мы решили увеличить расстояние между элементами <li>, чтобы пользователю было проще выбрать нужную ссылку.

targeting_mobile/iPhone.css

```
ul li{
  padding-top:10px;
}
```

Наконец, чтобы со списком было удобнее работать, добавим немного места вокруг знаков «+» и «–», которые показывают, какие узлы списка свернуты, а какие — нет.

targeting_mobile/iPhone.css

```
ul.collapsible li:before {
  width: 20px;
}
```

Если вы откроете эту страницу на iPhone (рис. 4.2), вы увидите, что теперь она гораздо лучше выглядит на маленьком экране. Мы попробовали открыть ее на Android с таким же разрешением экрана (рис. 4.3).

Теперь благодаря Media Queries мы можем создавать внешнее оформление сайта для самых разных устройств и ориентаций экрана. А поскольку на мобильных устройствах взаимодействие пользователя с контентом осуществляется иначе, чем на настольных компьютерах, некоторые элементы поведения должны зависеть от типа устройства; в этом нам также может помочь Media Queries.

Дополнительные возможности

Следующий шаг — реализация навигации для данного типа устройства. Кроме того, можно добавить специальное выделение к определенным типам контента — например, телефонным номерам и именам. А если вы захотите создать классическое оформление iOS, подключите к Media Queries «iOS Inspired jQuery Mobile Theme» Тейта Брауна[1].

[1] https://github.com/taitems/iOS-Inspired-jQuery-Mobile-Theme

Рис. 4.2. Список товаров на экране iPhone

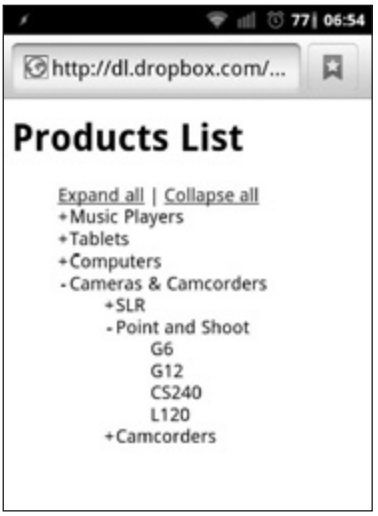

Рис. 4.3. Список товаров на экране Android

Некоторые фреймворки, такие как Skeleton[1], изначально поддерживают Media Query. Более подробно мы расскажем об этом позже в рецепте 26 («Удобный дизайн с помощью сетки»).

Смотрите также

- Рецепт 36. Как разместить статический сайт на Dropbox
- Рецепт 25. Спрайты с помощью CSS
- Рецепт 24. Интерфейсы с помощью jQuery Mobile
- Рецепт 26. Удобный дизайн с помощью сетки
- «HTML5 и CSS3. Веб-разработка по стандартам нового поколения» («HTML5 and CSS3: Develop with Tomorrow's Standards Today») [Hog10]

[1] http://www.getskeleton.com/

Рецепт 22. Меню, раскрывающееся при касании

Задача

Многие современные сайты используют навигацию с помощью раскрывающихся меню. Поэтому на сегодняшний день существуют надежные общепринятые приемы создания таких элементов интерфейса. Чтобы реализовать их в браузерах для настольных компьютеров, ничего особенного не требуется — разве что немного волшебства в виде CSS. Однако в рецепте 23 («Drag and Drop на мобильных устройствах») мы уже говорили о том, что мобильные устройства, на которых нет мыши, требуют особого подхода: в частности, в них невозможно событие `:hover` — по крайней мере, оно работает непоследовательно. Тем не менее мы обязаны учесть все эти особенности и предоставить пользователям мобильных устройств те же возможности, что и пользователям настольных компьютеров.

Ингредиенты

- jQuery

Решение

Для начала мы позаботимся о том, чтобы весь контент можно было открыть без использования раскрывающихся меню. Для этого сделаем так, чтобы каждая ссылка верхнего уровня указывала на страницу со ссылками более низких уровней. Благодаря этому пользователь сможет обращаться к категориям любого уровня даже в случае, если раскрывающиеся меню не будут работать. На этом можно было бы закончить раздел, ведь нам уже удалось реализовать навигацию для мобильных устройств (рис. 4.4). Однако не нужно забывать о том, что возможности пользователей мобильных устройств должны совпадать с возможностями пользователей настольных компьютеров. Поэтому следующим нашим шагом будет все-таки создание раскрывающихся меню.

На настольных компьютерах за раскрывающиеся меню отвечает CSS-событие `:hover`. Однако без мыши «наведение» курсора на ссылку оказывается невозможным. Устройства iOS предлагают удобный альтернативный способ: первое прикосновение к ссылке `:hover` включает режим наведения, и только второе позволяет перейти по этой ссылке. К сожалению, хотя в других мобильных браузерах первое прикосновение к ссылке также включает режим наведения, для выхода из этого режима пользователь должен увести курсор с этой ссылки, не отпуская его, иначе ссылка будет открыта. В таком случае идея с раскрывающимся меню оказывается бессмысленной, поскольку такое меню появляется на экране на какую-то долю секунды, после чего открывается другая страница.

Рис. 4.4. Ссылки верхнего уровня без эффекта наведения

Чтобы решить проблему с таким нелогичным поведением, мы решили использовать во всех браузерах поведение по принципу iOS. Для этого нужно будет отслеживать все прикосновения к элементам веб-страницы. Если пользователь прикоснется к заголовку, поведение по умолчанию будет запускаться только в том случае, если система обнаружит два прикосновения подряд. Таким образом, нам нужно будет следить одновременно за несколькими событиями: прикосновением к любому элементу страницы, прикосновением к элементам категории верхнего уровня и прикосновением к элементам других категорий.

mobiledropdown/mobiledropdown.js

```
var lastTouchedElement;
$('html').live('click', function(event) {
  lastTouchedElement = event.target;
});
```

Для начала добавим глобальную переменную для обозначения последнего элемента страницы, к которому прикасался пользователь. Иначе мы не узнаем, чего хотел пользователь: выбрать категорию, скрыть меню (щелкнув на любом другом элементе страницы) или же пройти по ссылке из нераскрывающегося меню.

Далее мы будем ждать, когда пользователь коснется ссылки с названием верхней категории. При этом мы должны будем проверить, является ли эта ссылка последним элементом страницы, к которому прикасался пользователь. При одиночном касании мы запретим выполнение действия по умолчанию: переходить по этой ссылке пока еще не нужно. На этом этапе страница будет выглядеть так, как показано на рис. 4.5. И только если пользователь прикоснется к этой ссылке еще раз, ее наконец-то можно будет открыть. Исключением здесь будут устройства iOS: поскольку они и так работают по нужному нам сценарию, запрещать поведение по умолчанию для них не нужно.

mobiledropdown/mobiledropdown.js

```
function doNotTrackClicks() {
  return navigator.userAgent.match(/iPhone|iPad/i);
}
$('navbar.dropdown > ul > li').live('click', function(event) {
  if (!(doNotTrackClicks() || lastTouchedElement == event.target)) {
    event.preventDefault();
  }
  lastTouchedElement = event.target;
});
```

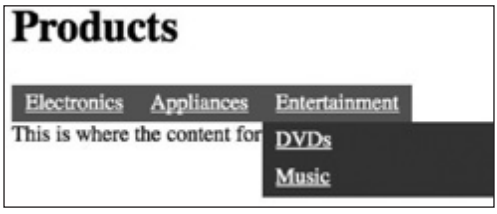

Рис. 4.5. Раскрывающееся меню после нажатия на категории Entertainment

Поскольку нажатая ссылка — это не тот элемент, к которому пользователь прикасался в последний раз, и поскольку данное устройство не принадлежит к типу iOS, мы запрещаем браузеру переходить по ссылке. Кроме того, мы обновляем значение переменной `lastTouchedElement`. Обычно за такое поведение отвечает обработчик событий, прикрепленный к элементу `<html>`, однако в нашей ситуации необходимо учесть еще одно событие.

Если запустить эту страницу в тестовом режиме, можно увидеть, что категории нижних уровней ведут себя точно так же, как категории верхнего уровня: чтобы пройти по ссылке нижнего уровня, нужно также нажать на нее дважды. Дело в том, что события щелка на таких ссылках «всплывают» к событиям категории верхнего уровня, наследуя ее поведение. Чтобы этого не происходило, в момент нажатия ссылки нижнего уровня мы вызовем метод `stopPropagation()`. (О распространении событий мы уже рассказывали во врезке «А почему бы просто не возвратить false?» на с. 67.)

mobiledropdown/mobiledropdown.js

```
$('navbar.dropdown li').live('click', function(event) {
  event.stopPropagation();
});
```

С помощью такого кода можно сделать поведение страницы одинаковым на разных платформах. А поскольку на странице каждой категории верхнего уровня приводятся ссылки на категории более низких уровней, с таким сайтом смогут работать также и пользователи устройств, не относящихся к классу смартфонов.

Дополнительные возможности

Такое поведение будет распространяться также и на браузеры настольных компьютеров. Это значит, что для перехода по ссылке пользователь должен будет щелкнуть на ней дважды. Однако можно сделать и так, чтобы этот код игнорировали не только устройства iPhone, но и любые не мобильные устройства. Для этого можно взять код, размещенный на http://detectmobilebrowsers.com/, и добавить его на наш сайт с помощью jQuery.

Смотрите также

- Рецепт 8. Удобное раскрытие и сворачивание

Рецепт 23. Drag and Drop на мобильных устройствах

В последние десять лет технология Drag and Drop применяется при создании огромного числа сайтов. Своим распространением она обязана простоте реализации: на сегодняшний день в открытом доступе находится множество соответствующих плагинов, а кроме того, написание такого плагина своими руками уже не представляет серьезной проблемы. Однако большинство таких решений основано на обработке событий щелчка мыши и, следовательно, не подходит для мобильных устройств. Чтобы решить эту проблему, нам придется использовать новые типы событий.

Ингредиенты

- jQuery

Решение

Браузеры мобильных устройств, таких как iPad, и интерфейсы всевозможных устройств с сенсорным экраном способны обрабатывать события, отличные от стандартных `mouseup` и `mousedown`. Для наших целей прекрасно подходят два из них: `touchstart` и `touchend`.

На нашем сайте описание товара отображается во всплывающем окне, которое пользователь может перетащить в любое место экрана. Однако пользователи iPad жалуются, что им не удается переместить это окно. Внимательно изучив код, мы обнаружили, что перемещение окна на нашем сайте основано на обработке событий `mouseup` и `mousedown`. Поэтому мы решили уделить внимание пользователям мобильных устройств и создать для них те же возможности, что и для пользователей настольных компьютеров.

Оформление и верстка

Для обработки этих событий мы будем использовать JavaScript. Однако начать следует все же с разметки. Наша страница содержит неупорядоченный список товаров и скрытый `<div>` для перетаскиваемого окна.

dragndrop/index.html

```
<header>
  <h1>Products list</h1>
</header>
<div id='content'>
  <ul>
    <li><a href="product1.html" class="popup">
      AirPort Express Base Station
    </a></li>
    <li><a href="product1.html" class="popup">
      DVI to VGA Adapter
    </a></li>
  </ul>
</div>
<div class="popup_window draggable" style="display: none;">
```

продолжение ⇩

```
  <div class="header handle">
    <div class="header_text">Product description</div>
    <div class="close">X</div>
    <div class="clear"></div>
  </div>
  <div class="body"></div>
</div>
```

При этом нужно указать, что позиционирование всплывающего окна должно быть абсолютным. Для оформления нам потребуются следующие таблицы стилей.

dragndrop/style.css

```
.clear {
  clear: both;
}
.popup_window {
  width: 500px;
  height: 300px;
  border: 1px solid #000;
  position: absolute;
  top: 50px;
  left: 50px;
  background: #EEE;
}
.popup_window .header {
  width: 100%;
  display: block;
}
.popup_window .header .close {
  float: right;
  padding: 2px 5px;
  border: 1px solid #999;
  background: red;
  color: #FFF;
  cursor: pointer;
  margin: 0;
}
.popup_window .header:after {
  clear: both;
}
```

Кроме того, нам потребуется специальная страница товара, на которую будут указывать ссылки. Обычно такая страница создается на сервере, но в нашем примере мы для простоты будем использовать для всех товаров одну и ту же страницу product1.html. Все файлы мы поместим в одну папку.

dragndrop/product1.html

```
<h3>Product Name</h3>
<div class='product_details'>
```

```
  <missing>Need a real product page</missing>
  <p>This is a product description. Below is a list of features:</p>
  <ul>
    <li>Durable</li>
    <li>Fireproof</li>
    <li>Impenetrable</li>
    <li>Fuzzy</li>
  </ul>
</div>
```

Простой пример использования Drag and Drop

Сейчас при нажатии на ссылке пользователь переходит на новую страницу, содержащую сведения о товаре. Вместо этого мы хотим, чтобы информация открывалась во всплывающем окне. Чтобы знать, какие ссылки должны открываться во всплывающих окнах, мы добавим к ним класс `popup`.

dragndrop/dragndrop.js

```
$('.popup').live('click', updatePopup);
function updatePopup(event) {
  $.get($(event.target).attr('href'), [], updatePopupContent);
  return false;
}
function updatePopupContent(data) {
  var popupWindow = $('div.popup_window');
  popupWindow.find('.body').html($(data));
  popupWindow.fadeIn();
}
$('.popup_window .close').live('click', hidePopup);
function hidePopup() {
  $(this).parents('.popup_window').fadeOut();
  return false;
}
```

Эти функции позволяют скрывать и отображать всплывающие окна. Внешне все выглядит прекрасно; новый контент можно загружать динамически, и при этом большая часть страницы остается в поле зрения. Единственная проблема заключается в том, что новое окно заслоняет старое (рис. 4.6), и мы никак не можем его передвинуть. Сейчас самое время добавить такую возможность. Мы начнем с реализации перетаскиваемого окна для браузеров настольных компьютеров, а затем применим ту же логику к событиям касания.

dragndrop/dragndrop.js

```
$('.draggable .handle').live('mousedown', dragPopup);
function dragPopup(event) {
  event.preventDefault();
  var handle = $(event.target);
  var draggableWindow = $(handle.parents('.draggable')[0]);
```

продолжение ⇩

```
  draggableWindow.addClass('dragging');
  var cursor = event;
  var cursorOffset = {
    pageX: cursor.pageX - parseInt(draggableWindow.css('left')),
    pageY: cursor.pageY - parseInt(draggableWindow.css('top'))
  };
  $(document).mousemove(function(moveEvent) {
    observeMove(moveEvent, cursorOffset,
      moveEvent, draggableWindow)
  });
  $(document).mouseup(function(up_event) {
    unbindMovePopup(up_event, draggableWindow);
  });
}
function observeMove(event, cursorOffset, cursorPosition, draggableWindow) {
  event.preventDefault();
  var left = cursorPosition.pageX - cursorOffset.pageX;
  var top = cursorPosition.pageY - cursorOffset.pageY;
  draggableWindow.css('left', left).css('top', top);
}
function unbindMovePopup(event, draggableWindow) {
  draggableWindow.removeClass('dragging');
}
```

Рис. 4.6. Всплывающее окно заслоняет основной контент

Все начинается с того, что мы наблюдаем за элементами `<div>` с классом `handle`, находящимися внутри элемента с классом `draggable`. При нажатии мыши вызывается функция `dragPopup()`. После этого мы начинаем следить за событием `mousemove`. Каждый раз при перемещении курсора мыши мы обновляем координаты элемента `draggable_window`. Событие сообщает нам положение курсора мыши. С его помощью нам нужно вычислить положение левого верхнего угла перетаскиваемого элемента. Для этого мы определяем расстояние (отдельно по горизонтали и по вертикали) между начальным положением окна и местом первого щелчка мыши. Затем внутри функции

observeMove() мы учитываем этот сдвиг, вычитая его из координат перемещаемого курсора мыши.

Чтобы понять, в какой момент необходимо остановить перемещение окна, мы добавим обработчик события mouseup. В случае запуска этого события нам нужно будет отменить некоторые изменения, выполненные с момента запуска mousedown, а именно перестать следить за mousemove и удалить дополнительный класс, который мы добавили к draggable_window.

Перенесение логики Drag and Drop на мобильные устройства

К счастью, после всей проделанной работы нам не составит труда перенести этот подход на мобильные устройства. Функция dragPopup() уже делает практически все, что нужно, если не считать того, что она ориентирована на события мыши. Поэтому нам достаточно написать аналогичный код, который вместо событий мыши обрабатывал бы события касания.

Для начала проверим, включена ли поддержка событий касания. Это важно, поскольку вызов функции, ориентированной на событие касания, на настольном компьютере вызовет сбой в работе страницы. Поэтому код, отвечающий за обработку событий касания, мы поместим внутри условного оператора isSupported().

dragndrop/dragndrop.js

```
function isTouchSupported() {
  return 'ontouchmove' in document.documentElement;
}
```

Далее добавим обработчик события для touchstart, который будет работать одновременно с обработчиком mousedown. Оба они будут запускать функцию dragPopup(). После этого событие touchstart запустит функцию dragPopup().

dragndrop/dragndrop.js

```
$('.draggable .handle').live('mousedown', dragPopup);
if (isTouchSupported()) {
  $('.draggable .handle').live('touchstart', dragPopup);
}
```

Так как пользователь может прикоснуться к экрану в нескольких местах одновременно, событие касания возвращает массив. Пока мы не рассматриваем передвижения с помощью нескольких пальцев, поэтому будем определять место касания по первому элементу массива. Эти координаты мы запишем в cursorPosition.

dragndrop/dragndrop.js

```
function dragPopup(event) {
  event.preventDefault();
  var handle = $(event.target);
  var draggableWindow = $(handle.parents('.draggable')[0]);
  draggableWindow.addClass('dragging');
  var cursor = event;
  if (isTouchSupported()) {
```

продолжение ⇩

```
    cursor = event.originalEvent.touches[0];
  }
  var cursorOffset = {
    pageX: cursor.pageX - parseInt(draggableWindow.css('left')),
    pageY: cursor.pageY - parseInt(draggableWindow.css('top'))
  };

  if (isTouchSupported()) {
    $(document).bind('touchmove', function(moveEvent) {
      var currentPosition = moveEvent.originalEvent.touches[0];
      observeMove(moveEvent, cursorOffset,
        currentPosition, draggableWindow);
    });
    $(document).bind('touchend', function(upEvent) {
      unbindMovePopup(upEvent, draggableWindow);
    });
  } else {
    $(document).mousemove(function(moveEvent) {
      observeMove(moveEvent, cursorOffset,
        moveEvent, draggableWindow)
    });
    $(document).mouseup(function(up_event) {
      unbindMovePopup(up_event, draggableWindow);
    });
  }
}
function unbindMovePopup(event, draggableWindow) {
  if (isTouchSupported()) {
    $(document).unbind('touchmove');
  } else {
    $(document).unbind('mousemove');
  }
  draggableWindow.removeClass('dragging');
}
```

К сожалению, в jQuery 1.7 возможности функции `live()`, использующейся для обработки событий касания, ограниченны. В частности, через событие jQuery невозможно обратиться к массиву `touches`, поэтому место касания приходится определять по исходному событию. Теперь нам осталось создать для события `touchmove` поведение, аналогичное `mousemove`. Для этого мы вызовем функцию `observeMove()`; в ней нам не придется ничего менять. Наконец, в случае запуска события `touchend` нам нужно будет остановить обработку `touchmove` — точно так же, как это было с `mouseup` и `mousemove`.

Дополнительные возможности

Зная, как обрабатывать одиночное событие касания, можно перейти к созданию более сложных команд, выполняемых движением нескольких пальцев. Так как событие касания возвращает массив из точек касания, с его помощью можно узнать, сколько всего пальцев задействовано и где каждый из них коснулся экрана. В результате вы сможете

распознавать такие события, как сведение и разведение пальцев, скольжение или любое другое действие, которое вы сами зададите. Здорово, что у веб-разработчиков наконец-то появилась возможность делать подобные вещи! Более подробно об этом API рассказывается на HTML5 Rocks[1].

Смотрите также

- Рецепт 31. Отладка JavaScript-кода
- Рецепт 22. Меню, раскрывающееся при касании

[1] http://www.html5rocks.com/en/mobile/touch/

Рецепт 24. Интерфейсы с помощью jQuery Mobile

Задача

Нам нужно разработать мобильный интерфейс для уже существующего веб-приложения. Конечно, идеальным решением было бы создание специальных интерфейсов для iOS и Android. Однако сейчас у нас нет на это ни времени, ни ресурсов, ни достаточных знаний. Поэтому мы попробуем соединить принципы создания обычных веб-приложений и приложений, специфических для данного типа устройств.

Ингредиенты

- jQuery
- jQuery Mobile[1]

Решение

Разработка приложений, специфических для данного типа устройств, не является простой задачей: для этого требуется слишком большой объем знаний в области программирования. Приложения для Android и iOS обычно используют языки Java и Objective-C, с которыми многие веб-разработчики никогда раньше не сталкивались. С помощью jQuery Mobile можно создавать интерфейсы, имитирующие поведение приложений, специально разработанных для платформ iOS и Android; при этом используются уже знакомые нам стандартные инструменты: HTML, JavaScript и CSS.

Чтобы понять, какие возможности предлагает нам jQuery Mobile, мы создадим сайт, на котором будет представлена продукция нашей компании. Мы планируем реализовать возможность просмотра и поиска товаров. В итоге должно получиться нечто похожее на интерфейс, изображенный на рис. 4.7.

Рис. 4.7. Стартовая страница на основе jQuery Mobile

Чтобы создать интерфейс с помощью jQuery Mobile, нам потребуется семантическая HTML-разметка и HTML5-атрибуты `data`. Используя такие атрибуты, мы сможем практически полностью отказаться от написания JavaScript-кода.

[1] http://jquerymobile.com/

Построение документа

Для начала подготовим наш HTML-документ к использованию jQuery Mobile. Так как приложение будет работать на сервере QEDServer, не забудьте проверить, что он запущен. После этого в папке public, относящейся к этому серверу, создайте файл index.html. Начнем со стандартного HTML-кода.

jquerymobile/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Incredible Products from AwesomeCo</title>

    <link rel="stylesheet"
      href="http://code.jquery.com/mobile/1.0rc1/jquery.mobile-1.0rc1.min.css">
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.4/jquery.min.js">
    </script>
     <script type="text/javascript"
      src="http://code.jquery.com/mobile/1.0rc1/jquery.mobile-1.0rc1.min.js">
    </script>
  </head>

  <body>
  </body>
</html>
```

Здесь используются три файла: таблицы стилей jQuery Mobile, библиотека jQuery и сам скрипт jQuery Mobile. Теперь можно заняться добавлением страниц и непосредственно контента. Для данной версии jQuery Mobile требуется jQuery 1.6.4.

Создание страниц

Приложение jQuery Mobile состоит из нескольких страниц. Эти страницы могут быть связаны друг с другом, но в каждый момент времени на экране отображается только одна из них. Для создания страницы в jQuery Mobile мы будем использовать элемент `<div>`, в котором для атрибута `data-role` будет задано значение `"page"`. Во время запуска приложения будет открываться та страница, которая окажется первой в элементе `body` нашего HTML-документа. Далее мы займемся созданием стартовой страницы.

jquerymobile/index.html

```
<div data-role="page">
  <div data-role="header">
    <h1>AwesomeCo</h1>
  </div>
  <div data-role="content">
  </div>
  <div data-role="footer">
    <h4>&copy; 2012 AwesomeCo</h4>
  </div>
</div>
```

Каждая страница подразделяется на три части: шапку, область контента и футер. В шапке содержится информация о текущей странице; она заключена в тег `<h1>`. Кроме того, как мы увидим позже, там могут располагаться навигационные кнопки. Область контента может содержать произвольное число абзацев, ссылок, списков, форм и любых других HTML-элементов. Футер является не обязательным разделом и обычно содержит сведения об авторских правах или любую другую общую информацию.

Итак, исходная страница готова. Теперь попробуем заполнить область контента. Чтобы пользователь мог переходить на другие страницы приложения, нам понадобится несколько кнопок.

jquerymobile/index.html

```
<p>Welcome to AwesomeCo, your number one source
  for all things awesome.</p>

<div data-role="controlgroup">
  <a href="#products" data-role="button">View All Products</a>
  <a href="#search" data-role="button">Search</a>
</div>
```

Сначала мы добавили отдельный абзац с информацией о приложении. Затем мы создали элемент `<div>` и присвоили его атрибуту `data-role` значение `"controlgroup"`. Эта роль убирает отступы между ссылками, так чтобы они располагались вплотную друг к другу (рис. 4.8). Чтобы ссылки выглядели как кнопки, мы указали для них значение атрибута `data-role`, равное `"button"`. Отсылка осуществляется за счет присвоения атрибуту `href` идентификатора необходимой страницы.

Рис. 4.8. Кнопки без значков

В итоге у нас получились приятные на вид кнопки. Однако мы можем сделать их еще лучше и понятнее, добавив специальные значки. Для каждой кнопки мы зададим значение атрибута `data-icon`. Список доступных значков можно найти в документации jQuery Mobile[1]; здесь нам понадобится стрелка вправо и значок поиска.

[1] http://jquerymobile.com/demos/1.0rc1/#/demos/1.0rc1/docs/buttons/buttons-icons.html

jquerymobile/index_icons.html

```
<div data-role="controlgroup">
  <a href="#products" data-role="button"
    data-icon="arrow-r">View All Products</a>
  <a href="#search" data-role="button"
    data-icon="search">Search</a>
</div>
```

После добавления этих кнопок навигацию стартовой страницы можно считать завершенной — у нас есть несколько удобных и понятных кнопок, отсылающих пользователя к самым разным частям приложения. Теперь наша начальная страница выглядит так, как мы и хотели (см. рис. 4.7).

Пока наши замечательные кнопки никуда не указывают. Чтобы убедиться в том, что они будут работать правильно, мы создадим еще одну страницу.

jquerymobile/index.html

```
<div data-role="page" id="products">
  <div data-role="header">
    <h1>Products</h1>
  </div>
  <div data-role="content">
  </div>
  <div data-role="footer">
    <h4>&copy; 2012 AwesomeCo</h4>
  </div>
</div>
```

Теперь после загрузки начальной страницы вы сможете нажать на кнопку просмотра товаров, и браузер перейдет на соответствующую страницу.

КАК ПРОТЕСТИРОВАТЬ JQUERY MOBILE

К сожалению, обычные браузеры не в состоянии справиться с тестированием jQuery Mobile; проблемы возникают, как правило, с размерами и масштабом. Поэтому с помощью обычного браузера можно сделать лишь общие выводы о работе приложения; для большей точности придется воспользоваться эмулятором браузера. Он работает так же, как и обычный браузер, но имеет размеры мобильного устройства. Самым распространенным эмулятором под Mac является iPhoney[1]. Для Windows и Linux существует бесплатное интернет-приложение testiphone.com[2].

С помощью таких браузеров очень удобно создавать веб-приложения на основе jQuery Mobile. При работе с iPhoney проверьте, чтобы галочка Zoom to Fit (Увеличить до размеров экрана) в меню View (Вид) была снята. Это важно для правильного определения масштаба.

Просмотр списка товаров

Теперь мы можем загрузить страницу со списком товаров, но на ней пока нет контента. Поскольку все данные поступают с сервера QEDServer, нам остается только оформить

[1] http://marketcircle.com/iphoney/

[2] http://testiphone.com/

их в виде списка с помощью jQuery. Для начала проверим, есть ли в базе данных наши товары; для этого откроем http://localhost:8080/products. Если там не окажется записей, добавьте несколько произвольных элементов.

Базовая структура для страницы с товарами у нас уже есть. Теперь мы создадим в области контента пустой элемент `<ul>`, в котором и будет размещаться список товаров.

jquerymobile/index.html

```
<div data-role="content">
  <ul id="products-list" data-role="listview"></ul>
</div>
```

Чтобы jQuery Mobile оформил элемент `<ul>` соответствующим образом, мы зададим для атрибута `data-role` значение `"listview"`. Мы также добавили к элементу `<ul>` идентификатор — так мы сможем легко обращаться к нему из JavaScript-кода. Если заново открыть наше приложение и перейти к списку товаров, на экране появится пустая страница. Загрузку товаров мы будем выполнять динамически по запросу пользователя. Для этого нам потребуются пользовательские события jQuery Mobile.

jquerymobile/index.html

```
$(function() {
  var productsPage = $("#products");
  var productsList = $("#products-list");

  productsPage.bind("pagebeforeshow", function() {
    $.mobile.showPageLoadingMsg();
    $.getJSON("/products.json", function(products) {
      productsList.html("");

      $.each(products, function(i, product) {
        productsList.append("<li><a href='#product'>" +
          product.name + "</a></li>");
      });

      productsList.listview("refresh");
      $.mobile.hidePageLoadingMsg();
    });
  });
});
```

С помощью события `pagebeforeshow` мы приостанавливаем процесс навигации, чтобы загрузить информацию о товарах. Первая строка обработчика события включает экран загрузки, сообщая пользователям о том, что система в данный момент занята. С помощью `getJSON()` мы получаем с сервера массив товаров. Далее все элементы этого массива по очереди добавляются в список. Создавая новый HTML-код, мы обновляем `listview`, таким образом сообщая jQuery Mobile о том, что для новых элементов необходимо создать оформление. После этого мы отключаем экран загрузки и загружаем новую страницу.

Теперь при попытке открыть страницу товаров на экране наконец-то появляется список (рис. 4.9).

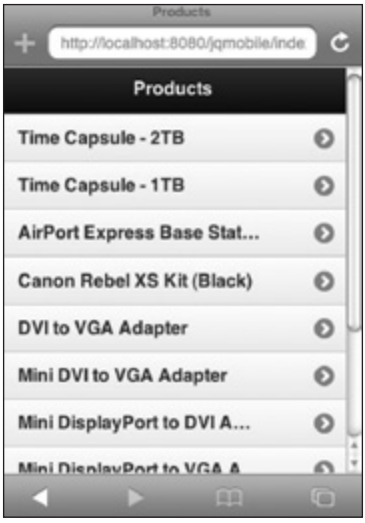

Рис. 4.9. Страница со списком товаров

И последнее: мы хотим, чтобы при нажатии на каждый товар из списка на экране появлялось дополнительное окно, содержащее всю доступную информацию об этом товаре. Конечно же, мы не станем создавать такую страницу отдельно для каждого товара; вместо этого мы будем использовать шаблон, в который данные о товаре будут загружаться динамически. Давайте вернемся к моменту создания контента для `listview`. Теперь нам нужно будет добавить к якорям атрибуты `data`, с помощью которых мы сможем запомнить идентификатор товара, к которому обращается пользователь. Для этого мы создадим пользовательский атрибут `data-product-id` и присвоим ему значение идентификатора, полученное с сервера.

jquerymobile/index_icons.html

```
$.each(products, function(i, product) {
  productsList.append("<li><a href='#product' data-product-id='" +
  product.id + "'>" + product.name + "</a></li>");
});
```

Далее нам нужно создать страницу, в которой будет отображаться информация о товаре. Пока эта страница не будет содержать контента: он будет загружаться динамически. Здесь, как и раньше, нам потребуется шапка, область контента и футер.

jquerymobile/index_icons.html

```
<div data-role="page" id="product">
  <div data-role="header" id="product-header">
    <a data-role="back" href="#products" data-direction="reverse">Back</a>
    <h1>Product</h1>
  </div>

  <div data-role="content" id="product-content">
    <p class="description"></p>
```

продолжение ⇩

```
    <span class="price"><strong></strong></span>
  </div>

  <div data-role="footer">
    <h4>&copy; 2012 AwesomeCo</h4>
  </div>
</div>
```

Внутри элемента `<div>` шапки мы решили создать кнопку Назад (Back), позволяющую «возвращаться» по истории посещений веб-страниц. Благодаря значению `reverse` для атрибута `data-direction` переход будет выполняться слева направо. Теперь, чтобы отобразить товар на странице, нужно приостановить навигацию и загрузить данные. Раньше нам приходилось запрашивать с сервера информацию о нескольких товарах. На этот раз нам нужны сведения только об одном товаре; эти сведения будут являться единственным источником контента. Чтобы завершить процесс навигации, напишем еще один JavaScript-код.

jquerymobile/index_icons.html

```
var productPage = $("#product");

$("#products a").live("tap", function() {
  var productID = $(this).attr("data-product-id");
  $.mobile.showPageLoadingMsg();
  $.getJSON("/products/" + productID + ".json", function(product) {
    $("#product-header h1").text(product.name);
    $("#product-content p.description").text(product.description);
    $("#product-content span.price strong").text("$" + product.price);
  });
  $.mobile.hidePageLoadingMsg();
  $.mobile.changePage($("#product"));
});
```

Сначала мы привязываем обработчик к событию `"tap"`. В jQuery Mobile это событие является пользовательским. Собственные события касания различных браузеров мобильных устройств сильно отличаются друг от друга. В такой ситуации событие касания jQuery Mobile является прекрасным решением, так как способно сглаживать все эти несоответствия. Далее, поскольку мы знаем идентификатор товара, мы можем вызвать `getJSON()`. Внутри вызова Ajax мы добавим полученные данные в текст страницы с информацией о товаре. Последняя строка наконец-то запустит нашу страницу.

Итак, у нас есть удобный и приятный интерфейс для просмотра товаров и информации о них. На рис. 4.10 можно увидеть, как будет выглядеть страница со сведениями об отдельном товаре.

Дополнительные возможности

На начальной странице нашего приложения есть кнопка поиска, однако страницу поиска мы так и не создали. Используя принцип создания динамических страниц, описанный в этой главе, вы сможете легко создать такую страницу. Для начала попробуйте создать новую страницу и добавьте на нее форму для отправки поискового запроса.

Рис. 4.10. Отдельный товар

```
<form id="search-form">
  <input type="search" name="query" id="search-query">
  <input type="submit" name="submit" value="Submit">
</form>
```

В этой форме есть одно поле ввода: поисковый запрос. Далее нам нужно будет ждать события `submit`, после чего отправлять соответствующий запрос с помощью Ajax. Используя встроенный поиск сервера QEDServer, можно создать необходимый код на основе уже имеющегося у нас кода для просмотра товаров. Далее, отправляя запрос getJSON(), нужно будет передать в качестве параметра поисковый запрос.

```
$.getJSON("/products.json?q=" + $('#search-query').val(), function(data) {
  // что нужно изменить на странице со списком товаров
});
```

Наконец, для перехода на страницу со списком товаров можно будет просто использовать `changePage()`. Фреймворк jQuery Mobile обладает множеством еще более интересных и сложных функций, с помощью которых можно создать поистине мощное приложение. Поэтому мы очень рекомендуем ознакомиться с документацией и примерами на сайте jQuery Mobile[1].

Смотрите также

- Рецепт 10. HTML с помощью Mustache
- Рецепт 18. Межсайтовый доступ с помощью JSONP
- Рецепт 21. Специальное оформление для мобильных устройств
- Рецепт 22. Меню, раскрывающееся при касании

[1] http://jquerymobile.com/demos/1.0b2/#/demos/1.0b2/

Рецепт 25. Спрайты с помощью CSS

Задача

Из-за роста стоимости трафика и ограничений на его объем загрузка изображений на телефоне или другом мобильном устройстве может оказаться слишком затратной с точки зрения как денег, так и времени. Поэтому нам хотелось бы минимизировать эти издержки, а заодно добиться оптимального времени загрузки контента, предоставив пользователю мобильного устройства удобное приложение, которое не нанесет серьезного ущерба его тарифному плану.

Ингредиенты

- CSS

Решение

Рецепт 21 («Специальное оформление для мобильных устройств») был посвящен созданию мобильного интерфейса для списка товаров из рецепта 8 («Удобное раскрытие и сворачивание»). Теперь нам нужно немного оживить сайт, добавив несколько графических элементов и изображений. Однако мы не хотим, чтобы загрузка этих изображений отнимала большую часть пропускной способности; мы также хотим добиться быстрой загрузки страниц. С помощью CSS-спрайтов можно уменьшить число загружаемых файлов, объединяя несколько значков в одно изображение и затем обрезая его соответствующим образом с помощью CSS-свойств. Таким образом, мы загружаем только один файл, но используем его для разных целей.

В отделе графики нам предложили использовать на нашем сайте следующее спрайтовое изображение.

Этот спрайт состоит из знаков + и –, которые мы хотим поставить вместо текстовых символов, обозначающих свернутые и развернутые узлы списка. Это изображение вы можете найти в исходных кодах, размещенных на сайте нашей книги.

В нашем проекте нужно создать отдельную папку для изображений и поместить в нее файл expand_collapse_sprite.png. В этом разделе мы будем работать с файлом iPhone.css, созданным в рецепте 21 «Специальное оформление для мобильных устройств».

Напомним, что в файле style.css из рецепта 8 «Удобное раскрытие и сворачивание» описаны два CSS-правила. Они отвечают за то, какой контент будет отображаться на странице.

css_sprites/style.css

```
ul.collapsible li.expanded:before {
  content: '-';
}
ul.collapsible li.collapsed:before {
  content: '+';
}
```

В iPhone.css мы переопределим эти правила, так чтобы браузер использовал не текстовые символы, а изображения. Для добавления спрайтов мы зададим CSS-атрибут `background` и некоторые параметры расположения, так чтобы на экране отображалось не все изображение, а только его часть.

css_sprites/iPhone.css

```
ul.collapsible li.expanded:before {
  content: '';
  width:30px;
  height:20px;
  background:url(images/expand_collapse_sprite.png) 0 -5px;
}
ul.collapsible li.collapsed:before {
  content: '';
  width:30px;
  height:25px;
  background:url(images/expand_collapse_sprite.png) 0 -30px;
}
```

В первой строке этих CSS-правил в качестве контента мы задаем пустую строку, таким образом перекрывая ранее заданный текстовый символ. Далее, поскольку нам нужно отобразить только часть изображения, мы задаем его высоту и ширину. Атрибут background также задает и координаты спрайта, сдвигая изображение соответствующим образом относительно окна, заданного значениями `width` и `height`.

Поскольку в верхней части нашего изображения есть немного свободного места, мы можем начать отсчет с точки –5 пикселов. Слева свободного пространства практически нет, так что мы начнем с 0. Второй значок располагается прямо под первым, поэтому для него отправной будет точка, расположенная на 30 пикселов ниже. Таким образом, при начальной координате –30 пикселов мы увидим вместо знака «+» знак «–». Теперь можно оценить плоды нашего труда (рис. 4.11).

Если в каком-то месте страницы вам необходимо использовать попеременно несколько разных изображений, CSS-спрайты являются превосходным решением. С их помощью все изображения можно объединить в одну картинку, а затем перемещать эту картинку так, чтобы на экране была видна только какая-то ее часть. Такой способ уменьшает потребляемую пропускную способность и объем передаваемых данных, позволяя снизить затраты на загрузку приложений, что крайне важно для пользователей мобильных устройств.

Дополнительные возможности

После того как мы добавили CSS-спрайт в таблицу стилей для мобильных устройств, самое время сделать то же самое для настольных компьютеров. Спрайты будут полезны во всех версиях вашего сайта, так как они уменьшают объем загружаемых данных, снижая, таким образом, время загрузки страницы.

Идея, лежащая в основе CSS-спрайтов, может быть использована и при создании изощренных анимированных изображений. Многие из интерактивных логотипов Google Doodle (картинок, появляющихся вместо логотипа Google и приуроченных к каким-либо важным датам) также основаны на CSS-спрайтах. Чтобы ознакомиться

Рис. 4.11. Список товаров с использованием спрайтов

с более сложными и интересными приемами их использования, попробуйте набрать в поисковой системе *Google Doodles CSS Sprites*.

Смотрите также

- Рецепт 36. Как разместить статический сайт на Dropbox
- Рецепт 24. Интерфейсы с помощью jQuery Mobile
- Рецепт 26. Удобный дизайн с помощью сетки
- «HTML5 и CSS3. Веб-разработка по стандартам нового поколения» («HTML5 and CSS3: Develop with Tomorrow's Standards Today») [Hog10]

Рабочий процесс

5

Решающее влияние на производительность оказывает выбор методов и инструментов. Как правило, веб-разработчики стремятся в первую очередь угодить пользователю, однако при этом они ни в коем случае не должны забывать и об организации своего собственного рабочего процесса. В этой главе мы обсудим различные схемы создания макетов, добавления контента, использования CSS и JavaScript, а также написания кода.

Рецепт 26. Удобный дизайн с помощью сетки

Задача

Еще до фактической реализации сайта заказчик или менеджер может попросить разработчика создать макет будущего проекта. С помощью такого макета можно проверить, насколько удобными для конечного пользователя будут его оформление и верстка; особенно это важно при создании интерфейсов для мобильных устройств и планшетов.

И хотя в нашем распоряжении есть множество самых разнообразных инструментов — от карандаша и бумаги до многофункциональных приложений, таких как OmniGraffle, Visio или Balsamiq Mockups, — нам бы не хотелось отказываться от привычных HTML и CSS, ведь в таком случае мы могли бы добавлять интерактивные элементы и, возможно, даже использовать фрагменты этого кода при создании будущего сайта.

Ингредиенты

- Skeleton[1]

Решение

Новые HTML- и CSS-фреймворки существенно упрощают процесс создания макета: они не только берут на себя решение сложных вопросов, связанных с CSS-верстской, но и учитывают специфику устройств с разными размерами экрана.

С помощью CSS-фреймворков, основанных на сетке, можно легко и быстро разместить на странице все необходимые элементы, не заботясь о том, какие значения нужно задать для свойств `float` и `clear`. Существует несколько таких фреймворков, и все они замечательные. Из них мы выбрали Skeleton благодаря встроенной поддержке различных размеров экрана.

Нам нужно создать макет для страницы с информацией об объекте недвижимости, разместив на ней несколько фотографий этого объекта, стоимость и некоторые сведения из мультилистинговой системы. При этом мы хотим, чтобы страница хорошо выглядела и на обычном ноутбуке, и на iPhone, предоставляя риелторам быстрый и удобный доступ к информации об объектах недвижимости. Этот макет мы хотим впоследствии использовать в качестве шаблона для действующего веб-приложения, поэтому текст мы зададим заранее, а вместо рисунков добавим заполнители. Но прежде чем перейти к созданию макета, мы расскажем о том, что такое Skeleton и как он работает.

Структура Skeleton

Как и другие похожие фреймворки, Skeleton берет за основу контейнер шириной 960 пикселов, выровненный по центру, и делит его на шестнадцать одинаковых столбцов. Далее ширина областей страницы измеряется количеством столбцов. Так, шапка страницы будет иметь ширину шестнадцать столбцов, а боковая панель — четыре (рис. 5.1). Используя надежные CSS-методы, Skeleton решает за вас все вопросы, связанные с обтеканием и выравниванием, а также задает подходящие значения межстрочного интервала и размера шрифта.

[1] http://getskeleton.com/

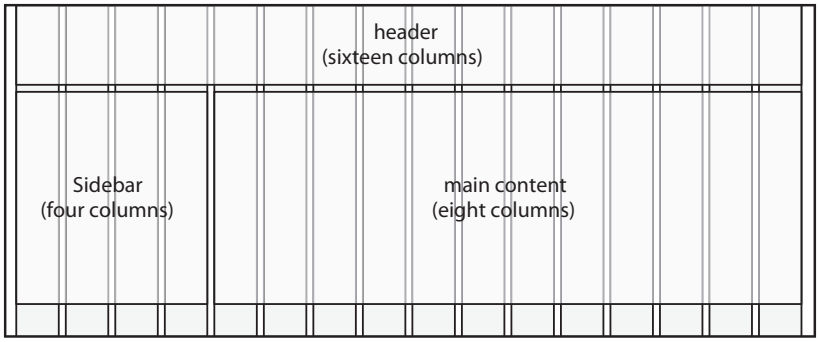

Рис. 5.1. Макет с двумя колонками на основе сетки Skeleton

Однако преимущества Skeleton не ограничиваются удобством CSS-верстки. После загрузки и распаковки Skeleton мы получаем готовую структуру размещения файлов, которая включает следующее: стандартный файл index.html, папку для JavaScript-файлов, папку для таблиц стилей и папку для изображений, в которой будут храниться значки, использующиеся по умолчанию (включая значки для «рабочего стола» iPhone). В этой структуре найдется место и для наших собственных таблиц стилей и JavaScript-кода.

Но это еще не все. Используя CSS Media Queries (см. «Рецепт 21. Специальное оформление для мобильных устройств»), Skeleton автоматически реагирует на изменение размера окна браузера и подстраивает интерфейс под размер экрана.

Теперь, зная кое-что о возможностях Skeleton, мы можем перейти к созданию нашего макета.

Макет

В шапку страницы мы поместим адрес объекта недвижимости. Первая колонка будет содержать информацию об этом объекте, вторая — фотографии. В итоге наша страница будет выглядеть так, как показано на рис. 5.2.

Рис. 5.2. Готовая страница

Версию Skeleton 1.1, которую мы будем использовать в этом рецепте, можно найти в исходных кодах этой книги. В качестве основы для шаблона Skeleton предлагает нам использовать уже готовый файл index.html. Давайте откроем его и удалим все, что находится между открывающим и закрывающим тегами `<div>` с классом `container`; сам контейнер мы оставим, поскольку он уже имеет нужную ширину 960 пикселов и выровнен по центру страницы.

Мы начнем с шапки страницы, которая будет содержать название сайта и адрес объекта недвижимости. Для этого воспользуемся тегом HTML5 `<header>`.

cssgrids/index.html

```
<header class="sixteen columns">
  <h1>SpotFindr</h1>
  <h3>123 Fake Street, Anytown USA 12345</h3>
</header>
```

В этом документе мы можем использовать любые семантические теги — например, `<section>`, `<header>` и `<footer>`. А поскольку наша страница основана на шаблоне Skeleton, раздел `<head>` по умолчанию подключает библиотеку HTMLShiv[1], так что мы сможем настроить оформление этих элементов и для старых версий браузеров.

Далее мы добавим левую колонку с информацией о стоимости объекта недвижимости и его кратким описанием. Для этого нам понадобится тег `<section>`. Мы хотим, чтобы этот раздел занимал ровно половину страницы, поэтому зададим его ширину равной восьми столбцам.

cssgrids/index.html

```
<section id="datasheet" class="eight columns">
 <h2 class="price">$109,900</h2>
  <p>
    Simple single-family home on the north side, within walking
    distance to schools and public transportation. New roof in 2005,
    central air in 2006. New windows and doors in 2010. Ready for you to
    move in!
  </p>
</section>
```

Теперь займемся правой колонкой. Для этого добавим еще одну область сразу после предыдущего тега `<section>`. Skeleton автоматически размещает колонки вплотную друг к другу, двигаясь слева направо, до тех пор пока общее число столбцов не достигнет шестнадцати; после этого он переходит на новую строку.

Так как у нас еще нет изображений объекта недвижимости, нам потребуются заполнители рисунка. С помощью сервиса Placehold.it[2] их можно создать прямо в теге `<img>`, указав при этом ссылку на сайт. Например, для добавления заполнителя шириной 460 пикселов и высотой 200 пикселов с текстом «Bedroom» («Ванная комната») достаточно написать такой запрос:

```
<img src="http://placehold.it/460x200?text=Bedroom">
```

[1] http://code.google.com/p/html5shiv/

[2] http://placehold.it/

В нашем макете таких заполнителей будет четыре.

cssgrids/index.html

```
<section class="eight columns">
  <img class="scale-with-grid"
       src="http://placehold.it/460x280&text=Exterior"
       alt="Exterior of house">
  <img src="http://placehold.it/150x100&text=Livingroom"
       alt="Livingroom">
  <img src="http://placehold.it/150x100&text=Kitchen"
       alt="Kitchen">
  <img src="http://placehold.it/150x100&text=Bedroom"
       alt="Bedroom">
</section>
```

При уменьшении размера окна браузера или просмотре страницы на устройстве с маленьким экраном наше изображение должно уменьшаться. С помощью класса `scale-with-grid` мы можем сделать так, чтобы Skeleton автоматически изменял ширину и высоту картинки в соответствии с новым размером окна.

Теперь нам остается добавить под описанием дома две колонки с дополнительной информацией. Эти две колонки мы хотим поместить внутри уже имеющейся левой колонки.

Колонки внутри колонки

На данный момент и в левой, и в правой колонке данные представлены в виде одной строки. Тем не менее Skeleton позволяет автоматически добиться удобного расположения фотографий в правой колонке. Однако не стоит всегда на это надеяться: иногда место перехода на новую строку лучше задать вручную.

Для этого внутри левой колонки после закрывающего тега `<p>` мы создаем новый элемент с классом `row`:

cssgrids/index.html

```
<div class="row">
</div>
```

В результате такой «очистки» плавающих блоков наш элемент начинается с новой строки. Внутри него мы можем создать две колонки. Поскольку ширина текущей области составляет восемь столбцов, ширина каждой из колонок будет равна четырем столбцам.

cssgrids/index.html

```
<div class="row">
  <div class="four columns alpha">
    <p><strong>Year Built:</strong> 1964</p>
    <p><strong>Bedrooms:</strong> 4</p>
    <p><strong>Baths:</strong> 1 Full/1 Half</p>
    <p><strong>Square footage:</strong> 1,144 (approx)</p>
    <p><strong>MLS#:</strong> 842089</p>
  </div>
```

продолжение ⇩

```
  <div class="four columns omega">
    <p><strong>Foundation:</strong> Poured</p>
    <p><strong>Heat:</strong> Gas Forced Air</p>
    <p><strong>Electrical</strong>Circuit Breaker</p>
    <p><strong>Water</strong> City Water</p>
    <p><strong>Sewer</strong> City Sewer</p>
  </div>
</div>
```

Вокруг каждой колонки Skeleton добавляет отступы. Поэтому, когда мы создаем колонки внутри колонок, мы должны указать, что дополнительные отступы нам не нужны. Для этого мы добавляем к первой колонке в ряду класс `alpha`, убирая левый отступ, а к последней колонке — класс `omega`, убирая, соответственно, правый.

Итак, нам удалось за небольшое время создать приятную на вид страницу, которая к тому же реагирует на изменение размера экрана: ее элементы меняют свое расположение (рис. 5.3). Не хватает еще одной детали — рамки с тенью вокруг контейнера. Правда, она будет актуальна только в полноэкранной версии страницы.

Рис. 5.3. Наша страница на экране iPhone

Оформление с помощью Media Queries

В файле stylesheets/layout.css, содержащем заполнители рисунков и медиа-запросы, мы можем задать настройки внешнего оформления нашего макета. Чтобы добавить границу, мы находим в этом файле раздел «Site Styles» («Оформление сайта») и пишем там что-то наподобие следующего кода. Этот код задает для страницы серый фон,

а для контейнера — белый, а также добавляет легкую тень, которая будет отображаться в браузерах с поддержкой соответствующих свойств.

cssgrids/stylesheets/layout.css

```
/* #Site Styles
================================================ */
body{
  background-color: #ddd;
  margin-top: 20px;
}
.container{
  background-color: #fff;
  -webkit-box-shadow: 5px 5px 5px #bbb;
     -moz-box-shadow: 5px 5px 5px #bbb;
       -o-box-shadow: 5px 5px 5px #bbb;
}
```

В обычном браузере такая тень будет прекрасным украшением сайта. К сожалению, на экране мобильного телефона она будет занимать слишком много места, поэтому нам придется добавить в раздел «Anything smaller than standard 960» («Любое устройство с размером экрана менее 960 пикселов») несколько строк кода, которые уберут ненужную границу.

cssgrids/stylesheets/layout.css

```
/* Anything smaller than standard 960 */
@media only screen and (max-width: 959px) {
  body{
    background-color: #fff;
  }

  .container{
    background-color: #fff;
    -webkit-box-shadow: none;
       -moz-box-shadow: none;
         -o-box-shadow: none;
  }
}
```

Этот код отменяет стили, которые мы перед этим добавили, при просмотре сайта на маленьком экране или в браузере мобильного устройства. Используя все то, что дает нам Skeleton, мы можем задать все необходимые настройки и для других размеров экрана.

Дополнительные возможности

Стандартный шаблон Skeleton использует самые передовые методы CSS и JavaScript и потому заслуживает более детального рассмотрения. В частности, он обеспечивает поддержку HTTP и HTTPS, выполняя загрузку jQuery из Сети доставки контента Google с учетом текущего протокола[1]. С помощью условных комментариев мож-

[1] http://paulirish.com/2010/the-protocol-relative-url/

но добавлять фрагменты кода, предназначенные для старых версий Internet Explorer и других браузеров. Так, используя Respond.js[1] или любой другой аналогичный скрипт, можно добавить поддержку медиа-запросов в Internet Explorer.

ВОПРОС/ОТВЕТ. РАЗВЕ В ТАКИХ ФРЕЙМВОРКАХ НЕ СМЕШИВАЮТСЯ ОФОРМЛЕНИЕ И ПРЕДСТАВЛЕНИЕ?

Вообще-то да. Если мы создадим элемент `<div>` с классом `four columns`, а потом захотим что-то переделать, нам придется обращаться к разметке. Сторонники «чистоты» кода скорее всего сочли бы такую ситуацию неприемлемой. И хотя эта идея все же лучше, чем `class="redImportantText"`, в ней действительно смешиваются контент и его представление.

Как показывает практика, вместо того чтобы переделывать старые макеты, разработчики предпочитают создавать новые. Поэтому шаблоны применяются многократно скорее в теории, чем на практике. В системах, подобных этой, мы вынуждены пожертвовать семантической разметкой в пользу производительности. В любом случае, как вы уже заметили, Skeleton и другие аналогичные фреймворки позволяют легко и быстро создавать прототипы, что само по себе здорово независимо от того, будете ли вы использовать этот макет в дальнейшем.

Если вас все же не устраивает такое положение вещей, но вы хотите сохранить общую идею, описанную в этом рецепте, мы рекомендуем вам Compass — фреймворк, предназначенный для создания таблиц стилей, но не основанный на сетке[2]. Compass использует Sass, о котором мы расскажем в рецепте 28 («Модульные таблицы стилей с помощью Sass»).

Кроме того, Skeleton включает удобные инструменты для создания вкладок (нечто похожее мы обсуждали в рецепте 7 («Навигация с помощью вкладок»)), а также средства для оформления полей HTML-форм.

Макет на основе Skeleton можно сделать гораздо быстрее, если заменить HTML на другой язык разметки — например, HAML[3]. HAML — это Ruby-библиотека, позволяющая создавать HTML-код, используя сокращенную запись. В таком случае наш код будет выглядеть примерно так:

cssgrids/index.haml

```
.container
  %header.sixteen.columns
    %h1 SpotFindr
    %h3 123 Fake Street, Anytown USA 12345
  %section.eight.columns
    %h2.price $109,900
    %p
      Simple single-family home on the north side, within walking...
  %section.eight.columns ...
```

[1] https://github.com/scottjehl/Respond
[2] http://compass-style.org/
[3] http://haml-lang.com/

В HAML нет закрывающих тегов (это достигается за счет отступов), а синтаксис классов такой же, как в CSS. После написания такого кода можно конвертировать HAML в обычный HTML. Такие библиотеки, как Skeleton, а также Sass (см. «Рецепт 28. Модульные таблицы стилей с помощью Sass») и HAML мы использовали в первую очередь ради быстроты создания и внедрения наших сайтов.

Смотрите также

- Рецепт 36. Как разместить статический сайт на Dropbox
- Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake
- Рецепт 28. Модульные таблицы стилей с помощью Sass

Рецепт 27. Простой блог с помощью Jekyll

Задача

Мы хотим создать блог, однако ресурсы нашего сервера ограниченны. У нас нет доступа к базе данных, и мы не можем запускать PHP-код. Это значит, что мы вынуждены отказаться от таких решений, как WordPress и Drupal. Тем не менее нам нужно создать блог, с которым было бы удобно работать, и для этого нам потребуется новое решение.

Ингредиенты

- Интерпретатор Ruby
- Библиотека Jekyll[1]

Решение

Поскольку применение баз данных в нашем блоге невозможно, мы воспользуемся *генератором статических сайтов*. Это удобный инструмент для создания статических сайтов на основе уже готового макета. Фреймворк Jekyll, в свою очередь, предназначен именно для блогов. Страницы и статьи в нем строятся на основе жесткой файловой структуры. Этот фреймворк включает простую и эффективную систему верстки и, не будучи предназначенным для среднего блогера, все же прекрасно подходит для создания пробных версий или простых блогов, так как позволяет избежать сложностей, связанных с использованием баз данных.

Чтобы продемонстрировать, как работает Jekyll, мы создадим музыкальный блог, в котором будем размещать понравившиеся нам мелодии.

Установка Jekyll

Для этого рецепта вам потребуется установить в операционной системе Ruby и Rubygems. Если при этом возникнут трудности, загляните в «Приложение 1. Установка Ruby». Для установки Jekyll необходимо выполнить следующую команду:

```
gem install jekyll
```

Так мы получаем исполняемый файл, который можно использовать для создания сайта.

Файловая структура

Для правильной работы с Jekyll нам необходима особая структура файлов и папок, а именно две папки для макетов и сообщений, начальная страница и файл настроек. Откройте программную оболочку и создайте в ней следующие файлы и папки.

- _layouts/
- _posts/
- index.html
- _config.yml

Последний файл предназначен для разного рода настроек, однако если оставить его пустым, сайт все равно будет выглядеть хорошо.

---

[1] http://jekyllrb.com/

Использование макетов

Мы начнем с создания начальной страницы, на которой приведем список последних записей блога. Поскольку Jekyll допускает «вложение» страниц, мы создадим наш макет с учетом этой возможности, чтобы нам не пришлось переписывать один и тот же HTML-код несколько раз. Теперь, чтобы внести изменения HTML-код нашего блога, достаточно будет только одного файла. В папке layouts мы создадим стандартный HTML-документ и назовем его base.html.

creatingablog/_layouts/base.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Music Blog</title>
  </head>
  <body>
    <header>
      <h1>My Music Blog</h1>
    </header>
    <section id="posts">
      {{ content }}
    </section>
  </body>
</html>
```

Для создания динамических страниц Jekyll использует язык шаблонов Liquid[1]. В нем теги шаблона заключаются в двойные фигурные скобки. Поэтому в нашем случае то, что мы захотим поместить в макет base.html — другой макет или запись блога, — будет помещено в область `{{content}}`. Позже нам понадобятся и другие теги шаблонов.

Имея готовую основу для макета, мы можем перейти к созданию остальной части начальной страницы. Контент мы зададим в файле index.html, который расположен в корне каталога. В языке шаблонов Liquid есть оператор цикла, который понадобится нам при создании разметки для каждой записи блога. Все наши записи будут отображаться в виде неупорядоченного списка.

creatingablog/index.html

```
---
layout: base
---

<ul>
  {% for post in site.posts %}
    <li>
      <!-- link to the post -->
    </li>
  {% endfor %}
</ul>
```

[1] http://liquidmarkup.org/

Первые три строки этого кода — специальный раздел в формате YAML, который всегда располагается перед основной разметкой и содержит метаданные, относящиеся к данной веб-странице, которые Jekyll будет учитывать при ее создании. YAML[1] представляет собой понятный человеку формат, предназначенный для хранения данных и совместимый с различными языками программирования; в этом смысле он очень похож на JSON. Начало и конец раздела, содержащего метаданные, обозначаются тремя дефисами. В данном случае этот раздел сообщает Jekyll, что в качестве макета нужно использовать созданный ранее файл base.html.

creatingablog/index.html

```
<li>
  <a href="{{ post.url }}">{{ post.title }}</a>
</li>
```

В операторе цикла используется тег шаблона `post`, который обозначает постоянную ссылку на запись блога.

Создание записи блога

Теперь на нашей странице можно размещать записи блога, но пока у нас нет ни одной такой записи. Jekyll позволяет создавать записи с помощью целого ряда языков разметки — например, Markdown, Textile и обычного HTML. Мы решили использовать простой и легко читаемый язык Markdown. При такой свободе выбора языка разметки мы должны строго соблюдать правила именования файлов записей: имя файла должно начинаться с даты, за которой следует название, причем слова должны отделяться друг от друга дефисами.

2011-08-12-my-first-post.markdown

Файлы записей располагаются в папке _posts. Создайте файл с таким названием, указав правильную дату.

Как и файл index.html, записи блога должны содержать начальный раздел в формате YAML. В нем мы укажем, какой макет мы хотим использовать, и зададим название записи блога. Специального макета для записи блога у нас пока нет — его созданием мы займемся чуть позже, — так что пока мы ограничимся нашим базовым макетом. Основной контент будет помещаться сразу после начального раздела.

```
---
layout: base
title: My First Post
---
Thank you for viewing my music blog! I plan to
write every day about how much I love and enjoy music.
```

Сборка сайта

Чтобы собрать сайт, нужно вызвать команду `jekyll`, которая поставляется вместе с нашим gem-файлом. Эта команда создает все необходимые статические файлы и помещает их в папку _site. Итак, в корневом каталоге нашего сайта мы запустим следующую команду:

```
$ jekyll
```

[1] http://yaml.org/

При создании блога удобно загружать файлы с сервера. Для этого в Jekyll существует специальный параметр `--server`:

```
$ jekyll --server
```

Эта команда выполняет сборку сайта и запускает сервер WEBrick, который будет работать через порт 4000. Теперь, чтобы посмотреть на наш сайт, достаточно открыть браузер и перейти на http://localhost:4000.

Наш блог представляет собой список записей, каждую из которых можно прочитать, щелкнув на соответствующей ссылке (рис. 5.4).

Рис. 5.4. Начальная страница

Чтобы отключить сервер, достаточно нажать Ctrl+C. Если мы захотим что-то исправить, нам придется снова выполнить сборку сайта и перезапустить сервер: только в таком случае изменения будут видны в браузере. Помните, что такой сервер может применяться только в процессе разработки; когда дело дойдет до внедрения сайта, мы будем использовать файлы из папки _site.

Макет для записи блога

Если вы откроете запись блога, пройдя по соответствующей ссылке, вы увидите, что кроме контента на экране не отображается никакой дополнительной информации. Давайте создадим в папке layouts файл post.html и с его помощью добавим к записи заголовок.

creatingablog/_layouts/post.html

```
---
layout: base
---

<article class="post">

  <h2>{{ post.title }}</h2>
  {{ content }}

</article>
```

Нам остается лишь немного подкорректировать нашу запись, заменив название шаблона на `post`. Теперь она будет выглядеть следующим образом:

creatingablog/_posts/2011-08-12-my-first-post.markdown

```
---
layout: post
title: My First Post
---

Thank you for viewing my music blog! I plan to write
every day about how much I love and enjoy music.
```

После того как мы еще раз выполним сборку сайта и перезапустим сервер, запись блога будет иметь заголовок.

Более изощренные макеты

Jekyll позволяет легко решать различные дизайнерские задачи. При создании макета можно использовать как таблицы стилей, так и изображения. Любые файлы и папки, находящиеся в корневом каталоге, становятся частью будущего сайта. Поэтому мы решили немного оживить наш сайт, добавив несколько таблиц стилей в виде внешнего файла. В корневом каталоге мы создадим папку css, а внутри нее — файл style.css.

Начнем с нескольких простых стилей.

creatingablog/css/styles.css

```
body {
  background: #f1f1f1;
  color: #111;
  font-size: 12px;
  font-family: Verdana, Arial, sans-serif;
}

ul {
  list-style: none;
}
```

Далее нам нужно будет изменить файл base.html, подключив только что созданную таблицу стилей.

creatingablog/_layouts/base.html

```
<link rel="stylesheet" href="/css/styles.css"
       type="text/css" media="screen" charset="utf-8">
```

ВОПРОС/ОТВЕТ. А МОЖНО НЕ ИСПОЛЬЗОВАТЬ НЕКОТОРЫЕ ФАЙЛЫ И ПАПКИ ИЗ КОРНЕВОГО КАТАЛОГА?

Конечно! Если вы предпочитаете хранить, к примеру, файлы Photoshop в той же папке, что и изображения, но не хотите, чтобы они загружались на сервер, об этом можно легко сообщить Jekyll с помощью файла _config.yml, который мы создали в начале рецепта.

Чтобы можно было указать, какие файлы не должны использоваться при создании сайта, существует специальный параметр `exclude`, после которого должен располагаться список игнорируемых файлов. Все это необходимо поместить в файл _config.yml.

```
exclude:
- images/psd/
- README
```

Более подробно о параметрах настройки Jekyll можно узнать на соответствующей странице[1].

[1] https://github.com/mojombo/jekyll/wiki/Configuration

После повторной сборки сайта папка css будет подключена; новое оформление можно будет увидеть, открыв страницу в браузере (рис. 5.5).

Рис. 5.5. Готовая страница

Таким же образом можно загружать изображения и JavaScript-файлы. Для этого нужно создать новую папку, например images или js, и добавить в разметку ссылку на файл из этой папки.

Статические страницы

С помощью Jekyll можно создавать не только блоги, но и обычные статическое вебстраницы. По своей структуре такие страницы очень похожи на записи блога: они могут иметь заголовки, использовать макеты и теги шаблонов.

Чтобы создать статическую страницу, мы воспользуемся уже готовым макетом для записи блога. Сначала нам нужно переименовать post.html в page.html, а затем изменить разметку так, чтобы в ней использовался тег шаблона page.

creatingablog/_layouts/page.html

```
---
layout: base
---

<section class="post">
  <h3>{{ page.title }}</h3>
  {{ content }}
</section>
```

На самом деле Jekyll считает страницей все что угодно, так что этот тег шаблона можно было бы использовать и для записей блога. Но поскольку мы переименовали файл макета, записи уже не будут работать правильно; чтобы это исправить, нужно заменить в них post на page.

После того как мы немного изменили макет, можно приступить к созданию статической страницы. Нам понадобится новый файл contact.markdown, который нужно поместить в корневой каталог. В начальном YAML-разделе этого файла мы укажем page в качестве шаблона, а также зададим название страницы.

creatingablog/contact.markdown

```
---
layout: page
title: Contact
---
```

продолжение ⇩

```
If you would like to get in contact with me,
send an email to
[johnsmith@test.com](mailto:johnsmith@test.com).
```

При создании статической страницы Jekyll ориентируется на имя Makedown-файла. В нашем случае это contact.makedown, поэтому новый файл будет называться contact.html. Давайте добавим ссылку на этот файл на начальной странице.

```
<a href="/contact.html">Contact Me</a>
```

Теперь у нас есть готовый блог, и мы можем разместить его на сервере. Для этого мы будем использовать содержимое папки _site, сгенерированное в результате запуска команды jekyll.

Дополнительные возможности

Если вы предпочитаете создавать блоги с помощью WordPress, Drupal или другого фреймворка, на странице Jekyll вы найдете информацию о том, как преобразовать ваши записи блога в форму, понятную Jekyll. Там также есть плагины, позволяющие добавлять подсветку синтаксических конструкций, облака тегов и многое другое. За более подробной информацией о возможностях Jekyll можно обратиться к GitHub[1].

Смотрите также

- Рецепт 36. Как разместить статический сайт на Dropbox
- Рецепт 26. Удобный дизайн с помощью сетки
- Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake
- Приложение 1. Установка Ruby

[1] https://github.com/mojombo/jekyll/wiki

Рецепт 28. Модульные таблицы стилей с помощью Sass

Задача

Яркие интерфейсы, удобные макеты, шрифтовое оформление — все это в значительной степени основано на таблицах стилей. Однако при всей своей эффективности этот инструмент обладает рядом серьезных недоработок. Пытаясь избежать копирования повторяющихся фрагментов кода, даже неопытные веб-разработчики чувствуют острый недостаток переменных и функций, и в результате им приходится обращаться к JavaScript и jQuery. Хотя CSS не обладает всеми необходимыми возможностями, существуют специальные инструменты для генерирования CSS-кода, в которых эти возможности реализованы. Одним из таких инструментов является Sass.

Ингредиенты

- Sass[1]

Решение

Sass — гибкий и удобный инструмент для создания таблиц стилей, расширяющий возможности CSS. Благодаря ему в нашем распоряжении оказывается то, чего мы так долго ждали: переменные и функции. При написании кода используется расширенный синтаксис CSS, после чего запускается прекомпилятор, который генерирует обычный CSS-код, понятный браузерам. По умолчанию синтаксис Sass поддерживает CSS3, поэтому для перехода в формат Sass достаточно просто изменить расширение CSS-файла, например переименовать style.css в style.scss.

При оформлении кнопок в рецепте 1 («Оформление кнопок и ссылок») и «облаков с текстом» в рецепте 2 («Оформление цитат с помощью CSS») нам достаточно часто приходилось копировать фрагменты кода. С помощью Sass мы выделим фрагменты кода, общие для всех кнопок и всех «облаков с текстом», а затем соберем из них одну большую таблицу стилей. В этом рецепте мы не будем рассказывать о том, как работает наш CSS-код; если это вызовет у вас трудности, обратитесь к предыдущим разделам книги.

Создание проекта Sass

Sass преобразует свои файлы в обычные CSS-файлы с помощью прекомпилятора. Это преобразование можно выполнять в специальном графическом интерфейсе, однако мы ограничимся версией для командной строки, написанной на Ruby. Чтобы установить прекомпилятор из командной строки, нужно написать следующее:

```
$ gem install sass
```

Поскольку в браузере будут работать только обычные CSS-файлы, мы будем хранить их отдельно от Sass-файлов. Для этого нам понадобятся две папки: sass и stylesheets.

```
$ mkdir sass
$ mkdir stylesheets
```

[1] http://sass-lang.com/

Инструмент командной строки `sass` может следить за изменениями в заданном каталоге и преобразовывать Sass-файлы в CSS-файлы. В данном случае изменения будут происходить в папке sass, а результат преобразования будет помещаться в папку stylesheets.

```
$ sass --watch sass:stylesheets
```

Это будет продолжаться до тех пор, пока мы не нажмем Ctrl+C или не перезапустим компьютер.

Итак, проект создан. Теперь мы познакомимся с самым простым, но невероятно полезным элементом Sass: переменной.

Переменные и импортирование

Для нашей кнопки мы задаем цвет фона и цвет границы. Работая с обычной таблицей стилей, мы часто используем одни и те же коды цветов HTML в разных ее частях. Понятно, что потом изменить эти значения будет не так уж просто. В языках программирования, таких как JavaScript, эта проблема решается с помощью переменных, однако в обычном CSS переменных нет. В Sass они есть, и к тому же их очень просто использовать.

Давайте создадим в папке sass файл style.scss. В верхней части этого файла определим две переменные: цвет фона и цвет границы.

sass/sass/style.scss

```
$button_background_color: #A69520;
$button_border_color: #282727;
```

В Sass перед названием переменной стоит знак доллара; значения переменных задаются точно так же, как и значения CSS-свойств.

Для удобства мы поместим определение стилей для кнопок в отдельный файл _buttons.css, который будет храниться в папке sass. Символ подчеркивания в названии файла сообщает Sass о том, что этот файл не является самостоятельной таблицей стилей и его не нужно преобразовывать в обычный CSS. Здесь мы зададим основное оформление кнопки, используя переменные для цвета границы и цвета фона.

sass/sass/_buttons.scss

```
.button {
  font-weight: bold;
  background-color: $button_background_color;
  text-transform: uppercase;
  font-family: verdana;
  border: 1px solid $button_border_color;
  font-size: 1.2em;
  line-height: 1.25em;
  padding: 6px 20px;
  cursor: pointer;
  color: #000;
  text-decoration: none;
}

input.button {
  line-height: 1.22em;
}
```

Затем мы импортируем этот Sass-файл в таблицу стилей style.scss с помощью оператора `@import`.

sass/sass/style.scss

```
@import "buttons.scss";
```

Во время обработки компилятор Sass увидит правило `@import` и вставит на его место содержимое другого файла; в итоге будет сгенерирован только один CSS-файл. Этот способ обеспечивает удобную организацию отдельных частей таблицы стилей на стадии разработки. Далее мы сделаем еще один шаг в этом направлении.

Совместное использование кода с помощью примесей[1]

При оформлении фона кнопок и «облаков с тестом» мы использовали градиент; точно так же оба элемента имеют закругленные уголки. При этом градиент кнопки меняется при наведении курсора мыши. Для всего этого требуется достаточно много CSS-кода, поскольку одно и то же свойство по-разному описывается в разных браузерах. К тому же код, отвечающий за добавление тени к кнопкам, было бы логично использовать и для других элементов, которые будут иметь тень.

Все эти правила можно определить как *примеси*, и тогда их можно будет использовать в любой части таблицы стилей. Все эти примеси мы зададим в файле _mixins.scss, а затем импортируем этот файл в style.scss.

sass/sass/style.scss

```
@import "mixins";
```

Давайте откроем файл _mixins.scss и начнем с примеси для закругленных уголков. Объявление примеси во многом похоже на объявление JavaScript-функции: параметры задаются в обычных скобках, контент — в фигурных.

sass/sass/_mixins.scss

```
@mixin rounded($radius){
          border-radius: $radius;
     -moz-border-radius: $radius;
  -webkit-border-radius: $radius;
}
```

После того как мы задали эту примесь, мы можем добавить ее к определению `.button` в файле _buttons.scss с помощью оператора `@include`.

sass/sass/_buttons.scss

```
@include rounded(12px);
```

После этого она становится обычным CSS-правилом.

Далее мы создадим примесь для градиента; здесь нам потребуется боле сложный код.

---

[1] Класс, участвующий во множественном наследовании и добавляющий некоторые свойства в производный класс.

sass/sass/_mixins.scss

```
@mixin gradient($color1, $color2, $alpha1: 100%, $alpha2: 100%){
  background:
    -webkit-gradient(linear, 0 0,
                     $alpha1, $alpha2,
                     from($color1), to($color2));
  background: -moz-linear-gradient($color1, $color2);
  background: -o-linear-gradient($color1, $color2);
  background: linear-gradient(top center, $color1, $color2);
}
```

Поскольку браузеры на движке WebKit (например, Google Chrome и Safari), а также многие браузеры мобильных устройств поддерживают альфа-прозрачность, мы решили добавить в примесь соответствующие параметры. В отличие от кнопок «облака с текстом» используют эти параметры, поэтому по умолчанию мы зададим для них значение 100 %. После этого можно добавить эту примесь в файл _buttons.css.

sass/sass/_buttons.scss

```
@include gradient(#FFF089, #A69520 );
```

Кроме того, этот код нам понадобится в тот момент, когда мы захотим изменить градиент при наведении курсора мыши. Давайте посмотрим, как Sass решает проблему псевдоклассов.

Как избежать копирования кода с помощью вложения

В такой ситуации, используя обычные CSS-селекторы, нам пришлось бы написать несколько однотипных фрагментов кода. Два типа оформления гиперссылки — в нейтральном состоянии и при наведении — мы, скорее всего, описали бы следующим образом:

```
a{
  color: #300;
}
a:hover{
  color: #900;
}
```

Sass позволяет помещать определение псевдокласса *внутри* родительского правила.

```
a{
  color: #300;
  &:hover{
    color: #900;
  }
}
```

Хотя такой способ записи не является более экономичным (с точки зрения количества символов), он позволяет сделать код более структурированным.

В _buttons.scss метод вложения удобно использовать при добавлении примеси для псевдокласса `hover`.

sass/sass/_buttons.scss

```
&:active, &:focus {
  @include gradient(#A69520, #FFF089 );
  color: #000;
}
```

При создании более сложной таблицы стилей вложение позволяет избежать повторения одних и тех же селекторов. Рассмотрим обычный пример.

```
#sidebar a{
  color: #300;
}
#sidebar a:hover{
  color: #900;
}
```

Используя метод вложения, мы можем переписать его в следующем виде:

```
#sidebar a{
  color: #300;
  &:hover{
    color: #900;
  }
}
```

Таким образом мы можем объединить в одну группу все правила, относящиеся к данному селектору, и при этом нам достаточно будет описать иерархию селекторов только один раз.

Далее, используя наши примеси, мы создадим оформление «облаков с текстом» в файле _speech_bubbles.scss.

sass/sass/_speech_bubble.scss

```
 1  blockquote {
 -    width: 225px;
 -    padding: 15px 30px;
 -    margin: 0;
 5    position: relative;
 -    background: #faa;
 -    @include gradient(#c40606, #ffaaaa, 20%, 100%);
 -    @include rounded(20px);
 -    p {
10      font-size: 1.8em;
 -      margin: 5px;
 -      z-index: 10;
 -      position: relative;
 -    }
15    + cite {
 -      font-size: 1.1em;
 -      display: block;
 -      margin: 1em 0 0 4em;
```

продолжение ⇩

```
 -    }
20    &:after {
 -      content: "";
 -      position: absolute;
 -      z-index: 1;
 -      bottom: -50px;
25      left: 40px;
 -      border-width: 0 15px 50px 0px;
 -      border-style: solid;
 -      border-color: transparent #faa;
 -      display: block;
30      width: 0;
 -    }
 -  }
```

Здесь вызов примесей начинается со строки 7, а вложение используется в строке 15. Теперь нужно импортировать этот файл в style.scss.

```
sass/sass/style.scss
@import "speech_bubble.scss";
```

Перед тем как закончить этот раздел, мы познакомимся с еще одной потрясающей возможностью Sass: оператором цикла.

Создание таблиц стилей с помощью оператора цикла

Если мы вернемся к первоначальной таблице стилей для кнопки, мы увидим, что в новом варианте оформления не хватает одной детали: тени.

Как и в случае с закругленными уголками, для каждого типа браузеров нам потребуется отдельное объявление. Однако теперь, вместо того чтобы писать весь необходимый код вручную, мы воспользуемся оператором цикла. Давайте добавим в файл _mixins.scss следующий код.

sass/sass/_mixins.scss

```
@mixin shadow($x, $y, $offset, $color){
  @each $prefix in "", -moz-, -webkit-, -o-, -khtml- {
    #{$prefix}box-shadow: $x $y $offset $color;
  }
}
```

Этот код позволяет перебрать все префиксы браузеров и для каждого из них сгенерировать соответствующее CSS-свойство. Список префиксов начинается с пустой строки, поскольку вариант box-shadow предусмотрен в спецификации CSS.

Теперь добавим эту примесь в _buttons.scss, и у кнопки появится тень.

sass/sass/_buttons.scss

```
@include shadow(1px, 3px, 5px, #555);
```

Все это время, пока мы один за другим добавляли разные элементы оформления, команда sass собирала их в одну большую таблицу стилей. В результате мы получили файл style.css, который и будем использовать в качестве таблицы стилей для нашей веб-

страницы; Sass-файлы можно не удалять, а оставить в репозитории вместе с исходным кодом.

Дополнительные возможности

С новыми возможностями Sass трудно представить себе процесс создания таблиц стилей, устроенный как-то иначе. В этом рецепте в нашем распоряжении оказался совсем небольшой CSS-код, но мы уже можем представить, насколько проще станет процесс создания и редактирования таблиц стилей для крупной системы управления контентом! Вы сможете создать свою собственную библиотеку примесей и обращаться к ней при оформлении различных элементов страницы, а с помощью переменных вы сможете легко изменять численные значения, коды цветов и названия шрифтов.

Но на Sass история не заканчивается. На его основе был построен CSS-фреймворк Compass со встроенной библиотекой примесей и плагинов, предназначенных для создания сеток и реализации возможностей CSS3[1].

СКАЗКА О ДВУХ СИНТАКСИСАХ

В Sass возможны два разных синтаксиса: синтаксис SCSS, который мы использовали в этом рецепте, и еще один вариант синтаксиса, который иногда называют «Sass с отступами» («Indented Sass»), или «классический Sass» («Sass Classic»). Второй вариант использует вместо фигурных скобок отступы; он предназначен для разработчиков, делающих выбор в пользу краткости — в ущерб сходству с обычным синтаксисом CSS. Он также не требует точек с запятой в конце объявлений. Ниже приведен пример использования альтернативного синтаксиса в таблице стилей Sass, задающей цвета ссылок для двух разделов страницы: `sidebar` и `main`.

```
#sidebar
  a
    color: #f00
  &:hover
    color: #000
#main
  a
    color: #000
```

Если вы решите использовать такой синтаксис, вам придется изменить расширение .scss на .sass, но на конечный результат это никак не повлияет. Оба варианта синтаксиса взаимозаменяемы, и в дальнейшем они будут так же широко поддерживаться. Так что выбор за вами.

Смотрите также

- Рецепт 1. Оформление кнопок и ссылок
- Рецепт 2. Оформление цитат с помощью CSS
- Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript
- Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake
- Практическое руководство по Sass (Pragmatic Guide to Sass) [CC11]

[1] http://compass-style.org/

Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript

Задача

JavaScript является общепринятым языком программирования Глобальной сети. Однако неумелое обращение с ним зачастую приводит к тому, что код выглядит небрежным и работает не так, как нужно. Иногда его правила и синтаксис попросту сбивают разработчиков с толку, что, конечно же, сказывается на производительности. Но поскольку JavaScript используется абсолютно везде, мы не можем просто заменить его на язык с более удобным синтаксисом. Но мы можем использовать другой язык, чтобы *сгенерировать* корректный и хорошо работающий JavaScript-код.

Ингредиенты

- CoffeeScript[1]
- Guard[2] и дополнение к CoffeeScript[3]
- QEDServer

Решение

С помощью CoffeeScript можно писать JavaScript-код в более сжатом формате, похожем на Ruby и Python. Далее этот код передается интерпретатору, преобразующему его в обычный JavaScript-код, который можно использовать при создании веб-страниц. При такой организации рабочего процесса выигрыш в производительности, как правило, оказывается существеннее затрат.

Например, мы уже не можем случайно забыть поставить двоеточие или закрывающую фигурную скобку, и мы не ошибемся при определении области видимости переменной. Таким образом, CoffeeScript снимает с нас ответственность за мелкие детали, чтобы мы могли полностью сконцентрироваться на решении основной задачи.

Чтобы понять, как работает CoffeeScript, мы попробуем использовать его вместе с jQuery для получения информации о товарах из API нашего магазина. Нам также понадобится сервер для тестирования, к которому мы уже обращались в рецепте 14 («Как структурировать код с помощью Backbone.js»).

CoffeeScript представляет собой самостоятельный язык программирования. Это значит, что для объявления переменных и функций, а также для выполнения ряда других действий нам придется использовать совершенно новый синтаксис. Несмотря на то что на сайте CoffeeScript и в книге Тревора Бернема «CoffeeScript: как ускорить написание JavaScript-кода» (Trevor Burnham «CoffeeScript: Accelerated JavaScript Development») [Bur11] можно найти описание этого языка в мельчайших подробностях, этот раздел мы все же начнем с рассмотрения нескольких особенностей CoffeeScript.

---

[1] http://coffeescript.org/

[2] https://github.com/guard/guard

[3] https://github.com/netzpirat/guard-coffeescript

Общие сведения о CoffeeScript

Синтаксис CoffeeScript создавался по аналогии с синтаксисом JavaScript с учетом стремления разработчиков избавиться от максимального числа ненужных символов. Для примера попробуем переписать следующее объявление функции.

```
var hello = function(){
  alert("Hello World");
}
```

В результате мы получим такой код:

```
hello = -> alert "Hello World"
```

Чтобы объявить переменную, нам больше не нужно использовать ключевое слово `var`. CoffeeScript поймет, в какой момент нужно создать переменную, и сам добавит объявление с `var` в нужное место.

При объявлении функций вместо ключевого слова `function` CoffeeScript использует символ `->`, за которым следует тело функции, не заключенное в фигурные скобки. Если тело функции занимает более одной строки, мы добавляем отступ.

```
hello = (name) ->
  alert "Hello " + name
```

При том что в CoffeeScript существует множество еще более интересных и эффективных возможностей, использование только этих двух способно преобразить следующее объявление:

```
$(function() {
  var url;
  url = "/products.json";
  $.ajax(url, {
    dataType: "json",
    success: function(data, status, XHR) {
      alert("It worked!");
    }
  });
});
```

В результате мы получаем такой код:

```
$ ->
  url = "/products.json"
  $.ajax url,
    dataType: "json"
    success: (data, status, XHR) ->
      alert "It worked!"
```

Внешне вариант с использованием CoffeeScript выглядит немного проще; к тому же на написание такого кода уйдет меньше времени. А если мы совершим синтаксическую ошибку, мы узнаем об этом уже на этапе преобразования CoffeeScript в JavaScript, и нам не придется судить о правильности синтаксиса по картинке в окне браузера.

Установка CoffeeScript

Существует множество способов запустить CoffeeScript. Если нам нужно только протестировать код, проще всего сделать это через браузер: в таком случае нам не придется ничего устанавливать. Достаточно загрузить интерпретатор CoffeeScript[1] и подключить его к веб-странице.

coffeescript/browser/index.html

```
<script
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
</script>
<script src="coffee-script.js"></script>
```

После этого мы добавим еще один блок `<script>`, в который и поместим наш CoffeeScript-код.

coffeescript/browser/index.html

```
<script type="text/coffeescript">
$ ->
  url = "/products.json"
  $.ajax url,
    dataType: "json"
    success: (data, status, XHR) ->
      alert "It worked!"
</script>
```

Изначально браузер не знает, что делать с элементами `<script>`, для которых указан тип `text/coffeescript`, поэтому он просто их игнорирует. Но как только мы добавляем на страницу интерпретатор CoffeeScript, он сразу же находит все такие элементы и преобразует их содержимое. Далее он добавляет на страницу получившийся JavaScript-код, который затем выполняется в браузере. Поскольку интерпретатор CoffeeScript написан на CoffeeScript, он также проходит компиляцию.

Компиляция в браузере имеет смысл только в том случае, если вы хотите проверить, как работает CoffeeScript. Однако для действующего сайта такой метод не подходит, поскольку запуск интерпретатора CoffeeScript на машине клиента занимает слишком много времени, так как это требует загрузки файла большого объема. Преобразование CoffeeScript-файлов необходимо выполнять заранее, так чтобы на сайте использовались только готовые JavaScript-файлы. Для этого нужно установить интерпретатор CoffeeScript, но сначала мы должны решить, как лучше это сделать.

Обычно для установки CoffeeScript используется Node.JS и менеджер пакетов NPM (Node Package Manager)[2], однако можно сделать это и с помощью Ruby. Поскольку нам уже не раз приходилось обращаться к Ruby в других рецептах, мы выберем именно этот вариант. Если Ruby уже установлен на вашем компьютере (в противном случае см. Приложение 1 «Установка Ruby».), наберите в командной строке следующее:

```
$ gem install coffee-script guard guard-coffeescript
```

[1] http://jashkenas.github.com/coffee-script/extras/coffee-script.js

[2] http://npmjs.org/

Эта команда установит интерпретатор CoffeeScript и Guard. С их помощью мы сможем сделать так, чтобы по мере внесения нами изменений CoffeeScript-файлы автоматически конвертировались в JavaScript-файлы. Это преобразование будет выполнять `guard-coffeescript`.

Теперь давайте создадим свой проект и посмотрим, как все это работает.

Как работать с CoffeeScript

В качестве сервера разработки мы будем использовать QEDServer и его API. Все наши файлы мы поместим в папку public, чтобы сервер разработки знал, где они находятся; кроме того, так мы сможем избежать проблем с запросами Ajax.

Так как мы собираемся преобразовывать CoffeeScript-файлы в JavaScript-файлы, нам понадобятся две соответствующие папки.

```
$ mkdir coffeescripts
$ mkdir javascripts
```

Теперь давайте создадим очень простую веб-страницу и подключим к ней jQuery, библиотеку Mustache (см. Рецепт 10 «HTML с помощью Mustache») и файл app.js, в котором будет содержаться код, отвечающий за загрузку данных и их отображение на странице.

coffeescript/guard/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
    </script>
    <script src="javascripts/mustache.js"></script>
    <script src="javascripts/app.js"></script>
  </head>
  <body>
  </body>
</html>
```

Файл mustache.js нужно поместить в папку javascripts, а файл app.js появится в этой папке чуть позже.

Далее мы добавим простой шаблон Mustache, с помощью которого на странице будет отображаться каждый отдельный товар.

coffeescript/guard/index.html

```
<script id="product_template" type="text/html">
  <div class="product">
    {{#products}}
    <h3>{{name}}</h3>
    <p>{{description}}</p>
    {{/products}}
  </div>
</script>
```

Далее мы создадим файл coffeescripts/app.coffee.

coffeescript/guard/coffeescripts/app.coffee

```
$ ->
  $.ajax "/products.json",
    type: "GET"
    dataType: "json"
    success: (data, status, XHR) ->
      html = Mustache.to_html $("#product_template").html(), {products: data}
      $('body').append html
    error: (XHR, status, errorThrown) ->
      $('body').append "AJAX Error: #{status}"
```

Чтобы обозначить код, который будет запускаться после того, как документ будет готов, мы используем запись `$ ->` — аналог не очень удобного сокращения jQuery `$(function(){})`. Мы создаем переменную для URL и вызываем метод jQuery `AJAX()`, который загружает шаблон Mustache в случае получения ответа на запрос и отображает сообщение об ошибке в противном случае. Логика и порядок операций здесь точно такие же, как и в обычном JavaScript-файле, однако этот код все же на несколько строк короче. Конечно же, в таком виде наша страница работать не будет, так как мы еще не сгенерировали JavaScript-код. Для этого нам понадобится Guard.

Преобразование CoffeeScript с помощью Guard

Guard — это инструмент командной строки, который следит за заданными файлами и в случае каких-либо изменений выполняет определенные действия. С помощью плагина `guard-coffeescript` Guard может преобразовать наши CoffeeScript-файлы.

Нам нужно, чтобы Guard следил за изменениями в папке coffeescripts и преобразовывал файлы из этой папки в JavaScript-формат, помещая новые файлы в папку javascripts. Для этого в корневом каталоге нашего проекта мы создадим файл Guardfile с описанием тех действий, которые необходимо выполнить с файлами CoffeeScript. Этот файл можно создать как вручную, так и из командной строки, используя следующие команды:

```
$ guard init coffeescript
```

После этого мы откроем файл Guardfile и изменим в нем названия папок, задав соответствующие значения `input` и `output`.

coffeescript/guard/Guardfile

```
# A sample Guardfile
# More info at https://github.com/guard/guard#readme
guard 'coffeescript', :input => 'coffeescripts', :output => "javascripts"
```

Далее запустим Guard из программной оболочки. С этого момента он начнет следить за изменениями в папке coffeescripts.

```
$ guard
Guard is now watching at '/home/webdev/coffeescript/public/'
```

Как только мы сохраним файл coffeescripts/app.coffee, Guard сразу же заметит это и преобразует наш файл в JavaScript-формат.

```
Compile coffeescripts/app.coffee
Successfully generated javascripts/app.js
```

Наконец-то наша страница работает! Чтобы в этом убедиться, откройте http://localhost:8080/index.html. А если вы заглянете в файл app.js, то увидите, что все обычные и фигурные скобки, а также точки с запятой стоят на своих местах. Таким образом мы доказали, что этот метод позволяет создавать действительно хороший JavaScript-код, и теперь можем продолжить работу над приложением по такой же схеме. Когда оно будет готово, все необходимые файлы будут содержаться в папке javascripts, а папку coffeescripts можно будет сохранить в репозитории вместе с исходным кодом.

Дополнительные возможности

Сейчас все больше и больше JavaScript-проектов переходят на CoffeeScript. Причина этого заключается не только в простоте написания кода, но и в том, что в CoffeeScript реализован ряд нетривиальных возможностей, которые раньше были доступны только в таких языках, как Ruby, — например, сворачивание списков и интерполяция строк. В JavaScript конкатенация строк выглядит так.

```
var fullName = firstName + " " + lastName;
```

CoffeeScript позволяет переписать этот код в следующем виде.

```
fullname = "#{firstName} #{lastName}"
```

Выражения, помещенные внутрь `#{}`, при необходимости преобразуются в строковый формат.

При работе с массивами и списками элементов нам часто хочется переписать следующий код в более простом виде.

```
var colors = ["red", "green", "blue"];
for (i = 0, length = colors.length; i < length; i++) {
  var color = colors[i];
  alert(color);
}
```

```
alert color for color in ["red", "green", "blue"]
```

Сократить объем кода можно и с помощью различных JavaScript-библиотек. Однако на самом деле подключение библиотек означает, что пользователю придется отдельно загружать тот код, который мы не написали. Используя CoffeeScript, мы получаем JavaScript-код, который будет работать везде, где есть JavaScript, и не требует подключения библиотек.

Помимо CoffeeScript в Guard также есть поддержка Sass (см. «Рецепт 28. Модульные таблицы стилей с помощью Sass»); она осуществляется с помощью `guard-sass`. Представьте себе, насколько эффективнее станет рабочий процесс, если использовать одновременно Guard, Sass и CoffeeScript! Для тех, кому по душе такая идея, существуют специальные инструменты: например, MiddleMan, с помощью которого можно легко создавать статические сайты на основе Sass и CoffeeScript[1]. А если добавить сюда стратегию автоматического внедрения (см., например, «Рецепт 42. Автоматизирован-

[1] http://middlemanapp.com/guides/getting-started

ное внедрение статического сайта с помощью Jammit и Rake»), мы сможем получить интереснейший опыт работы по новой эффективной схеме.

Смотрите также

- CoffeeScript: как ускорить написание JavaScript-кода (CoffeeScript: Accelerated JavaScript Development) [Bur11]
- Рецепт 28. Модульные таблицы стилей с помощью Sass
- Рецепт 14. Как структурировать код с помощью Backbone.js
- Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake

Рецепт 30. Управление файлами с помощью Git

Задача

Как и большинство веб-разработчиков, мы часто оказываемся в таких ситуациях, когда нам нужно создавать одновременно несколько версий кода. Иногда нам хочется поэкспериментировать с новым плагином, и сначала все складывается как нельзя лучше, но потом мы неожиданно заходим в тупик, пытаясь исправить серьезную ошибку. При этом мы всегда пользуемся управлением версий в том или ином виде — даже если просто сохраняем копии рабочего файла. Однако найти нужный файл среди беспорядочного множества других файлов бывает не так уж просто. Поэтому мы хотим найти современный, быстрый и надежный способ управления версиями кода, который обеспечивал бы возможность совместного доступа.

Ингредиенты

- Git[1]

Решение

Сегодня существует много различных систем управления версиями. Система Git пользуется большой популярностью среди веб-разработчиков, так как она является локальной и работает быстрее, чем обычное сохранение копий файла. Git позволяет работать параллельно над несколькими версиями кода; при этом мы можем сохранять изменения сколь угодно часто, создавая множество точек восстановления. Все эти возможности не оставляют разработчиков равнодушными, поэтому многие современные проекты с открытым кодом выбирают в качестве системы управления версиями именно Git.

На утреннем совещании директор повернулся к нам и сказал: «Помните те два макета, которые вы представляли на прошлой неделе? Я хочу, чтобы вы сделали две версии нашего сайта на их основе. Да, кстати, в основной версии сайта нужно тоже кое-что исправить».

Теперь нам придется работать одновременно над тремя версиями сайта. Для удобного хранения файлов и четкой синхронизации мы воспользуемся системой Git.

Настройка Git

Начнем с установки Git. Откройте сайт Git и загрузите пакеты, соответствующие вашей операционной системе[2]. Если вы работаете в Windows, выберите MsysGit[3], а также загрузите Git Bash — аналог командной строки, без которого вы не сможете выполнить операции, описанные в этом разделе.

Система Git определяет автора изменений по специально заданному имени пользователя Git. Благодаря этому мы можем легко узнать, кто и когда вносил изменения в код. Поэтому нам нужно сообщить Git свое имя и адрес электронной почты. Откройте программную оболочку и введите следующее:

---

[1] http://git-scm.com/

[2] http://git-scm.com/

[3] http://code.google.com/p/msysgit/

```
$ git config --global user.name "Firstname Lastname"
$ git config --global user.email "your_email@youremail.com"
```

После установки настройки Git мы можем перейти к обсуждению ключевых идей этой системы.

Основные принципы работы Git

Сначала мы попробуем превратить наш проект в репозиторий Git. Давайте создадим папку под названием git_site и обозначим ее как репозиторий Git. В командной строке (или в Git Bash, если вы работаете в системе Windows) наберите следующие команды:

```
$ mkdir git_site
$ cd git_site
$ git init
```

После инициализации каталога мы получим сообщение о подтверждении:

```
Initialized empty Git repository in /Users/webdev/Sites/git_site/.git/
```

Эти команды создают в корне нашего каталога скрытую папку под названием .git. Вся история изменений и другие детали, касающиеся нашего репозитория, будут помещаться в эту папку. Система Git будет регистрировать изменения в нашей папке и сохранять «снимки» кода, но сначала нужно сообщить системе, за изменением каких файлов она должна следить.

Давайте скопируем все файлы, относящиеся к нашему сайту, в папку git_site. Эти файлы хранятся в папке git вместе с остальными исходными кодами данной книги.

После этого добавим эти файлы в репозиторий, чтобы, в крайнем случае, мы всегда могли вернуться к тому, с чего начали. Для этого нужно выполнить следующую команду.

```
$ git add .
```

Команда add не выводит никакой информации; для этого нам придется воспользоваться командой `git status`. Эту команду можно выполнять в любой момент, когда вам необходимо узнать текущее состояние репозитория Git.

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "Git rm --cached <file>..." to unstage)
#
#       new file: index.html
#       new file: javascripts/application.js
#       new file: styles/site.css
#
```

Это называется «подготовкой файлов к регистрации». Таким образом мы можем узнать, какие изменения Git собирается зафиксировать, и, если нужно, отменить некоторые из них перед передачей файлов в репозиторий. «Подготовка файлов к регистрации»

означает, что Git готов следить за их изменениями. Сейчас все в порядке, и мы можем зарегистрировать добавление этих файлов.

```
$ git commit -a -m "initial commit of files"
```

Здесь мы использовали два флага: `-a` и `-m`. С помощью первого мы сообщаем Git о том, что мы хотим добавить все изменения в индекс до передачи; второй флаг позволяет задать сообщение. В отличие от других систем управления версиями в Git каждый коммит должен сопровождаться сообщением. Не следует относиться к таким сообщениям с пренебрежением: с их помощью гораздо проще отслеживать коммиты. После завершения передачи мы получим подтверждение с описанием выполненных действий:

```
                              [master (root-commit) 94c75a2] Initial Commit
1 files changed, 17 insertions(+), 0 deletions(-)
create mode 100644 index.html
create mode 100644 javascripts/application.js
create mode 100644 styles/site.css
```

Чтобы убедиться в том, что передача файлов действительно была произведена, можно запустить `git status`. Если мы это сделаем, то увидим, что все в порядке.

```
# On branch master
nothing to commit (working directory clean)
```

Теперь у нас есть «снимок» кода. Это значит, что мы можем вносить изменения в файлы и одновременно их регистрировать.

Работа с ветвями

Ветвление позволяет работать независимо над разными частями сайта. С его помощью мы можем заниматься разработкой какого-то отдельного компонента, сохраняя основной код в рабочем состоянии. В отличие от других систем управления версиями использовать ветвление в Git гораздо проще, поэтому оно применяется очень широко.

Директор попросил нас разработать версии сайта на основе двух макетов, которые мы назовем *layout_a* и *layout_b*. Давайте создадим отдельную ветвь для *layout_a*.

```
$ git checkout -b layout_a
Switched to a new branch 'layout_a'
```

Если мы сейчас вызовем `git status`, мы увидим, что наша текущая ветвь — *layout_a*. Давайте откроем файл index.html и заменим текст в теге `<h1>` на `"Layout A"`, после чего сохраним файл. Теперь мы снова вызовем `git status`. На экране появится следующее:

```
# On branch layout_a
# Changed but not updated:
#   (use "Git add <file>..." to update what will be committed)
#   (use "Git checkout -- <file>..." to discard changes in working directory)
#
#       modified: index.html
#
no changes added to commit (use "Git add" and/or "Git commit -a")
```

Далее мы зафиксируем изменения для ветви *layout_a*.

```
$ git commit -a -m "changed heading to Layout A"
```

Пока мы занимались разработкой этой ветви, директор прислал нам следующее письмо: «На начальной странице сказано, что мы доставляем товары за один день. Но эта акция больше не действует. Срочно замените время доставки на два дня, пока никто ничего не заказал по старой акции!» Давайте вернемся к ветви *master* и выполним просьбу директора.

```
$ git checkout master
```

Если мы сейчас откроем index.html, мы не найдем там текст, который был изменен в ветви *layout_a*. Причина в том, что эти изменения были произведены в другой ветви. Git выполняет за нас всю работу, меняя содержимое файла при изменении ветви. Теперь мы можем сделать то, о чем нас попросил директор, и затем зафиксировать изменения в ветви *master*.

```
$ git commit -a -m "fixed shipping promotion from one day to two-day"
[master d00d2de] fixed shipping promotion from one day to two-day
1 files changed, 1 insertions(+), 1 deletions(-)
```

Это изменение было сделано в ветви *master*, однако если мы теперь переключимся на другую ветвь, мы его там не увидим. Если бы мы потеряли часть работы, это было бы по-настоящему страшно. Поэтому давайте добавим наши изменения в ветвь *layout_a* и вернемся к работе над этим макетом.

```
$ git checkout layout_a
$ git merge master
```

Эта команда находит все, что было изменено в ветви *master* и не было изменено в ветви *layout_a*, и добавляет к ветви *layout_a*.

Далее мы добавим еще одну ветвь для макета *layout_b*. Эта ветвь должна «отпочковываться» от основного сайта, а не от его версии *layout_a*, поэтому нам нужно снова переключиться на ветвь *master*, и только после этого можно будет создавать новую ветвь.

```
$ git checkout master
$ git checkout -b layout_b
```

На этот раз мы заменим текст в теге `<h1>` на `"Layout B"`. После этого мы сохраним и зафиксируем это изменение.

```
$ git commit -a -m "Changed heading to Layout B"
```

В этой версии макета нам потребуются еще два файла: products.html и about_us.html. Давайте создадим эти файлы и подготовим их к регистрации изменений.

```
$ touch products.html
$ touch about_us.html
$ git add .
```

Если мы запустим `git status`, то увидим, что в списке появились два новых файла.

```
# On branch layout_b
# Changes to be committed:
#   (use "Git reset HEAD <file>..." to unstage)
```

```
#
#       new file: about_us.html
#       new file: products.html
#
```

Теперь зафиксируем изменения.

```
$ git commit -a -m "added products and about_us, no content"
```

Для полноты картины добавим в файл products.html элемент `<h1>` с текстом `"Current Products"`.

Пока мы занимались вторым макетом, пришло еще одно письмо от директора: «На начальной странице нужно еще раз изменить время доставки товаров. Мы заключили выгодную сделку с крупной транспортной компанией и теперь снова доставляем все за один день. Обновите информацию как можно скорее!» Теперь нам нужно срочно изменить начальную страницу. Но пока мы не можем отправить коммит с текущими изменениями.

Для таких ситуаций в Git предусмотрена специальная команда `stash`: она позволяет сохранять внесенные изменения, переключаясь между ветвями. Это своего рода «тайник», в котором можно хранить изменения, не регистрируя их.

```
$ git stash
```

Если выполнить команду `git status`, можно увидеть, что на данный момент в списке нет изменений, готовых к регистрации. Теперь давайте переключимся на ветвь *master*.

```
$ git checkout master
```

После этого мы откроем файл index.html и обновим информацию о доставке товаров, а затем зафиксируем внесенные изменения.

```
$ git commit -a -m"updated shipping times"
```

Теперь давайте вернемся к ветви *layout_b*, выполнив `git checkout layout_b`, и посмотрим, что можно сделать с нашим «тайником». Для начала заглянем в список доступных «тайников» с помощью команды `git stash list`.

```
$ git stash list
stash@{0}: WIP on layout_b: f8747f4 added products and about_us, no content
```

Если открыть файл products.html, можно увидеть, что он пуст. Однако наши изменения можно легко восстановить с помощью следующей команды:

```
$ git stash pop
```

Теперь в файле products.html снова появится тег `<h1>`, который мы добавили до перехода на другую ветвь.

ВОПРОС/ОТВЕТ. ПОЧЕМУ МЫ ТАК ЧАСТО РЕГИСТРИРУЕМ ИЗМЕНЕНИЯ?

Представьте себе, что коммиты — это моментальные снимки, или точки восстановления, вашего проекта. Тогда чем больше коммитов вы отправите, тем эффективнее и гибче будет работать Git. Если коммит относится к конкретному свойству или элементу, мы можем применить его к другой ветви проекта, используя `cherry-pick`. Если вы предпочитаете более крупные коммиты, можно объединять их с помощью команды `rebase`.

Добавив еще несколько изменений в оба макета (а также выполнив пару «чрезвычайно важных» указаний директора), мы получили сообщение о том, что для внедрения сайта был выбран макет *layout_b*. Поэтому теперь нам нужно объединить эту ветвь с ветвью *master*.

```
$ git checkout master
$ git merge layout_b
$ git commit -a -m "merged in layout_b"
```

В традиционных системах управления версиями ветви принято всегда оставлять в репозитории. Git отличается от этих систем тем, что в нем и ветви, и теги содержат ссылку на коммит. При удалении ветви Git удаляет только саму ссылку, а коммиты при этом остаются на месте. Поэтому после добавления наших изменений к ветви *master* мы можем смело удалять ненужные ветви. Но прежде чем это сделать, давайте посмотрим на список ветвей. Выполнив команду `git branch`, мы увидим, что сейчас мы находимся в ветви *master* и что кроме этой ветви существуют еще две: *layout_a* и *layout_b*. Именно их мы и удалим.

```
$ git branch -d layout_a
$ git branch -d layout_b
```

Если какая-то ветвь не была объединена с текущей ветвью, Git сообщит нам об этом. Чтобы удалить ветвь в любом случае, можно использовать параметр `-D`.

Работа с удаленными репозиториями

Все это время мы работали с локальным репозиторием. И хотя локальное управление версиями, безусловно, имеет огромное значение, подключение удаленных репозиториев позволяет работать над проектом вместе с другими разработчиками, сохраняя код одновременно в нескольких местах.

Мы начнем с настройки удаленного Git-сервера на виртуальной машине, которую мы создали специально для разработки проекта (см. «Рецепт 37. Как установить виртуальную машину»). С помощью SSH-ключей можно избавить себя от необходимости вводить пароль каждый раз при входе в систему или при передаче файлов. Создание такого ключа на сервере позволяет выполнять проверку прав доступа к удаленному репозиторию быстро и без использования паролей.

SSH-ключи состоят из двух частей: приватного ключа, который мы оставляем у себя, и публичного ключа, который мы передаем на сервер. Когда мы заходим на этот сервер, он проверяет, авторизован ли наш ключ. После этого наша локальная система сравнивает публичный ключ с приватным, таким образом подтверждая, что мы — именно те, за кого себя выдаем. В случае с Git весь этот процесс происходит открыто при входе в систему.

Прежде чем двигаться дальше, проверьте, есть ли в вашей системе SSH-ключи. Попробуйте перейти в каталог ~/.ssh. Если на экране появится сообщение о том, что такого каталога не существует, значит, вам нужно генерировать ключи. Если же вы увидите что-то вроде id_rsa или id_rsa.pub, значит, у вас уже есть ключи и вы можете пропустить следующее действие.

Чтобы сгенерировать новый SSH-ключ, мы воспользуемся командой `ssh-keygen`. В качестве комментария мы передадим ей свой адрес электронной почты:

```
$ ssh-keygen -t rsa -C "webdev@awesomeco.com"
```

Во время загрузки ключа на сервер эта информация поможет нам или другим администраторам сервера быстро определить владельца ключа.

Программа для генерирования SSH-ключей спросит, где вы хотите хранить ключи; здесь можно просто нажать клавишу Enter и таким образом выбрать папку, заданную по умолчанию. Далее вам нужно будет ввести кодовую фразу в качестве дополнительного уровня безопасности. Пока мы оставим это поле пустым, так что просто нажмите клавишу Enter.

Готовые ключи мы добавим в виртуальную машину. Наш локальный публичный ключ можно передать на сервер в файл authorized_keys: так мы сообщим виртуальной машине, что у нашего компьютера есть доступ к серверу.

```
$ cat ~/.ssh/id_rsa.pub | ssh webdev@192.168.1.100 \
"mkdir ~/.ssh; cat >> ~/.ssh/authorized_keys"
```

Выполнив эту команду, сервер попросит нас ввести пароль; это нужно для подтверждения легальности запроса. После этого мы можем проверить, как работает новый SSH-ключ, обратившись по нему к виртуальной машине.

```
$ ssh webdev@192.168.1.100
```

На этот раз нам не нужно будет вводить пароль.

После успешного входа на сервер мы попробуем установить на нем Git, используя менеджер пакетов Ubuntu.

```
$ sudo apt-get install git-core
```

Теперь мы создадим на виртуальной машине пустой репозиторий, который выглядит как обычный каталог. Для него удобно использовать расширение .git, по которому можно легко находить нужные файлы. Далее внутри этого каталога мы запустим команду `git` с параметром `--bare`, которая выполнит инициализацию папки.

```
$ mkdir website.git
$ cd website.git
$ git init --bare
```

Чтобы выйти из репозитория, размещенного на удаленной машине, достаточно просто выполнить команду `exit`.

Вернувшись на локальную машину, мы зададим расположение удаленного репозитория и передадим ему ветвь *master*.

```
$ git remote add origin ssh://webdev@192.168.1.100/~/website.git
$ git push origin master
```

Предположим, что нам нужно создать новый элемент сайта вместе с другим разработчиком. Мы можем добавить новую ветвь *new_feature* и спокойно работать над дизайном. Когда все будет готово, мы добавим эту ветвь в удаленный репозиторий.

```
$ git checkout -b new_feature
$ git push origin new_feature
```

Теперь давайте посмотрим, какие ветви находятся в нашем удаленном репозитории.

```
$ git branch -r
```

Эта команда выдаст нам список ветвей. Мы не найдем в нем версий *layout_a* и *layout_b*, так как они были удалены на локальной машине и никогда не выходили за ее пределы.

```
origin/HEAD -> origin/master
origin/new_feature
origin/master
```

Чтобы получить доступ к нашему репозиторию, другой разработчик может «клонировать» весь проект. После этого ему нужно будет переключиться на ветку *new_feature*, а затем синхронизировать удаленную ветвь с соответствующей ветвью на локальной машине.

```
$ git clone ssh://webdev@192.168.1.100/~/website.git
$ git checkout -b new_feature
$ git pull origin new_feature
```

После того как ветвь оказывается на машине другого разработчика, цикл начинается заново. С помощью Git с одним и тем же кодом может работать одновременно несколько человек. При этом внесенные изменения можно легко объединять — точно так же, как мы делали это на локальной машине.

Дополнительные возможности

Итак, мы познакомились с основными возможностями системы Git, и теперь самое время поискать другие сферы ее применения. В этом рецепте мы ограничились только текстовыми файлами, но на самом деле Git поддерживает и все остальные типы. Например, Git прекрасно подойдет для управления версиями документов Photoshop, использующихся при создании нескольких вариантов дизайна. Также вы можете научиться загружать старые версии файлов — к примеру, чтобы восстановить вариант сайта, который не понравился директору неделю назад, но теперь вдруг снова ему понадобился.

Git удобно использовать для совместной работы над проектом с открытым исходным кодом. Попробуйте зайти на GitHub[1] и открыть любой такой проект — например, jQuery (подойдет и любая другая библиотека из числа тех, о которых мы говорили в этой книге) — и «клонируйте» библиотеку, создав свой собственный репозиторий Git. С помощью ветвления вы сможете добавить в этот проект что-то новое и, таким образом, стать полноправным участником сообщества веб-разработчиков.

Смотрите также

- Рецепт 36. Как разместить статический сайт на Dropbox
- Рецепт 37. Как установить виртуальную машину
- Практическое руководство по управлению версиями с помощью Git (Pragmatic Version Control Using Git) [Swi08]

[1] http://www.github.com/

6 Тестирование

Если уж мы пишем код, то этот код должен *работать*. Часто нам приходится вручную проверять, правильно ли работает приложение. Иногда за нас это делают другие. В этой главе мы расскажем о том, как выполнять отладку кода в процессе его написания, а также научим вас создавать приемочные тесты, которые можно запускать каждый раз при внесении изменений в код, чтобы убедиться, что он все еще работает.

Рецепт 31. Отладка JavaScript-кода

Задача

После того как мы кое-что изменили на нашем сайте, часть JavaScript-кода перестала правильно работать. Никто не знает, что произошло и как это исправить. Поэтому теперь нам нужно разобраться, в чем проблема, и сделать так, чтобы все снова работало, как раньше.

Ингредиенты

- Современный браузер
- Firebug Lite[1]

Решение

Без правильных инструментов поиск дефекта в JavaScript-коде может быть очень длинным и утомительным. К счастью, существует множество инструментов, способных облегчить нашу задачу. Сейчас во многих браузерах есть встроенная консоль, которая позволяет запускать JavaScript-код из командной строки, обращаться к элементам страницы и проверять, как все это работает, не сохраняя только что добавленные изменения и не перезагружая страницу. В более старых браузерах такую консоль можно добавить с помощью Firebug Lite — букмарклета, совместимого с большинством браузеров.

ВОПРОС/ОТВЕТ. А ЧТО, ЕСЛИ Я ХОЧУ ИСПОЛЬЗОВАТЬ ВСТРОЕННЫЕ ИНСТРУМЕНТЫ ИЛИ РАСШИРЕНИЕ БРАУЗЕРА?

Возможности Firebug Lite полностью покрывают то, что мы собираемся делать в этой главе. Но если вас интересуют более изощренные инструменты, позволяющие изучать код более детально и с меньшими затратами, вот как их можно найти:

- Chrome: Настройка и управление Google Chrome > Инструменты > Консоль JavaScript (View > Developer > JavaScript Console)
- Safari: Safari > Preferences > Advanced > Show Develop menu, а затем Develop > Show Web Inspector
- Firefox: установка Firebug[2]
- Firefox: Инструменты > Веб-разработка > Firebug (Tools > Web Console > Firebug)
- IE: установка IE Developer Toolbar[3]

Поскольку в Firebug Lite есть все инструменты, которые нам понадобятся в этой главе, мы не будем подробно рассказывать об особенностях встроенных консолей разных браузеров. Все, что мы будем делать с помощью Firebug Lite, можно сделать и с помощью встроенных инструментов браузеров и расширений, включая полное расширение Firebug для Firefox. Чтобы вы могли сами выбрать, в каком браузере вы будете

[1] http://getfirebug.com/firebuglite
[2] http://getfirebug.com/
[3] http://www.microsoft.com/download/en/details.aspx?id=18359

работать, мы будем тестировать код с помощью Firebug. Однако не стоит пренебрегать встроенными возможностями вашего браузера. Названия инструментов и надписи на кнопках могут отличаться, но суть остается неизменной.

Общие сведения о Firebug

Как только Firebug Lite появится в панели закладок, мы можем одним щелчком мыши открыть в нижней части окна консоль Firebug. Эта консоль позволяет запускать JavaScript-код, а затем следить за поведением элементов страницы.

Чтобы продемонстрировать, как работает эта консоль, а заодно и проверить, работает ли она вообще, мы создадим окно с сообщением об ошибке. В нижней части окна после символов `>>>` введите `alert('Pretty neat!');` и нажмите клавишу Enter. На экране появится окно с сообщением об ошибке (рис. 6.1).

Кроме того, Firebug позволяет просматривать HTML- и CSS-код элементов страницы, а также объектную модель документа (DOM) (рис. 6.2). В левой верхней части консоли

Рис. 6.1. Выполнение JavaScript-кода в Firebug

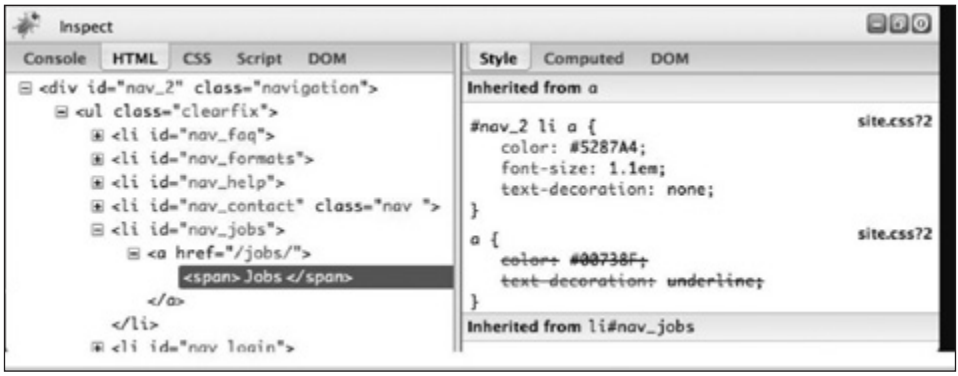

Рис. 6.2. Анализ элементов в Firebug

Firebug есть кнопка Анализировать элемент (Inspect). Щелкните на ней мышью и переместите курсор на любой элемент страницы. В левой области будет показано местоположение этого элемента в HTML-коде, а в правой — его таблицы стилей. Щелкнув на выбранном элементе, мы можем внимательно его изучить и внести любые изменения прямо в браузере; при необходимости — например, если вам понадобился список атрибутов — можно перейти на вкладку DOM.

Возможности Firebug этим не ограничиваются. Однако на начальном этапе нам этого будет вполне достаточно. О том, какие еще инструменты для отладки JavaScript предлагает нам Firebug, можно узнать на http://getfirebug.com/.

Отладка с помощью Firebug

Имея представление об основных возможностях Firebug, мы можем приступить к отладке сайта, о котором говорили в начале раздела. В код из рецепта 5 («Создание и оформление встроенных окон подсказки») были внесены кое-какие изменения, и теперь ссылки не заменяются на значки. К сожалению, мы не использовали Git[1], и никто не догадался сохранить старую версию кода, так что теперь нам придется разбираться во всем самим. Чтобы запустить последнюю версию сайта, загрузите файлы из рецепта 5 («Создание и оформление встроенных окон подсказки») и замените содержимое helper-text.js на helper-text-broken.js.

javascriptdebugging/helper-text-broken.js

```
function display_help_for(element) {
  url = $(element).attr("href"); //URL, который нужно загрузить через AJAX
  help_text_element =
    "#_"+$(element).attr("id")+"_"+
    $(element).attr("data-style");
  //Если контент уже загружен, не нужно делать это еще раз
  if ($(help_text_element).html() == "") {
    $.get(url, { },
      function(data){
        $(help_text_element).html(data);
        if ($(element).attr("data-style") == "dialog") {
          activate_dialog_for(element,$(element).attr("data-modal"));
        }
        toggle_display_of(help_text_element);
    });
  }
  else { toggle_display_of(help_text_element); }
}
```

Загрузив страницу с неправильно работающим JavaScript-кодом, мы сразу видим, что ссылки не заменяются на значки. Как вы, возможно, помните, эта замена производилась в функции `append_help_to()`. Очевидно, в этой функции что-то работает не так и соответствующая строка кода не выполняется. Давайте попробуем ее запустить и посмотрим, что произойдет. Поскольку функции нужно передать параметр, мы воспользуемся

[1] Более подробно об управлении исходным кодом см. «Рецепт 30. Управление файлами с помощью Git».

кнопкой Анализировать элемент (Inspect), чтобы узнать идентификатор ссылки. Далее мы перейдем в консоль и вызовем `append_help_to($('#help_link_1'));`.

Ничего не произошло. Давайте проверим, что мы находимся в нужном месте и что код действительно выполняется. Для этого мы вызовем из функции helper-text-broken.js метод `console.log()`, который выводит в консоль то, что передается ему в качестве параметра:

javascriptdebugging/helper-text-broken.js

```
helperDiv.setAttribute("title", title);

console.log(element);
console.log(helperDiv);
console.log(icon);

$(element).after(helperDiv);
$(element).html(icon)
```

Методу `console.log()` можно передать строку, объект или вызов функции. В данном случае мы хотим узнать, с каким элементом мы в данный момент работаем, и получить значения тех параметров, которые добавляем на страницу. Так мы надеемся узнать, в чем причина сложившейся ситуации.

После того как мы изменили код, необходимо обновить страницу. После этого мы откроем Firebug и снова запустим `append_help_to($('#help_link_1'));` (рис. 6.3).

Рис. 6.3. Анализ элементов в Firebug

Трижды вызванный метод `console.log()` должен возвратить три значения: элемент, с которым мы работаем; `<div>`, который будет содержать текст подсказки; значок, который будет использоваться вместо текста ссылки. Вместо этого мы видим только первые два значения, а третье отображается как `undefined`, — это значит, что при создании значка что-то пошло не так.

javascriptdebugging/helper-text-broken.js

```
1   function set_icon_to(help_icon) {
2   is_image = /jpg|jpeg|png|gif$/
3   if (help_icon = undefined)
4      { icon = "[?]"; }
```

продолжение ⇩

```
5    else if (is_image.test(help_icon))
6      { icon = "<img src='"+help_icon+"'>"; }
7    else
8      { icon = help_icon; }
9  }
```

Посмотрев на функцию `set_icon_to()`, можно сразу понять, в чем причина ошибки. В строке 3 вместо знака `==` для сравнения `help_icon` с `undefined` мы используем знак `=`, который присваивает `help_icon` значение `undefined`. Давайте исправим это и обновим страницу.

```
if (help_icon == undefined)
```

Теперь все выглядит просто прекрасно. Значение `help_icon` задается так, как нужно, и мы можем спокойно работать дальше. Благодаря Firebug мы смогли легко убедиться в том, что наши функции действительно запускаются, посмотреть на свойства некоторых элементов и переменных и в итоге понять, что было не так с нашим кодом.

Дополнительные возможности

В одном из вызовов `console.log()` мы передали в качестве параметра элемент. Запустите функцию еще раз и щелкните на этом элементе в консоли. Точно такой же эффект получится, если щелкнуть на элементе страницы в режиме анализа. При написании нового JavaScript-кода такой прием поможет вам следить за тем, чтобы элементы были выбраны правильно. Кроме того, с помощью вкладок CSS и DOM можно проверять, правильно ли задаются значения атрибутов.

До сих пор мы запускали в консоли только однострочные команды. Но что, если мы захотим написать в Firebug целую функцию? Справа в самом конце поля ввода текста консоли есть кнопка с изображенным на ней треугольником. Если щелкнуть на ней, на экране появится дополнительная панель, в которой можно писать JavaScript-функции с переносами строк, а затем запускать их с помощью кнопки Выполнить (Run). Так можно тестировать новые функции перед добавлением их на страницу.

Написание тестов для JavaScript-кода сильно упростило бы нам задачу. С помощью тщательно разработанного набора тестов — об одном из них рассказывается в рецепте 35 («Тестирование JavaScript-кода с помощью Jasmine») — можно легко следить за тем, как внесенные изменения влияют на работу сайта. Загляните в эту главу и попробуйте написать тесты для рецепта 5 («Создание и оформление встроенных окон подсказки») или других наших проектов, использующих JavaScript.

Смотрите также

- Рецепт 35. Тестирование JavaScript-кода с помощью Jasmine
- Рецепт 5. Создание и оформление встроенных окон подсказки

Рецепт 32. Определение областей активности пользователей с помощью «тепловой карты»

Задача

При раскрутке или модернизации сайта важно знать, какие детали работают эффективно, а какие нет. Так гораздо проще понять, на что стоит потратить свое время. Для этого мы должны научиться определять, какие области страницы используются наиболее активно.

Ингредиенты

- PHP-сервер
- ClickHeat[1]

Решение

Мы можем определять, в каких областях страницы пользователь использует щелчок мыши, и отображать эти данные в виде дополнительного графического слоя — «тепловой карты». Посмотрев на такую «карту», можно моментально определить, какие элементы страницы используются наиболее активно. Существует несколько коммерческих программ, позволяющих составлять «тепловые карты», однако в этом разделе мы будем использовать бесплатный скрипт ClickHeat, так как на современных хостах установить его почти так же просто, как аналогичные коммерческие продукты.

С помощью ClickHeat мы попробуем разрешить небольшой внутренний спор. Наш клиент давно бы уже выпустил свой новый продукт, если бы не разногласия между двумя его партнерами: один из них считает, что кнопка Регистрация (Sign Up) используется недостаточно активно, а другой думает то же самое о кнопке Подробнее (Learn More). В интерфейсе эти две кнопки расположены рядом. Если мы в течение некоторого времени будем фиксировать щелчки мыши на странице, мы сможем узнать, какая из кнопок используется активнее.

Настройка ClickHeat

Хотя для работы ClickHeat требуется PHP, его можно использовать на любом сайте. Для этого нужно просто загрузить ClickHeat с сайта проекта и поместить все его скрипты на сервер в папку с подключенным PHP. В этом рецепте мы будем работать с виртуальной машиной, размещенной в нашей сети на http://192.168.1.100. Чтобы узнать, как создать свою собственную виртуальную машину для тестирования, обратитесь к рецепту 37 («Как установить виртуальную машину»).

Распаковав архив ClickHeat, мы получим папку clickheat. Далее ее нужно передать в /var/www — папку на виртуальной машине, в которой хранятся текущие веб-страницы. Так как на виртуальной машине есть поддержка SSH, файлы можно скопировать с помощью всего одной команды:

```
scp -R clickheat webdev@192.168.1.100:/var/www/clickheat
```

[1] http://www.labsmedia.com/clickheat/index.html

Другой способ передать файлы на сервер в папку /var/www — использовать STFP-клиент, например FileZilla.

После того как мы скопировали код на сервер, необходимо изменить настройки доступа для папок, находящихся внутри clickheat; после этого мы сможем записывать данные в файлы регистрации и изменять параметры доступа. Давайте войдем на сервер и разрешим запись в папки config, tmp и logs с помощью команды chmod.

```
$ ssh webdev@192.168.1.100
$ cd /var/www/clickheat
$ chmod -R 777 config logs cache
$ exit
```

Теперь попробуйте открыть http://192.168.1.100/clickheat/index.php. После того как ClickHeat подтвердит, что в папке с настройками разрешена запись, мы сможем задать все оставшиеся параметры.

В нашем случае эти значения можно и не задавать, но мы все-таки введем имя пользователя и пароль администратора. Если вы щелкнете на кнопке Check Configuration и система не выдаст сообщений об ошибках, это будет означать, что настройка завершена. Теперь самое время подключить ClickHeat к нашей веб-странице.

Регистрация щелчков мыши и просмотр результатов

Чтобы запустить регистрацию щелчков мыши, достаточно добавить несколько строк JavaScript-кода на начальную страницу после закрывающего тега <body>.

heatmaps/index.html

```
<script type="text/javascript"
  src="http://192.168.1.100/clickheat/js/clickheat.js"></script>
<script type="text/javascript">
  clickHeatSite = 'AwesomeCo';
  clickHeatGroup = 'buttons';
  clickHeatServer = 'http://192.168.1.100/clickheat/click.php';
  initClickHeat();
</script>
```

Для построения «тепловой карты» необходимо задать «сайт» («site») и «группу» («group»); на самом деле ClickHeat позволяет работать с несколькими сайтами одновременно.

После того как мы перенесем нашу страницу на этот сервер, щелчки мыши пользователей будут записываться в файлы регистрации ClickHeat. Через пару часов можно зайти на http://192.168.1.100/clickheat/index.php и посмотреть, что у нас получилось (рис. 6.4).

Похоже, гораздо больше людей используют кнопку Регистрация (Sign Up)!

Рис. 6.4. Наша «тепловая карта»

Дополнительные возможности

Как только вы запустите ClickHeat, он будет работать самостоятельно, не требуя от вас никаких дополнительных действий. Однако существует множество параметров, которые вы можете настроить перед запуском, — например, сколько щелчков следует регистрировать для одного пользователя. Если на сервере PHP работает слишком медленно и его невыгодно запускать при каждом отдельном запросе, можно записывать результаты в файлы регистрации Apache, а затем анализировать их с помощью специального скрипта. Наконец, запустив ClickHeat на отдельном сервере, можно настроить его для сбора данных с нескольких сайтов или доменов. Чтобы узнать о других параметрах ClickHeat, загляните в его документацию на сайте или просто внимательно изучите интерфейс.

Если вам нужен более простой вариант, выберите один из коммерческих продуктов — к примеру, Crazy Egg[1], обладающий сходными функциональными возможностями.

Иногда «тепловая карта» сайта может показать, что пользователи ведут себя немного странно. Если вы вдруг увидите всплеск активности в области страницы, где нет никаких ссылок, подумайте, не следует ли сделать эту область активной. «Тепловые карты» часто открывают нам то, о чем мы раньше и не догадывались.

Смотрите также

- Рецепт 37. Как установить виртуальную машину

[1] http://www.crazyegg.com/home3

Рецепт 33. Тестирование в браузере с помощью Selenium

Задача

Тестирование — длительный и трудоемкий процесс. С ростом сложности сайтов возрастает необходимость в последовательном наборе тестов, которые можно было бы использовать многократно. Без автоматизированного тестирования единственный способ содержать сайт в рабочем состоянии — нанять высококвалифицированного сотрудника, который будет работать круглые сутки, заполняя бесконечно длинные контрольные листы. Все это будет выполняться крайне медленно. Поэтому нам нужно как-то ускорить процесс тестирования и научиться создавать такие тесты, которые можно будет запускать по мере необходимости, сверяя работу сайта сегодня с тем, как он будет работать несколько месяцев спустя.

Ингредиенты

- Firefox[1]
- Selenium IDE[2]
- QEDServer (для сервера тестирования); см. «QEDServer» (с. xvi)

Решение

В дополнение к тестам, которые мы запускаем вручную, можно использовать автоматизированные инструменты для тестирования. С помощью плагина для Firefox Selenium IDE мы можем создавать тесты в графической среде, выполняя операции, требующие проверки, прямо на сайте; при этом все наши действия будут записываться. По мере выполнения этих действий можно создавать *проверки* — небольшие тесты, которые будут определять, есть ли на странице необходимые элементы. Далее их можно будет запустить в любой момент. Таким образом мы получим автоматизированные тесты, пригодные для повторного использования.

Наша команда разработчиков создала сайт для управления продукцией, и теперь начальник просит нас сделать так, чтобы этот сайт всегда работал правильно. Команда разработчиков уже добавила несколько модульных тестов, относящихся к бизнес-логике сайта; нам же поручили создание автоматизированных тестов для пользовательского интерфейса. Благодаря таким тестам мы, а также команда разработчиков сможем не беспокоиться о том, что произойдет, когда в пользовательский интерфейс будут внесены изменения.

Настройка среды тестирования

Для начала нам нужно установить браузер Firefox. Откройте сайт Firefox и следуйте указаниям в соответствии с вашей операционной системой.

---

[1] http://getfirefox.com/

[2] http://seleniumhq.org/download/

Далее необходимо установить Selenium IDE. Откройте Firefox, зайдите на сайт Selenium[1] и загрузите последнюю версию[2].

После того как мы все установили, можно приступить к написанию первого теста.

Наш первый тест

Чтобы создать наш первый тест, мы с помощью Selenium IDE запишем последовательность действий на сервер для тестирования. Этот сервер будет работать на нашем компьютере, используя QEDServer. Запустите QEDServer, а затем откройте Firefox. Перейдите на http://localhost:8080, и вы увидите интерфейс сервера для тестирования (рис. 6.5).

Так как мы работаем над приложением для управления продукцией, логично для начала проверить, действительно ли на начальной странице есть ссылка Manage Products и действительно ли она ведет туда, куда мы хотим.

Рис. 6.5. Изображение нашей домашней страницы

Чтобы открыть Selenium IDE, выберите его в меню Инструменты (Tools) в браузере Firefox. Чтобы начать запись, щелкните на кнопке записи. После этого в браузере щелкните на ссылке Manage Products. В этот момент в окне Selenium IDE начнут появляться новые элементы (рис. 6.6). В поле Базовый URL (Base URL) в верхней части окна вы увидите значение http://localhost:8080, а ниже — одну из ключевых команд Selenium IDE: метод clickAndWait(). Работа с веб-приложениями в значительной степени состоит в том, что мы щелкаем на кнопках и ссылках, а затем ждем, пока страница загрузится. Именно это и делает clickAndWait(). При каждом нажатии на ссылке Selenium IDE добавляет в наш тест этот метод, а также текст, позволяющий идентифицировать ссылку. Когда мы запустим тест, этот метод и текст ссылки будут служить указаниями к действиям браузера.

---

[1] http://seleniumhq.org/download/

[2] Если вы работаете в Firefox 4, вам, возможно, потребуется расширение Add-On Compatibility Reporter версии 0.8.2.

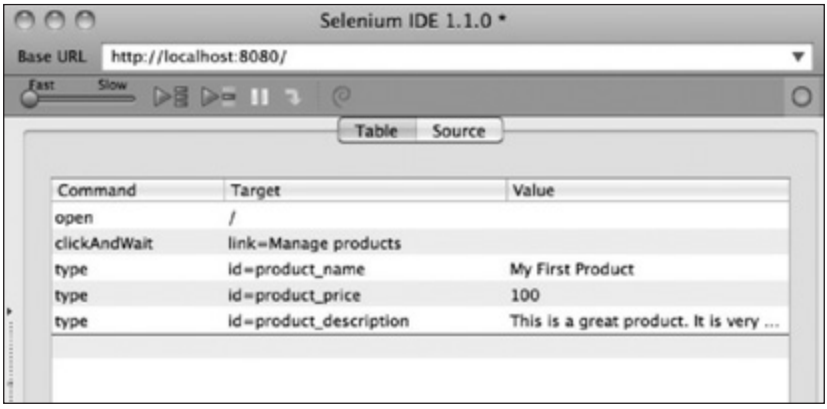

Рис. 6.6. Первый тест, созданный с помощью Selenium IDE

В окне Selenium IDE для каждого действия отображаются три составляющие: Команда (Command) — действие, которое выполняет Selenium; Цель (Target) — элемент, над которым совершается это действие; Значение (Value) используется для полей, требующих ввода данных (например, текстовых полей или переключателей).

Selenium IDE известен своими функциями определения местоположения. С помощью этого мощного инструмента мы можем находить элементы на странице не только по идентификатору, но и по объектной модели документа, запросу XPath, CSS-селектору или даже обычному тексту. Когда мы нажали на ссылке Manage Products, наша цель — link=Manage products. В этом выражении link= является селектором, и именно по нему мы выбираем блок текста, над которым будет производиться действие. Помните, что по умолчанию локаторы ищут идентификатор, за которым сразу же следует заданная строка. Поиск идентификатору может существенно ускорить работу ваших тестов и увеличить их точность, однако такие тесты будет гораздо сложнее читать.

Итак, мы уже знаем кое-что о локаторах, поэтому давайте перейдем к командам. Команды — это действия, которые Selenium выполняет при запуске теста. Selenium может делать то же самое, что и человек, за исключением одного: он не может передавать файлы — по крайней мере, без значительных модификаций. Это позволяет нам имитировать взаимодействие человека с машиной, что, в свою очередь, делает наши тесты максимально приближенными к реальности.

Давайте проверим, что на странице, открывшейся после нажатия Manage Products, присутствует слово «Products». Для начала нужно щелкнуть на ссылке Manage Products, а затем проверить, есть ли на экране слово «Products». Чтобы добавить в тест такое действие, найдите это слово и щелкните на нем правой кнопкой мыши. В контекстном меню выберите команду verifyTextPresent. То же самое можно сделать в окне Selenium IDE: для этого нужно щелкнуть на свободном месте под командой clickAndWait() и задать команду, цель и значение в соответствующих полях формы. Однако это занимает гораздо больше времени, поэтому, чтобы ускорить процесс, Selenium IDE добавил в контекстное меню несколько вспомогательных функций.

Чтобы сохранить этот тест, выберите Сохранить тестовый сценарий (Save Test Case) в меню Файл (File) окна Selenium IDE. Для запуска теста достаточно нажать на кнопке запуска, расположенной под окном Базовый URL (Base URL). По мере того как браузер будет передвигаться по сайту, после успешного выполнения каждого шага фон страницы будет окрашиваться в зеленый цвет. В случае неудачи фон станет красным, а в окне

регистрации появится красный полужирный текст с описанием возникшей проблемы. Эта информация поможет вовремя обнаружить проблему и попытаться ее решить.

Более сложный тест

Итак, мы хотим проверить, что наше приложение действительно работает правильно, а именно что мы можем добавлять и удалять товары, а также просматривать информацию о них. Поскольку такая проверка будет выполняться в несколько этапов, мы снова обратимся к Selenium.

Для начала вернитесь на http://localhost:8080 и запустите Selenium IDE. Щелкните на ссылке Manage Products и дождитесь загрузки страницы. Далее найдите строку с текстом «New Product», щелкните на ней правой кнопкой мыши и выберите verifyTextPresent New Product. Оставьте форму пустой и щелкните на кнопке Add Product. Так как приложение требует указания минимальных сведений о товаре, пустая форма не будет отправлена и на экране появится сообщение об ошибке.

Мы сделаем это частью нашего теста. Щелкните на сообщении The product was not saved правой кнопкой мыши, а затем с помощью команды verifyTextPresent добавьте проверку, которая будет искать это сообщение на странице Products.

Убедившись в том, что сообщение об ошибке работает правильно, мы можем заполнить форму и отправить ее. Selenium IDE выделит отдельную строку для каждого заполненного поля формы и отобразит в ней значение, которое мы ввели.

На этот раз после отправки формы мы перейдем на страницу со списком товаров и увидим на ней сообщение Created. Чтобы проверить, что это сообщение действительно отображается, мы снова воспользуемся командой verifyTextPresent().

Теперь у нас наконец-то есть такой тест, который проверяет работу сразу нескольких ключевых элементов сайта. Сейчас мы его просто сохраним. Если впоследствии кто-то повредит наш сайт, мы сможет легко узнать, что не так, просто запустив этот тест.

Дополнительные возможности

Следующим шагом может стать запуск набора тестов. Пока все наши тесты приходится загружать в Selenium IDE по отдельности, что при большом количестве тестов становится крайне неудобным. Чтобы создавать автоматизированные наборы тестов и запускать их в нескольких браузерах, воспользуйтесь Selenium Remote Control и Selenium Grid[1].

Хотя Selenium в первую очередь является инструментом для тестирования, его можно использовать и при решении других задач автоматизации. Например, если вы пользуетесь системой учета рабочего времени или консолью управления, требующей выполнения большого числа повторяющихся операций, Selenium IDE может избавить вас от лишних нажатий клавиш и щелчков мыши.

Смотрите также

- Рецепт 34. Тестирование с помощью Selenium и Cucumber
- Рецепт 35. Тестирование JavaScript-кода с помощью Jasmine

[1] http://selenium-grid.seleniumhq.org/

Рецепт 34. Тестирование с помощью Selenium и Cucumber

Задача

Тестирование в браузере может отнимать много времени и сил. В рецепте 33 («Тестирование в браузере с помощью Selenium») мы рассказывали о том, как создавать тесты в Selenium IDE. К сожалению, такой способ работает только для Firefox. Следовательно, он нам не подходит: тестирование должно охватывать все браузеры, из которых пользователи могут зайти на наш сайт. Запускать тесты в каждом из браузеров вручную — тоже не самая лучшая идея, поскольку в таком случае нам придется установить все эти браузеры на нашем компьютере. Таким образом, нам нужен инструмент для автоматизированного тестирования в нескольких браузерах, не требующий установки этих браузеров.

Ингредиенты

- Cucumber Testing Harness [1]
- QEDServer[2]
- Аккаунт на Sauce Labs[3]
- Sauce Connect[4]
- Bundler[5]

Решение

Сервер, на котором мы проводим тестирование нашего сайта, содержит интерфейс для администрирования, с помощью которого можно управлять списком товаров. Мы уже протестировали его на нашем компьютере в Firefox и Safari и теперь хотим сделать то же самое еще в нескольких браузерах. Для этого, используя целый набор инструментов, включая Cucumber и Selenium, мы создадим среду для автоматизированного тестирования в нескольких браузерах.

Инструменты

Как мы увидели в рецепте 33 («Тестирование в браузере с помощью Selenium»), Selenium прекрасно подходит для имитации поведения пользователя. Перемещаясь по сайту, мы записали свои действия, используя Selenium IDE. С помощью Cucumber мы хотим расширить наши возможности: теперь мы будем сами программировать код.

В рецепте 35 («Тестирование JavaScript-кода с помощью Jasmine») мы еще поговорим о разработке, управляемой поведением (behavior-driven development, BDD), и о написании тестов «снаружи внутрь». Cucumber позволяет перейти на более высокий уровень BDD, обеспечивая связь разработчиков с партнерами проекта. Это

[1] http://pragprog.com/book/wbdev/web-development-recipes
[2] http://webdevelopmentrecipes.com/files/qedserver.zip
[3] http://saucelabs.com/
[4] https://saucelabs.com/downloads/Sauce-Connect-latest.zip
[5] http://gembundler.com/

достигается за счет того, что тесты Cucumber пишутся на языке, понятном обычному человеку.

selenium2/cucumber_test/features/manage_products.feature

```
1  Feature: Manage products with the QED Server
2    Scenario: When I view the product details of a new product it should
3                take me to the page where the product information is displayed
4        Given I am on the Products management page
5        And I created a product called "iPad 3" with a price of "500"
6          dollars and a description of "My iPad 3 test product"
7        When I view the details of "iPad 3"
8        Then I should see "iPad 3"
9        And I should see a price of "500"
10        And I should see a description of "My iPad 3 test product"
```

Этот пример наглядно показывает, насколько прост язык тестов Cucumber: по такому описанию можно легко понять, какие действия выполняет тест.

Как мы уже говорили, Selenium позволяет запускать тесты только в Firefox. Чтобы проверить, как сайт работает в других браузерах, мы можем выбрать один из следующих вариантов: установить все браузеры на нашем компьютере и проверить все вручную или использовать облачный сервис для тестирования Selenium — например, тот, который нам предлагает Sauce Labs. Чтобы не мучиться с установкой действительно огромного числа браузеров (пока мы писали эту книгу, Firefox успел выпустить версии 5 и 6!), мы запустим наши тесты с помощью Sauce Labs. Поскольку Sauce Labs записывает работу тестов, мы можем просматривать записи как пройденных, так и непройденных тестов. Такие ролики можно показывать партнерам, и тогда им не придется вручную открывать ваш сайт и проверять, как работает приложение. На момент написания этой книги Sauce Labs предоставлял каждому пользователю 200 минут бесплатного тестирования.

Чтобы собрать все это вместе, нам потребуется Cucumber Testing Harness (CTH, «средство тестирования Cucumber»). Это базовый фреймворк, специально созданный для того, чтобы писать тесты легко и быстро. Иногда бывает очень трудно начать писать тесты — именно так дело обстоит с Cucumber и Selenium. Благодаря CTH этот барьер гораздо легче преодолеть. Sauce Labs справляется со всеми неконтролируемыми процессами, происходящими в Selenium, и обеспечивает Cucumber возможность удаленного управления процессами. Имея доступ к множеству версий каждого браузера и запуская в них тесты, Sauce Labs позволяет имитировать действия пользователей не только современных, но и более старых компьютеров. Сняв таким образом проблему установки множества версий браузеров, мы можем сконцентрировать свое внимание непосредственно на тестировании. Кроме того, так как Sauce Labs является облачным сервисом, во время работы тестов нагрузка на нашу локальную машину будет менее существенной. CTH также упростит нам работу, взяв на себя структурирование кода и управление зависимостями и предлагая нестандартные решения, предназначенные для Sauce Labs. Чтобы начать выполнение тестов, нужно просто изменить несколько файлов конфигурации.

Настройка среды разработки

Для работы CTH требуется Ruby. Если в вашей системе он еще не установлен, сделайте это, прежде чем двигаться дальше, при необходимости заглянув в Приложение 1 «Установка Ruby».

Поскольку для выполнения тестов мы будем использовать Sauce Labs OnDemand, нам понадобится бесплатный аккаунт на Sauce Labs[1].

Большая часть нашей работы будет выполняться в CTH. Загрузите его с сайта книги[2], а затем распакуйте cucumber_test.zip в тот каталог, где будут храниться наши тесты и откуда они будут запускаться. После этого откройте программную оболочку и перейдите в этот каталог.

Внутри CTH вы найдете Gemfile со списком Ruby-джемов, необходимых для правильной работы CTH. Но мы не будем сами их устанавливать: за нас это сделает другой джем — bundle. Давайте установим его, а затем запустим в программной оболочке.

```
$ gem install bundler
$ bundle install
```

Теперь мы можем перейти к следующему этапу настройки. Давайте добавим к CTH ключ API Sauce Labs.

Зайдите на Sauce Labs и откройте страницу My Account. На момент написания этой книги ключ API можно было получить, щелкнув на ссылке View My API Key (рис. 6.7). Скопируйте его и поместите в файл config/ondemand.yml.

selenium2/cucumber_test/config/ondemand.yml

```
#---
username: my_sauce_user_name
access_key: my_super_secret_key
```

Рис. 6.7. Как найти ключ API Sauce Labs

CTH будет обращаться к файлу config/ondemand.yml во время запуска тестов на Sauce Labs. В поле `username` нужно ввести имя пользователя, под которым вы зарегистрированы на Sauce Labs.

Чтобы проще было следить за процессом тестирования, мы можем указать в тестах имя хоста. Так как для тестирования мы используем локальный сервер, давайте назовем его «qedserver.local.». Нам нужно сообщить компьютеру, запрашивающему данное имя хоста, куда идти дальше. Для этого мы откроем файл hosts (/etc/hosts в OS X и Linux и C:\Windows\system32\drivers\etc\hosts в Windows) и добавим в него следующее:

```
127.0.0.1 qedserver.local
```

Теперь наш компьютер будет направлять запросы к «qedserver.local.» на 127.0.0.1, то есть на нашу локальную машину. Давайте запустим сервер для тестирования и проверим, как это работает, набрав в браузере http://qedserver.local:8080.

---

[1] https://saucelabs.com/signup

[2] http://pragprog.com/book/wbdev/web-development-recipes

Далее нам нужно установить связь между Sauce Labs и нашим сервером для тестирования. К счастью, специально для этого существует Sauce Connect[1]. После загрузки его нужно распаковать, а затем перейти в соответствующий каталог в программной оболочке и выполнить jar-файл, заменив поле `USERNAME` на ваше имя пользователя, а `API_KEY` — на ваш ключ API.

```
$ java -jar Sauce-Connect.jar USERNAME API_KEY
```

Таким образом, мы смогли обеспечить Sauce Labs доступ к нашей локальной машине и при этом нам не пришлось выяснять свой публичный IP-адрес или открывать порты на маршрутизаторе.

Итак, соединение с Sauce Labs установлено и мы можем перейти к заключительному этапу настройки CTH. Внутри папки config вы найдете файл cucumber.yml. В нем мы зададим несколько параметров по умолчанию, общих для всех браузеров, а затем — параметры, относящиеся к каждому отдельному браузеру (из числа тех, в которых мы будем проводить тестирование).

selenium2/cucumber_test/config/cucumber.yml

```
<% defaults = "HOST_TO_TEST=http://qedserver.local
               APP_PORT=8080
               HUB=sauce" %>
```

Здесь мы определяем переменную `defaults`. Она представляет собой строку, состоящую из нескольких значений. `HOST_TO_TEST` — URL тестируемого приложения, которым в данном случае является QEDServer. Так как наш сервер работает на порте 8080, мы укажем этот номер в объявлении `APP_PORT`. Объявление `HUB` сообщает инструменту для тестирования о том, что мы будем использовать хаб *sauce*, который управляется Sauce Labs.

ВОПРОС/ОТВЕТ. А ЧТО, ЕСЛИ МЫ ХОТИМ, ЧТОБЫ ВСЕ ЭТО РАБОТАЛО НА НАШЕЙ МАШИНЕ?

CTH можно использовать также и на локальной машине, однако если мы не хотим ограничиваться браузером Firefox, придется установить Ant[2] и Selenium Grid[3]. С установкой Selenium Grid процесс тестирования станет несколько сложнее, но если вы вынуждены отказаться от использования внешних ресурсов, существует возможность сделать так, чтобы все это работало полностью на вашем компьютере или сервере.

Все это затевалось главным образом для того, чтобы иметь возможность запускать тесты в нескольких браузерах. Поэтому теперь мы зададим флаги, с помощью которых можно будет легко переключаться между браузерами. Мы решили добавить поддержку Internet Explorer 7, 8 и 9, а также последних версий Safari, Firefox и Chrome. Здесь, как и везде, удобно будет придерживаться соглашения об именовании: при большом количестве браузеров так гораздо легче находить нужные профили. Имя каждого профиля будет начинаться с префикса `sauce` (поскольку этот тест будет запускаться на Sauce Labs), за которым будут следовать сокращенные названия браузера и операционной

[1] https://saucelabs.com/downloads/Sauce-Connect-latest.zip
[2] http://ant.apache.org/
[3] http://selenium-grid.seleniumhq.org/download.html

системы. Например, для профиля IE7 будет использоваться имя `sauce_ie7_03`, где `ie7` означает Internet Explorer 7, а `03` — Windows Server 2003.

selenium2/cucumber_test/config/cucumber.yml

```
sauce_ie7_03: BROWSER=iehta VERSION='7.' <%= defaults %>
sauce_ie8_03: BROWSER=iehta VERSION='8.' <%= defaults %>
sauce_ie9_08: BROWSER=ietha VERSION='9.' OS='Windows 2008'<%= defaults %>
sauce_f_03: BROWSER=firefox VERSION='3.' <%= defaults %>
sauce_s_03: BROWSER=safariproxy VERSION='5.' <%= defaults %>
sauce_c_03: BROWSER=googlechrome VERSION=' ' <%= defaults %>
```

Теперь у нас есть полный комплект браузеров. Большинство тестов мы будем запускать на Windows Server 2003; исключение составит Internet Explorer 9, доступный только на Windows Server 2008[1]. По мере необходимости в этот список можно добавлять другие конфигурации браузеров и серверов.

Наш первый тест

Начнем с чего-нибудь очень простого: проверим, что на начальной странице есть ссылки Manage Products и See quick tutorial.

Тесты Cucumber группируются по элементам (features): в каждой из таких групп тест проверяет определенный элемент приложения. В разделах `feature` мы описываем поведение приложения в виде отдельных сценариев (scenarios). Сценарии описываются по схеме «Пусть...; если..., то...» (given, when, then), которая четко задает порядок действий при тестировании.

Операторы `given` задают контекст теста и как бы говорят: «Не думай о том, что было до этого; сейчас мы находимся здесь, и ситуация выглядит так». Операторы `when` описывают действие, которое необходимо выполнить. Операторы `then` являются собственно подтверждениями теста; то, что стоит после оператора `then`, должно совпадать с результатом сценария.

Поскольку первое, что мы хотим проверить, касается домашней страницы сервера, мы поместим этот код в файл qedserver_home_page.feature. Давайте начнем с объявления `feature` и двух объявлений `scenario`, соответствующих двум этапам тестирования.

selenium2/cucumber_test/features/qedserver_home_page.feature

```
Feature: Testing the QED Server home page to make sure we have the
  manage products link and the See a quick tutorial link

  Scenario: Verify the manage products link is on the home page
    Given I am on the QED Server home page
    Then I should see the "Manage products" link

  Scenario: Verify the See a quick tutorial link is on the home page
    Given I am on the QED Server home page
    Then I should see the "See a quick tutorial" link
```

Этот раздел `feature` описывает то, чего мы ждем от нашей начальной страницы. В нем содержится два сценария — по одному для каждой ссылки, наличие которых

[1] https://saucelabs.com/docs/ondemand/browsers/env/ruby/se2/windows

мы и хотим проверить. В данном случае сценарии используют только операторы `given` и `then`. Ниже, в разделе «Более сложный тест», мы поговорим о сценариях, способных выполнять действия.

Теперь давайте запустим наш тест. Здесь в игру вступают ранее созданные профили. С помощью команды `$ cucumber -p sauce_f_03` мы сообщим Cucumber, что приложение нужно протестировать в `sauce_f_03` или в Firefox на Sauce Labs.

После запуска feature-файла Cucumber в программной оболочке появляется ответ, который выглядит примерно следующим образом:

```
$ cucumber -p sauce_f

Using the sauce_f profile...
Feature: Testing the test server home page to make sure we have the
manage products link and the See a quick tutorial link

Scenario: Verify the manage products link is on the home page
# features/qedserver_home_page.feature:3
  Given I am on the QED Server home page
# features/qedserver_home_page.feature:4
  Then I should see the "manage products" link
# features/qedserver_home_page.feature:5

Scenario: Verify the See a quick tutorial link is on the home page
# features/qedserver_home_page.feature:7
  Given I am on the QED Server home page
# features/qedserver_home_page.feature:8
  Then I should see the "See a quick tutorial" link
# features/qedserver_home_page.feature:9

2 scenarios (2 undefined)
4 steps (4 undefined)
0m26.580s

You can implement step definitions for undefined steps with these snippets:

Given /^I am on the QED Server home page$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^I should see the "([^"]*)" link$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Ответ Cucumber содержит достаточно много информации о нашем тесте. После указания профиля, на котором был запущен тест, выводится оператор `feature`. Далее следуют сценарии, в которых для каждого оператора указывается номер соответствующей строки в тесте. После сценариев приводится информация о том, что произошло во время работы теста. В данном случае два сценария из двух оказались неопределенными, так же как и четыре шага (steps) из четырех. В последней части ответа содержатся

небольшие фрагменты кода, на основе которых мы будем строить нашу реализацию шагов, заданных оператором `feature`.

Теперь нам нужно реализовать шаги, необходимые для тестирования элементов (features). Давайте создадим файл /features/step_definitions/qedserver_home_page_steps.rb, в котором будут храниться определения шагов. Эти определения представляют собой код на языке Ruby, который наши элементы (features) будут запускать, для того чтобы заставить Selenium выполнить в браузере определенные действия.

selenium2/cucumber_test/features/step_definitions/qedserver_home_page_steps.rb

```
1  Given /^I am on the QED Server home page$/ do
2    pending # express the regexp above with the code you wish you had
3  end
4
5  Then /^I should see the "([^"]*)" link$/ do |arg1|
6    pending # express the regexp above with the code you wish you had
7  end
```

По такому step-файлу Selenium не сможет понять, как должен происходить процесс тестирования. Строки, начинающиеся со слова `pending`, нужно переписать так, чтобы они действительно проверяли содержимое страницы. На первом шаге мы вызовем оператор `given`, который сообщает нам, что тестирование должно начинаться с начальной страницы сайта. На следующем шаге нам нужно будет определить, действительно ли на странице есть нужная ссылка. Для этого нам потребуется локатор Selenium.

```
Given /^I am on the QED Server home page$/ do
  @selenium.open("/")
end
Then /^I should see the "([^"]*)" link$/ do |link_text|
  @selenium.element?("link=#{link_text}").should be_true
end
```

Строки 2 и 5 начинаются с объекта `@selenium`, который благодаря CTH может передавать команды Selenium. В строке 2 мы использовали метод `open()`, передав ему значение `"/"`. А поскольку в файле CTH cucumber.yml значение `HOST_TO_TEST` уже задано, Selenium знает, что в качестве базового URL нужно использовать http://qedserver.local:8080.

В строке 5 использовали метод `element?()`, передав ему строку с локатором. Не углубляясь в тонкости языка Ruby, мы создаем эту строку динамически с помощью интерполяции строк. Благодаря этому мы можем использовать не константу, а переменную `link_text`. Значение этой переменной мы получаем из оператора `then` с помощью регулярного выражения.

Давайте снова запустим наши тесты. На этот раз они будут выполнены.

```
$ cucumber -p sauce_f
....
2 scenarios (2 passed)
4 steps (4 passed)
0m36.439s
```

Теперь самое время перейти на Sauce Labs и посмотреть на результат выполнения каждого теста. Щелкнув на ссылке My Jobs[1], можно узнать, какие тесты были запущены. Чтобы получить информацию о тесте (рис. 6.8), а также просмотреть запись его прохождения, щелкните на названии теста.

Более сложный тест

Несмотря на все преимущества простых тестов, иногда нам требуется выполнять гораздо более сложные действия: например, если нужно проверить, как работает интерфейс управления товарами. Для решения этой задачи мы создадим отдельный feature-файл manage_products.feature. В объявление `feature` мы поместим следующее: «Manage products with the QED Server» (Управление товарами с помощью сервера QED). Для такой проверки мы будем использовать сценарий, описывающий создание нового товара и просмотр информации о нем.

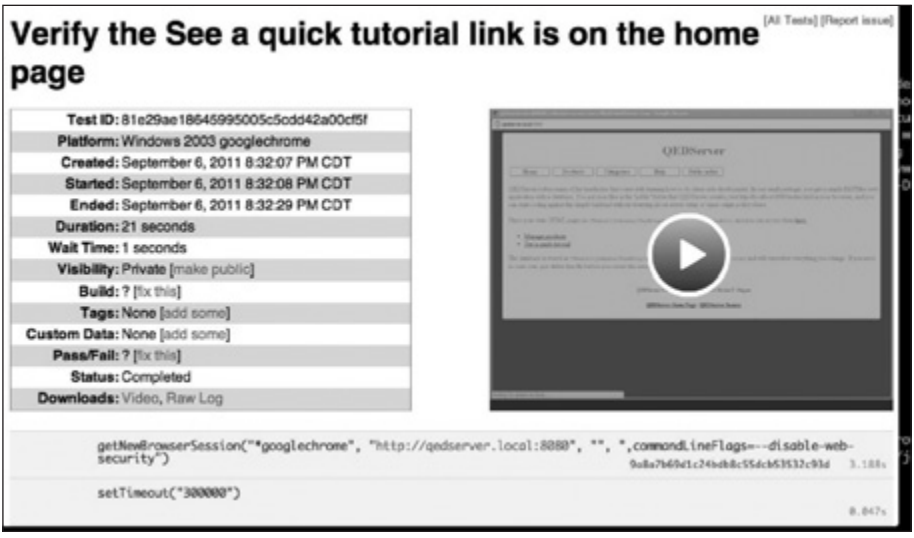

Рис. 6.8. Информация о выполненном тесте

selenium2/cucumber_test/features/manage_products.feature

```
1  Feature: Manage products with the QED Server
2    Scenario: When I view the product details of a new product it should take me
3                to the page where the product information is displayed
4        Given I am on the Products management page
5        And I created a product called "iPad 3" with a price of "500"
6          dollars and a description of "My iPad 3 test product"
7        When I view the details of "iPad 3"
8        Then I should see "iPad 3"
9        And I should see a price of "500"
10        And I should see a description of "My iPad 3 test product"
```

[1] https://saucelabs.com/jobs

По структуре этот тест не отличается от нашего первого теста: он начинается с операторов `feature` и `scenario`, за которыми следуют первые строки теста. В строке 5 появляется новый для нас оператор — `and`, который используется для соединения нескольких операторов. Оператор `and` также встречается в строках 9 и 10. Чтобы определить, когда нужно поставить этот оператор, достаточно прочитать строки теста вслух: если пауза или слово «и» звучат естественно, самое время использовать `and`. В строке 7 мы описываем действие с помощью оператора `when`.

Теперь давайте запустим наш feature-файл, чтобы получить шаблоны для описания шагов. Поскольку мы запускаем отдельный feature-файл, мы добавим его в конце команды `cucumber`. Для разнообразия будем использовать Google Chrome.

```
$ cucumber -p sauce_c_03 features/manage_products.feature
...
1 scenario (1 undefined)
6 steps (1 skipped, 5 undefined)
0m9.701s

You can implement step definitions for undefined steps with these snippets:

Given /^I am on the Products management page$/ do
  pending # express the regexp above with the code you wish you had
end

Given /^I created a product called "([^"]*)" with a price of "([^"]*)"
dollars and a description of "([^"]*)"$/ do |arg1, arg2, arg3|
  pending # express the regexp above with the code you wish you had
end

When /^I view the details of "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then /^I should see a price of "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then /^I should see a description of "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Здесь мы сначала указали профиль `sauce_c_o3`, а затем файл, который хотим запустить. Поскольку этот тест еще не готов, он, конечно же, не был пройден. Чтобы закончить написание теста, мы создадим рядом с qedserver_home_page_steps.rb новый step-файл manage_products_steps.rb и скопируем в него фрагменты кода шагов, которые для нас сгенерировал Cucumber.

Давайте пройдемся по этому списку. Первое определение похоже на определения из нашего первого теста, в которых требовалось открыть заданную страницу.

selenium2/cucumber_test/features/step_definitions/manage_products_steps.rb

```
Given /^I am on the Products management page$/ do
  @selenium.open("/products")
end
```

Как и в предыдущем тесте, мы запускаем метод `open()`, но на этот раз просим Selenium загрузить `"/products"` вместо `"/"`.

На следующем шаге нам нужно заполнить и отправить форму и, таким образом, добавить товар в базу данных. Для этого мы просим Selenium поместить значения из объявления `given` в соответствующие поля на странице, а затем щелкнуть на кнопке Add Product.

selenium2/cucumber_test/features/step_definitions/manage_products_steps.rb

```
1  Given /^I created a product called "([^"]*)" with a price of "([^"]*)"
2    dollars and a description of "([^"]*)"$/ do |name, price, description|
3    @selenium.type("product_name", name)
4    @selenium.type("product_price", price)
5    @selenium.type("product_description", description)
6    @selenium.click("css=input[value='Add Product']")
7  end
```

Чтобы было понятнее, какие значения подаются на входе, мы изменили имена переменных в конце объявления `given`: вместо `arg1`, `arg2` и `arg3` мы будем использовать `name`, `price` и `description`. В строке 3 мы с помощью метода `type()` мы сообщаем Selenium, что содержимое переменной `name` нужно поместить в текстовое окно с идентификатором `product_name`. Аналогичные действия описываются в строках 4 и 5.

Далее, чтобы щелкнуть на кнопке Add Product, мы используем метод `click()`, которому нужно передать локатор. Эту кнопку мы найдем по ее CSS-селектору. Поиск элемента по CSS-селектору выполняется так же, как в CSS. После этого мы находим выражение вида `value='Add Product'` и определяем, является ли этот элемент тем, что нам нужно. Такой локатор будет искать на странице кнопку со значением `Add Product`.

После добавления нового товара мы должны убедиться в том, что он появился в списке, а также проверить, можно ли просмотреть сведения об этом товаре, открыв соответствующую ссылку.

selenium2/cucumber_test/features/step_definitions/manage_products_steps.rb

```
When /^I view the details of "([^"]*)"$/ do |product_name|
  @selenium.is_element_present("css=td:nth(0):contains(#{product_name})")
  @selenium.click("css=td:nth(1) > a:contains(Details)")
end
```

Как и в прошлый раз, мы заменили название переменной `arg1` на `product_name`. Так гораздо проще понять, что хранится в этой переменной. Чтобы определить, появилось ли на странице название товара, мы используем метод `is_element_present()`, которому также нужно передать локатор. На этот раз CSS-локатор будет искать элемент `<td>`, значение которого совпадает со значением переменной `product_name`. Чтобы щелкнуть на ссылке, открывающей сведения о товаре, мы с помощью CSS-локатора

найдем нужный элемент <a>, расположенный во втором дочернем элементе <td> в том же ряду.

Последние два шага выполняют похожие задачи: они проверяют, есть ли на странице заданный текст. В CTH есть несколько подходящих определений. Вариант `Then I should see "some text"` с помощью метода `text?()` находит на странице блок текста, переданный на входе. Вызвав это определение из определения шага, мы сможем немного упростить наш тест.

selenium2/cucumber_test/features/step_definitions/manage_products_steps.rb

```
Then /^I should see a price of "([^"]*)"$/ do |price|
  Then "I should see \"#{price}\""
end

Then /^I should see a description of "([^"]*)"$/ do |description|
  Then "I should see \"#{description}\""
end
```

В обоих тестах мы используем один и тот же синтаксис и добавляем нашу переменную к строке. Поскольку Cucumber будет выполнять сопоставление регулярных выражений, нам пришлось поставить перед кавычками знаки \. Теперь давайте снова запустим наши тесты Google Chrome.

```
$ cucumber -p sauce_c_03
....
3 scenarios (3 passed)
10 steps (10 passed)
0m43.184s
```

Зная, что наши тесты успешно выполняются в Google Chrome, мы можем попробовать запустить их в других браузерах: например, в Internet Explorer 7 с помощью команды `$ cucumber –p sauce_ie7_03` или в Safari с помощью команды `$ cucumber -p sauce_s_03`. После успешного выполнения тестов в других браузерах мы будем знать, что наше приложение работает в самых разных средах. А если вы хотите убедиться в правильности оформления сайта, просмотрите снимки экрана.

Дополнительные возможности

Тест, который мы создали в этом разделе, проверяет некоторые функциональные возможности нашего сервера для тестирования. Для проверки других возможностей, например удаления товара, можно также написать тесты. Кроме того, Cucumber Testing Harness можно настроить для использования на отдельном сайте.

Можно попробовать и другие возможности Sauce Labs — например, Sauce Scout. Sauce Scout позволяет обращаться через наш туннель к сайту, открытому в любом из браузеров, которые поддерживает Sauce Labs. С его помощью можно указывать браузеру, куда идти и где щелкнуть мышью; это может быть полезно во время отладки.

Альтернативой Sauce Labs может служить selenium-rc gem[1], позволяющий запускать тесты во всех браузерах, установленных на вашем компьютере. Но, как уже упоминалось выше, не стоит забывать, что этот процесс требует большого количества ресурсов.

[1] http://selenium.rubyforge.org/getting-started.html

Смотрите также

- Рецепт 33 Тестирование в браузере с помощью Selenium
- Рецепт 35 Тестирование JavaScript-кода с помощью Jasmine
- Cucumber: Разработка, управляемая поведением для тестеров и разработчиков (The Cucumber Book: Behavior-Driven Development for Testers and Developers) [WH11]

Рецепт 35. Тестирование JavaScript-кода с помощью Jasmine

Задача

Гибкость и динамическая природа JavaScript могут осложнять процесс тестирования, делая код как бы «движущейся мишенью». Использование Selenium IDE (см. «Рецепт 33. Тестирование в браузере с помощью Selenium») все равно требует ручной отладки (см. Рецепт 31 «Отладка JavaScript-кода») и не дает возможности получить точную информацию о том, какая функция является источником проблемы. Очевидно, что нам нужен полноценный фреймворк для тестирования JavaScript-кода.

Ингредиенты

- jQuery
- Jasmine[1]
- Jasmine-jQuery[2]
- Firefox[3]

Решение

Jasmine — фреймворк для тестирования JavaScript, созданный Pivotal Labs и реализующий идею разработки, управляемой поведением (Behavior-Driven Development, BDD) для JavaScript. Синтаксис Jasmine очень похож на синтаксис фреймворка для тестирования Ryby RSpec[4] (более подробную информацию о RSpec и BDD можно найти в книге «Руководство по RSpec» (The RSpec Book) [CADH09]). BDD работает по принципу «снаружи внутрь»: это значит, что в центре внимания оказывается не структура, а поведение.

Нашим первым приложением с полностью тестируемым JavaScript-кодом будет список задач, реализованный с помощью jQuery. При этом мы будем пользоваться методами разработки через тестирование (Test-Driven Development, TDD): сначала мы будем писать тест, а уже потом — код, для которого этот тест будет выполняться. Единственное отличие состоит в том, что мы будем описывать не отдельные элементы кода, а их поведение.

Для начала создадим отдельную папку для нашего приложения и распакуем в нее библиотеки для тестирования Jasmine, загруженные с GitHub. Нам также понадобится плагин Jasmine-jQuery, который нужно поместить в папку jasmine/lib/jasmine-1.0.2. Этот плагин дает нам несколько дополнительных возможностей, которые будут полезны при работе с фикстурами.

Внутри папки Jasmine вы найдете еще три папки и файл SpecRunner.html. Два файла с расширением .js, расположенные в папках spec и src, можно удалить: это примеры, которые поставляются вместе с библиотеками Jasmine.

---

[1] http://pivotal.github.com/jasmine/downloads/jasmine-standalone-1.0.2.zip
[2] https://github.com/downloads/velesin/jasmine-jquery/jasmine-jquery-1.2.0.js
[3] http://www.mozilla.com/en-US/firefox/new/
[4] http://rspec.info/

Теперь можно приступить к созданию тестов и самого приложения. Мы начнем с основы, к которой по мере необходимости будем добавлять новые элементы. Давайте создадим в папке spec файл add_todo_spec.js. Структура каталога должна выглядеть так, как показано на рис. 6.9. Чтобы лучше представлять себе, что мы собираемся делать, посмотрите на макет нашего приложения (рис. 6.10).

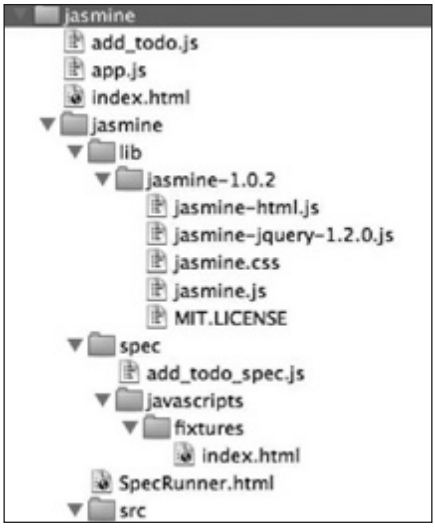

Рис. 6.9. Структура папок

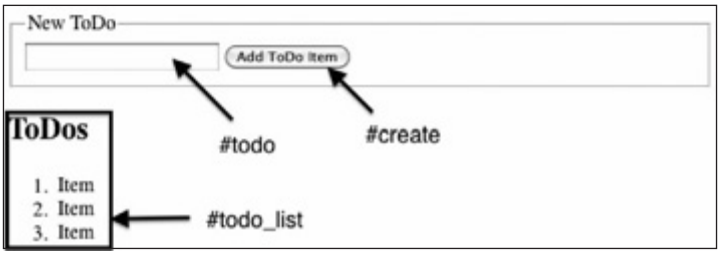

Рис. 6.10. Макет списка задач

Наш первый тест

Мы начнем с создания блока `describe()`, который служит для группировки тестов. Как видно из макета (см. рис. 6.10), основной функцией нашего приложения является добавление нового элемента в список задач. Здесь можно заметить некоторые сходства с фреймворком Ruby RSpec. У нас есть функция `describe()` с двумя аргументами: сообщением и еще одной функцией. Внутрь блока `describe()` мы добавим несколько примеров, описывающих определенные элементы поведения.

jasmine/jasmine/spec/add_todo_spec.js

```
 1  describe('I add a ToDo', function () {
 2    it('should call the addToDo function when create is clicked', function ()
{
```

продолжение ⇨

```
3    });
4    it('should trigger a click event when create is clicked.', function() {
5    });
6  });
```

Наш первый пример в строке 2 описывает поведение при нажатии кнопки `create`. В этом тесте говорится о том, что при нажатии этой кнопки приложение должно вызывать функцию добавления нового элемента. Второй пример описывает событие, которое должно вызываться при нажатии кнопки `create`. В данном случае мы хотим убедиться в том, что событие `click()` действительно вызывается.

Чтобы использовать Jasmine, нужно сообщить ему местоположение нашего теста, приложения и библиотек. Для настройки Jasmine мы внесем некоторые изменения в файл SpecRunner.html: уберем ссылки на удаленные файлы спецификации и добавим местоположение файла add_todo_spec.js.

jasmine/jasmine/SpecRunner.html

```
<!--The jQuery additional commands, for fixtures and such -->
<script type="text/javascript"
  src="lib/jasmine-1.0.2/jasmine-jquery-1.2.0.js">
</script>
<!-- include source files here... -->
<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js"
  charset="utf-8">
</script>
<script src="../add_todo.js" type="text/javascript" charset="utf-8"></script>
<!-- include spec files here... -->
<script type="text/javascript" src="spec/add_todo_spec.js"></script>
```

Чтобы запустить файлы спецификации, откройте SpecRunner.html в Firefox. Так как все окрашено в зеленый цвет, кажется, будто наши тесты успешно выполнены. Однако это не совсем так. Если посмотреть на тесты, которые мы написали, можно увидеть, что на самом деле они ничего не делают. Но нам нужна полноценная проверка. Поэтому сначала мы напишем несколько тестов, которые не будут пройдены, а затем создадим окончательную версию кода и посмотрим, как она будет выполнена.

С помощью нашего первого теста мы хотим убедиться в том, что при нажатии кнопки Create вызывается функция `addToDo()`.

jasmine/jasmine/spec/add_todo_spec.js

```
$('#create').click();
expect(ToDo.addToDo).toHaveBeenCalledWith(mocks.todo);
```

Мы хотим проверить, что событие щелчка мыши действительно запускает функцию `addToDo()`. Чтобы вызвать это событие, нам нужен HTML-код, в котором мы запустим JavaScript-код. Одним из преимуществ плагина Jasmine-jQuery является поддержка фикстур, которые позволяют создавать фрагменты HTML-кода, обеспечивая воспроизводимость наших тестов. Наше приложение состоит из формы с одним текстовым полем и кнопкой Create, под которой располагается список. Такое приложение можно смоделировать в файле фикстуры. Jasmine будет искать фикстуры в каталоге jasmine/

spec/javascripts/fixtures/ нашего приложения. Здесь мы и создадим файл index.html, который будет представлять наше приложение.

jasmine/jasmine/spec/javascripts/fixtures/index.html

```
<fieldset>
  <legend>New ToDo</legend>
  <form>
      <input type="text" id="todo"/>
    <button id="create">Add ToDo Item</button>
  </form>
</fieldset>
<h2>ToDos</h2>
<ol id="todo_list">
</ol>
```

После того как мы создали фикстуру, необходимо сообщить о ней нашим тестам. Для этого мы воспользуемся функцией Jasmine `beforeEach()`, которая будет задавать определенные настройки перед запуском каждого теста. В нашем случае ее нужно поместить внутри функции `describe()`. Для загрузки фикстуры мы будем использовать функцию `loadFixtures()`.

jasmine/jasmine/spec/add_todo_spec.js

```
beforeEach(function () {
  loadFixtures("index.html");
});
```

Поместив функцию `beforeEach()` внутри функции `describe()`, мы сообщаем Jasmine о том, что `beforeEach()` нужно запускать перед каждым тестом из данного блока `describe()`. Внутри `beforeEach()` удобно размещать любой код, использующийся во всех тестах блока.

ВОПРОС/ОТВЕТ. ПОЧЕМУ МЫ ИСПОЛЬЗУЕМ FIREFOX?

Firefox — надежный браузер с поддержкой сторонних ресурсов, который находится в арсенале большинства веб-разработчиков уже в течение многих лет. Одно из его дополнений — Firebug[1] — является своего рода «швейцарским ножом» веб-разработки. В нем есть множество инструментов для анализа и изменения разметки, JavaScript-кода и CSS. Firebug может определять время загрузки элементов страницы и сообщать вам порядок их загрузки. Он обладает огромным разнообразием средств для отладки JavaScript. Более подробно о Firebug рассказывается в рецепте 31 («Отладка JavaScript-кода»).

Мы собираемся проверить, как работает добавление нового элемента в список задач. Мы только что создали фикстуру; теперь нам нужны фиктивные данные. Mocks — это инструмент для имитации реальных данных, которые можно использовать при запуске каждого теста. Для начала мы создадим пустой объект `mocks`, к которому можно будет добавлять разные значения. Прямо перед функцией `beforeEach()` мы напишем следующее:

jasmine/jasmine/spec/add_todo_spec.js

```
var mocks = {};
```

[1] http://getfirebug.com/

Создавая глобальную переменную в самом начале теста, мы получаем объект, к которому можно добавлять функции и значения. Поскольку тест Jasmine взаимодействует с кодом приложения, использование такого объекта поможет отграничить объекты теста.

Внутри функции `beforeEach()` мы добавим к объекту `mocks` переменную `todo`. Чтобы задать значение текстового поля с помощью `todo`, можно использовать jQuery. Как мы уже знаем (см. макет приложения), наше текстовое поле должно иметь идентификатор `todo`.

jasmine/jasmine/spec/add_todo_spec.js

```
mocks.todo = "something fun";
$('#todo').val(mocks.todo);
```

Здесь мы присваиваем `todo` значение `"something fun"` и вводим эту строку в тестовое поле.

Следуя принципам TDD, сначала мы напишем тест, а уже потом код, для которого этот тест будет выполняться. Поскольку у нас еще нет никакого кода, тест не будет пройден, и на экране появится нечто похожее на рис. 6.11.

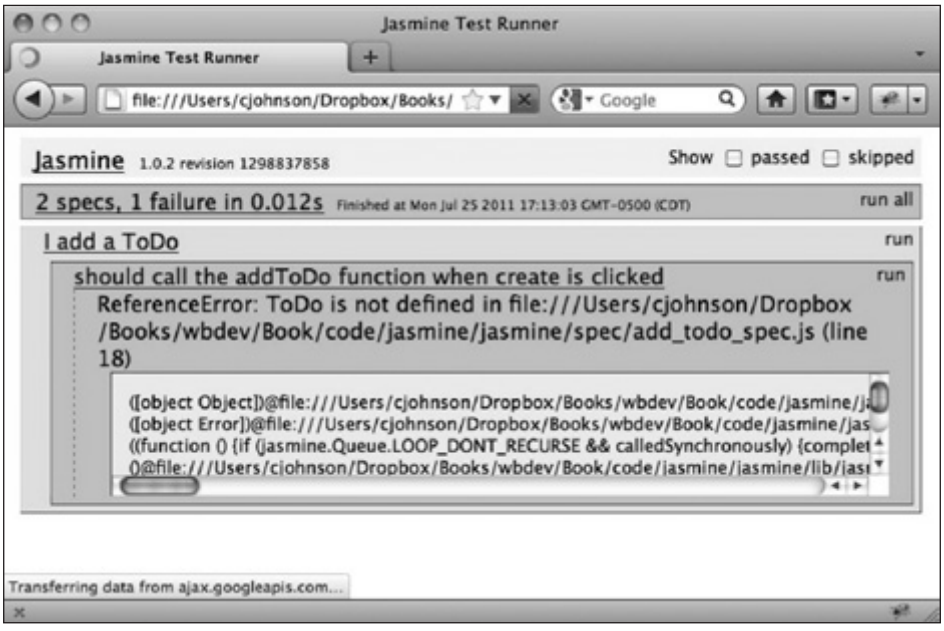

Рис. 6.11. Наш первый тест не пройден

Успешное выполнение теста

Нам нужно где-то хранить код приложения. Давайте создадим в корневом каталоге приложения файл add_todo.js. Чтобы удобно разместить наши функции и упростить процесс тестирования, мы будем использовать JavaScript-объект `ToDo`. Внутри него мы напишем три функции.

jasmine/add_todo.js

```
var ToDo = {
  setup: function(){
  },
  setupCreateClickEvent: function(){
  },
  addToDo: function(todo){
  }
};
```

После этого нам нужно добавить все функции, отвечающие за работу приложения. Начнем с функции `setup()`, которую мы будем вызывать как в приложении, так и в тестах. Она будет запускать функцию `setupCreateClickEvent()`, связывающую событие `click()` с кнопкой `create`. Когда пользователь нажмет кнопку `create`, браузер запустит событие `click()`, которое в свою очередь запустит функцию `addToDo()`.

jasmine/add_todo.js

```
var ToDo = {
  setup: function(){
    ToDo.setupCreateClickEvent();
  },
  setupCreateClickEvent: function(){
    $('#create').click(function(event){
      event.preventDefault();
      ToDo.addToDo($('#todo').val());
      $('#todo').val("");
    });
  },
  addToDo: function(todo){
    $('#todo_list').append("<li>" + todo + "</li>");
  }
};
```

В функции `setupCreateClickEvent()` мы вызываем `preventDefault()` для события, переданного функции `click()`. Этот метод отменяет отправку формы. Далее мы вызываем функцию `addToDo()`, передавая ей значение из текстового поля `todo`. После этого мы присваиваем `todo` пустую строку, чтобы потом можно было добавить следующий элемент списка. Внутри `addToDo()` мы с помощью функции jQuery `append()` добавляем в список задач новый элемент.

Давайте вернемся в файл спецификации и добавим вызов `ToDo.setup()` в блок `beforeEach()`.

jasmine/jasmine/spec/add_todo_spec.js

```
ToDo.setup();
```

Теперь перед каждым тестом будет вызываться функция `ToDo.setup()`, и событие create `click()` будет привязываться к кнопке `create` в нашей фикстуре.

Основной задачей нашего первого теста является проверка вызова функции `ToDo.addToDo()`. Для этого нам понадобится шпион Jasmine[1]. Шпион — это универсальный тестовый двойник, который можно использовать в качестве заглушки, fake-объекта или mock-объекта. Заглушка — это заранее заданный ответ на что-то; обычно это метод, возвращающий конкретное значение. Заглушка всегда возвращает одно и то же значение независимо от параметров, переданных ей на входе. Fake-объект выполняет реальные действия, но использует краткую запись; он «маскируется» под метод, хотя на самом деле является сокращенной версией оригинала. Mock-объект похож на fake-объект, но он выполняет несколько дополнительных действий: он умеет следить за тем, что происходит вокруг, — например, кто его вызывает, сколько раз и с какими параметрами. Добавив шпиона к функции, можно выполнять множество разных проверок, в частности узнать, была ли она вызвана, сколько раз она была вызвана и даже какие аргументы передавались ей при каждом вызове. Чтобы с помощью `expect(ToDo.addToDo).toHaveBeenCalledWith(mocks.todo);` выполнить проверку, не вызвав при этом саму функцию, нужно добавить `spyOn()` в начало теста. Этот шпион «захватит» функцию `AddToDo()` в момент ее вызова. После этого он проверит, что `toHaveBeenCalledWith(mocks.todo)` имеет значение `true`, то есть что функция была вызвана; при этом неважно, какое значение имеет аргумент `mocks.todo`.

jasmine/jasmine/spec/add_todo_spec.js

```
spyOn(ToDo, 'addToDo');
```

Итак, мы «шпионим» за функцией `addToDo()` объекта `ToDo`. Нам нужно проверить, что эта функция была вызвана со значением `mocks.todo`. Этот тест позволяет легко представить, как будет выглядеть его реализация.

Зная, как работают шпионы, мы можем перейти ко второму тесту. Он будет проверять, что при нажатии кнопки Create вызывается событие `click()`. Наш тест должен будет прикрепить шпиона к событию `click()`, затем щелкнуть на кнопке Create и проверить, что событие `click()` было вызвано. Давайте добавим в наш второй тест следующий код.

jasmine/jasmine/spec/add_todo_spec.js

```
spyOnEvent($('#create'), 'click');
$('#create').click();
expect('click').toHaveBeenTriggeredOn($('#create'));
```

Мы не хотим запускать функцию `click()`, но хотим убедиться в том, что она была вызвана. Поэтому здесь мы снова обратимся к Jasmine и «захватим» функцию `click()` с помощью `spyOnEvent()`.

После того как мы закончили написание тестов и связанного с ними кода, можно посмотреть, как они будут выполняться. Откройте файл SpecRunner.html в Firefox. После успешного выполнения файлов спецификации на экране появится информация, показанная на рис. 6.12.

Итак, у нас есть работающие тесты. Теперь нам нужно добавить еще один компонент и создать страницу index.html. После этого в нашем распоряжении окажется полноценный список задач.

[1] http://pivotal.github.com/jasmine/jsdoc/symbols/jasmine.Spy.html

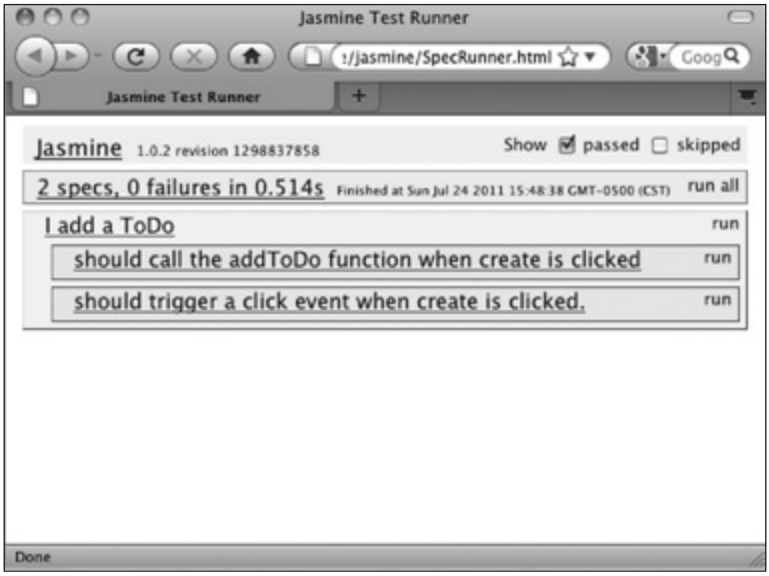

Рис. 6.12. Файлы спецификации Jasmine успешно выполнены

Последние штрихи

Напоследок давайте создадим JavaScript-файл, в который мы поместим функцию `DomReady()`. Выделение такого небольшого кода в отдельный файл позволяет нам обеспечить независимость тестов от внешних источников. В корневом каталоге проекта мы создадим файл `app.js`.

jasmine/app.js

```
$(function() {
  ToDo.setup();
});
```

Здесь мы просто вызываем функцию `ToDo.setup()`. Такой способ позволяет добиться максимальной гибкости, поскольку основная часть кода хранится в файле `add_todo.js`.

Наконец, давайте создадим файл index.html на основе нашей фикстуры. Здесь нам нужно будет подключить оба наших JavaScript-файла: app.js и add_todo.js. Начнем с простой шапки.

jasmine/index.html

```
<!DOCTYPE html>
  <head>
    <title>My Great ToDo List</title>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js">
    </script>
    <script type="text/javascript" src="app.js"></script>
    <script type="text/javascript" src="add_todo.js"></script>
  </head>
```

В качестве основной части страницы мы возьмем содержимое нашей фикстуры, так как тесты будут выполняться для точно такого же кода, как в реальном приложении.

jasmine/index.html

```
  <body>
    <fieldset>
      <legend>New ToDo</legend>
      <form action="#" method="post" accept-charset="utf-8">
        <input type="text" id="todo"/> <button id="create">Add ToDo Item</
button>
      </form>
    </fieldset>

    <h2>ToDos</h2>
    <ol id="todo_list">
    </ol>

  </body>
</html>
```

Теперь давайте откроем файл index.html в Firefox и увидим работающее приложение (рис. 6.13).

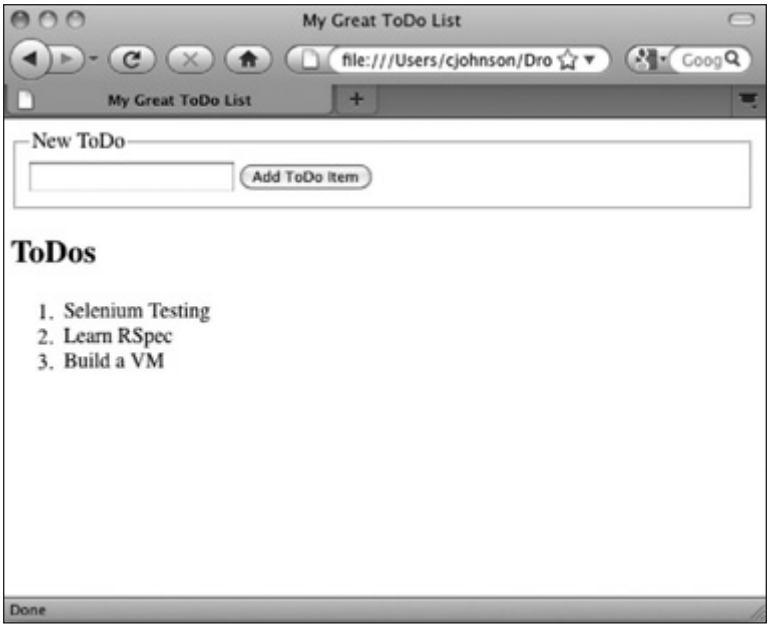

Рис. 6.13. Работающее приложение

В этом рецепте мы прошли все стадии процесса TDD, и в результате наши тесты успешно выполняются. Теперь мы можем гордиться полноценным работающим приложением.

Дополнительные возможности

Если вы хотите продолжить изучение Jasmine, попробуйте написать тесты для примеров из других рецептов этой книги — например, для рецепта 9 («Общение с веб-страницей с помощью горячих клавиш») или рецепта 11 («Отображение данных с автоматической дозагрузкой»). Можно продолжить работу над примером из этого рецепта, добавив ограничение на добавление пустых задач. Кроме того, Jasmine можно использовать вместе с CoffeeScript (см. «Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript»): это прекрасный способ получить тестируемый JavaScript-код с синтаксисом, по степени надежности сравнимым с компилятором.

Смотрите также

- Рецепт 33. Тестирование в браузере с помощью Selenium
- Рецепт 34. Тестирование с помощью Selenium и Cucumber
- Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript
- Рецепт 31. Отладка JavaScript-кода

7 Хостинг и внедрение

Мы хотим показать результат нашей работы широкой публике, однако для этого недостаточно просто разместить его в Сети: нужно еще позаботиться о безопасности. В рецептах этой главы мы расскажем о том, как внедрить сайт, а также как с помощью сервера Apache перенаправить запросы, обезопасить контент и разместить защищенный сайт.

Рецепт 36. Как разместить статический сайт на Dropbox

Задача

Вместе с нами над созданием сайта удаленно работает еще один человек. У него нет VPN-доступа к нашей группе серверов, а брандмауэр не разрешает размещать проекты за пределами нашей сети. Кроме того, нам бы хотелось получить URL, с помощью которого нашу работу могли бы оценить другие.

Ингредиенты

- Dropbox[1]

Решение

Чтобы иметь возможность совместно работать над статическими HTML-файлами и предоставлять доступ к этим файлам для внешних пользователей, мы будем использовать Dropbox. В таком случае нам не придется думать о брандмауэрах, FTP-серверах или отправке файлов по электронной почте. Так как Dropbox является платформенно-независимым, нам не нужно будет тратить время на создание специальных версий приложения для различных операционных систем, и в итоге мы выиграем в производительности.

Наша компания вместе со своим партнером AwesomeCableCo является спонсором «Дней технологии для молодежи». У AwesomeCableCo есть свой дизайнер Роб, и нам придется работать над сайтом вместе с ним. Кроме того, мы хотим время от времени показывать руководству, как продвигается наша работа.

Давайте пройдем весь процесс установки и запишем все наши действия, чтобы затем отправить Робу подробную инструкцию. Откройте сайт Dropbox и загрузите установочную программу.

После завершения установки мы сможем открыть папку Dropbox на нашем компьютере. Dropbox автоматически создает папку Public (рис. 7.1), с помощью которой мы можем предоставить всему миру доступ к нашим файлам. Внутри этой папки мы создадим папку youth_tech_days.

Рис. 7.1. Папки, которые создает Dropbox

Теперь у нас есть рабочая среда, и нам нужно пригласить в нее Роба, чтобы вместе работать над файлами, размещенными в этой папке. Щелкнув правой кнопкой мыши на

[1] http://www.dropbox.com

созданной папке, можно увидеть контекстное меню. Один из его пунктов позволяет открыть совместный доступ (рис. 7.2).

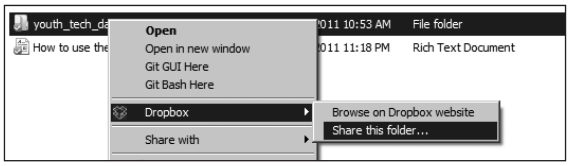

Рис. 7.2. Контекстное меню для предоставления совместного доступа

Выбрав Share This Folder, мы переходим на сайт Dropbox, чтобы завершить настройку совместного доступа (рис. 7.3). Здесь вам нужно просто ввести необходимую информацию.

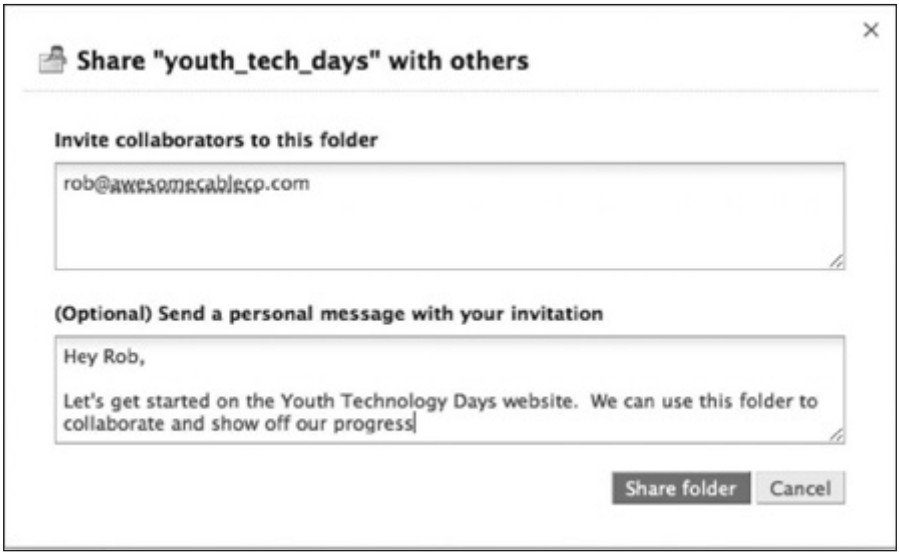

Рис. 7.3. Форма для настройки совместного доступа

Теперь мы можем скопировать в папку youth_tech_day файлы нашего сайта — они хранятся вместе с исходными кодами этой книги в папке dropbox. В результате мы должны получить каталог, изображенный на рис. 7.4.

Когда мы будем добавлять в эту папку новые файлы, они сразу же будут появляться на компьютере Роба. Если Роб внесет в один из файлов какие-то изменения, наша версия этого файла тоже обновится. Во время работы над файлом мы бы хотели иметь возможность сообщить Робу о том, что мы делаем, чтобы случайно не стереть работу Роба. Когда несколько человек одновременно работают с одним и тем же файлом, Dropbox сохраняет несколько копий этого файла и добавляет к названию файла сообщение о конфликте. В простых ситуациях этого вполне достаточно, однако при большом количестве участников лучше использовать Git (см. «Рецепт 30. Управление файлами с помощью Git»).

Теперь нам осталось только показать руководству результат нашей работы. Поскольку все файлы хранятся в папке public, доступ к ним открыт для любого пользователя

Глобальной сети, которому известен URL. Чтобы узнать адрес файла index.html, щелкните на нем правой кнопкой мыши и выберите Copy public link. Нужная ссылка окажется в буфере обмена. Чтобы проверить, как она работает, откройте ее в браузере. Наш URL будет выглядеть примерно так: http://dl.dropbox.com/u/33441336/youth_tech_days/index.html.

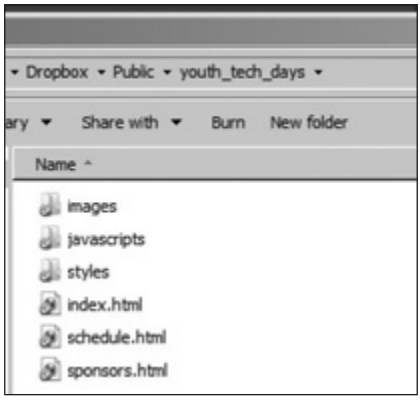

Рис. 7.4. Файлы сайта «Дни технологии для молодежи»

Это замечательный и простой способ работать с людьми, находящимися за пределами нашей компании, который позволяет следить за развитием проекта и не требует FTP-сервера, веб-сервера или VPN-соединения. С помощью URL за нашей работой могут наблюдать все заинтересованные лица, включая новых спонсоров проекта.

Дополнительные возможности

Папки, не являющиеся общедоступными, можно открыть для друзей и коллег. Также с помощью Dropbox можно сохранять резервные копии файлов и открывать их с разных компьютеров. В папке public можно разместить патч для Internet Explorer, который ваша мама почему-то не может найти. А можно предложить вашим клиентам сохранять в этой папке фотографии или любые другие файлы, которые они захотят добавить на сайт. Среди других способов применения Dropbox мы назовем следующие:

- хранение файлов, прикрепляемых к записи блога;
- создание общей папки для удобной работы с клиентами;
- использование собственного домена для общедоступного сайта;
- создание блога с помощью Jekyll и его размещение на Dropbox.

Если ваш регистратор или DNS-провайдер поддерживает перенаправление, можно использовать для вашей страницы на Dropbox более запоминающийся URL.

Смотрите также

- Рецепт 30. Управление файлами с помощью Git
- Рецепт 27. Простой блог с помощью Jekyll

Рецепт 37. Как установить виртуальную машину

Задача

Чтобы протестировать PHP-скрипты и файлы конфигурации в безопасной среде, нам нужно создать имитацию нашего рабочего сервера на локальной машине.

Ингредиенты

- VirtualBox[1]
- Образ Ubuntu 10.04 Server[2]

Решение

С помощью инструментов для виртуализации и программ с открытым исходным кодом мы можем создать рабочий сервер, который будет запускаться прямо на нашем компьютере или рабочей станции. Чтобы построить такую среду, мы будем использовать бесплатный программный продукт VirtualBox и дистрибутив Ubuntu Server Linux. После этого мы настроим веб-сервер Apache для работы с PHP, чтобы в нем можно было протестировать наши PHP-проекты.

Создание виртуальной машины

Итак, нам понадобится два программных продукта: серверная операционная система Ubuntu и инструмент для виртуализации с открытым исходным кодом VirtualBox. С помощью VirtualBox можно создать виртуальную рабочую станцию или сервер, работающий прямо в нашей операционной системе. В результате мы получим «песочницу», в которой можно делать все что угодно, не боясь испортить реальную операционную систему.

Для начала нам нужно зайти на страницу загрузки Ubuntu[3] и загрузить 32-битную *серверную* версию Ubuntu 10.04 LTS (а не самую последнюю версию). LTS расшифровывается как «Long-term Support» — «долгосрочная поддержка». Такая версия может обновляться в течение очень долгого времени, не требуя полного обновления операционной системы. Как правило, LTS-версии включают далеко не все самые современные возможности, однако они прекрасно подходят для серверов.

Пока эта программа загружается, перейдите на веб-страницу VirtualBox[4] и выберите самую последнюю версию для вашей платформы. После того как она загрузится, установите ее, используя параметры по умолчанию, а затем запустите программу VirtualBox.

Когда программа VirtualBox запустится, откройте Мастер создания виртуальной машины (Virtual Machine Wizard) с помощью кнопки Создать (New). Нашей виртуальной машине нужно будет придумать имя — например, My Web Server. Далее в качестве операционной системы выберем Linux, а в качестве версии — Ubuntu. Кроме того, нужно будет решить, сколько памяти и дискового пространства мы хотим выделить

[1] https://www.virtualbox.org/
[2] http://www.ubuntu.com/download/ubuntu/download
[3] http://www.ubuntu.com/download/server/download
[4] https://www.virtualbox.org/

нашей виртуальной машине; здесь удобно использовать параметры по умолчанию — 512 Мбайт памяти и 8 Гбайт жесткого диска.

После создания виртуальной машины мы перейдем к дополнительным настройкам с помощью кнопки Параметры (Settings). Здесь нам нужно задать тип сети: чтобы обеспечить нашему хосту доступ к серверам, выберем вместо NAT вариант Сетевой мост (Bridged) (рис. 7.5).

Рис. 7.5. Задание типа сети Сетевой мост (Bridged)

Теперь можно запустить виртуальную машину, щелкнув на кнопке Старт (Start). VirtualBox автоматически определит, что это первый запуск, и поможет нам установить Ubuntu. Сначала нам нужно будет выбрать тип носителя, с которого будет устанавливаться операционная система: это может быть как компакт-диск, так и ISO-образ. Мы выберем второй вариант, так как именно в таком виде программа была загружена с сайта. После этого запустится виртуальный сервер и начнется установка Ubuntu.

При установке Ubuntu мы можем принять все настройки, заданные по умолчанию. В качестве имени хоста можно ввести все что угодно, однако вариант по умолчанию будет также вполне уместен. Когда речь пойдет о разделении диска, лучше принять настройки по умолчанию. Далее, когда система будет предлагать вам сохранить изменения на жесткий диск, разрешайте ей это сделать: так как вы работаете на виртуальной машине, данные на настоящем жестком диске не пострадают.

Ближе к концу установки вас попросят создать учетную запись пользователя. Так как по ней мы будем входить на сервер и изменять его настройки, назовем ее «webdev». Это значение можно ввести одновременно в качестве полного имени и имени пользователя. Нам также потребуется пароль — придумайте его сами, но постарайтесь не забыть!

Когда вам предложат установить встроенное ПО, просто нажмите Далее (Continue). В конце мы сами установим все, что нужно.

После завершения процесса установки виртуальная машина перезапустится, и нам нужно будет ввести логин и пароль. Когда мы это сделаем, наш сервер наконец-то будет работать.

Настройка Apache и PHP

Благодаря менеджеру пакетов Ubuntu мы можем легко настроить сервер Apache для работы с PHP: для этого нужно войти на сервер и ввести следующие команды:

```
$ sudo apt-get install apache2 libapache2-mod-php5
$ sudo service apache2 restart
```

Первая команда устанавливает веб-сервер Apache и язык программирования PHP5, а также настраивает Apache для работы с PHP. Вторая команда перезагружает файлы конфигурации Apache, чтобы активировать новые настройки PHP. Теперь давайте настроим виртуальный выделенный сервер и скопируем файлы в каталог нашего веб-сервера.

Передача файлов на виртуальный сервер

Для работы с виртуальным сервером нам нужно установить службы, необходимые для копирования файлов.

Apache работает с файлами, размещенными в папке /var/www, но записывать файлы в эту папку может только пользователь root. Чтобы это изменить, мы воспользуемся следующей командой.

```
$ sudo chown -R webdev:webdev /var/www
```

Чтобы копировать файлы с помощью SFTP-клиента, мы установим OpenSSH; то же самое мы бы сделали, если бы работали с хостинг-компанией.

```
$ sudo apt-get install openssh-server
```

Теперь можно войти в систему, используя любой SFTP-клиент. Для этого нам понадобится IP-адрес нашей виртуальной машины. Чтобы его получить, введите следующую команду.

```
$ ifconfig eth0
```

Наш IP-адрес будет выглядеть примерно так.

```
inet addr: 192.168.1.100
```

Теперь мы можем подключиться к этому адресу с помощью SFTP-клиента. При этом нам нужно будет ввести имя пользователя и пароль, заданные во время настройки виртуальной машины. В системе Windows это можно сделать с помощью FileZilla. На компьютерах Macintosh удобно использовать Cyberduck или даже командную строку, в которой копирование файлов выполняется с помощью команды `scp`. К примеру, обычный HTML-файл, расположенный в нашем начальном каталоге, можно передать на сервер следующим образом:

```
scp index.html webdev@192.168.1.100:/var/www/index.html
```

Здесь мы задаем имя исходного файла и конечный путь. Конечный путь состоит из имени пользователя, к которому мы обращаемся, знака `@` и IP-адреса сервера, за которыми следует полный путь к папке для сохранения файла, отделенный двоеточием.

Установив виртуальную машину, мы можем начать использовать ее в качестве площадки для тестирования. Этот опыт окажется очень полезным, когда придет время внедрять код в рабочую среду.

Дополнительные возможности

Итак, на виртуальной машине можно как угодно экспериментировать с кодом, не боясь повредить операционную систему. Однако это еще не все. «Моментальные снимки» VirtualBox позволяют создавать точки восстановления, к которым можно будет вернуться в случае, если произойдет серьезная ошибка. Это особенно удобно, когда мы хотим попробовать новый метод. Кроме того, мы можем создавать специальные виртуальные машины со встроенными пакетами — например, с установленными PHP, MySQL и Apache. Такую машину можно открыть для других пользователей, чтобы им было проще выполнить тестирование своего кода.

Виртуальные машины очень удобны при внедрении реальных приложений. К примеру, с помощью «моментальных снимков» можно восстановить рабочую систему после неудачного обновления или интернет-атаки. Кроме того, копирование виртуальных машин позволяет реализовать горизонтальное масштабирование. Программные продукты с закрытым исходным кодом, такие как VMware, предлагают ряд решений для предприятий, позволяющих создавать несколько виртуальных машин на одном физическом сервере[1]. Более того, VMware включает инструменты, способные превратить физическую машину в виртуальную[2].

Смотрите также

- Рецепт 39. Как защитить Apache с помощью SSL и HTTPS

[1] http://www.vmware.com/virtualization/

[2] http://www.vmware.com/products/converter/

Рецепт 38. Изменение файлов конфигурации веб-сервера с помощью Vim

Задача

Когда нам нужно изменить файл конфигурации сервера, можно загрузить этот файл, отредактировать его на нашей рабочей станции и отправить обратно на сервер. Однако нам было бы гораздо удобнее внести изменения в файл прямо на сервере.

Ингредиенты

- Виртуальная машина, созданная в рецепте 37 («Как установить виртуальную машину»)[1]
- Текстовый редактор Vim

Решение

Многие рабочие серверы работают с системой Linux и не позволяют нам использовать графические интерфейсы. Однако все необходимые изменения можно добавить с помощью Vim — текстового редактора командной строки. Этот мощный и эффективный текстовый редактор прекрасно подходит для работы с файлами на сервере благодаря своей доступности, простоте и гибкой системе настроек.

Недавно мы внедрили сайт на рабочий сервер клиента, но забыли настроить отображение ошибки 404 «Страница не найдена» («Page Not Found»): вариант по умолчанию не устраивает нашего клиента, поэтому мы хотим добавить пользовательскую страницу. Для этого нам нужно изменить настройки Apache.

В этом рецепте мы будем использовать виртуальную машину, созданную в рецепте 37 («Как установить виртуальную машину»). Но перед тем как создать нашу страницу ошибки, давайте познакомимся с приемами редактирования в Vim.

Редактирование файлов в Vim

Сначала мы войдем в систему через консоль виртуальной машины. После этого запустим Vim, набрав в командной строке сервера следующую команду.

```
$ vim
```

Если открыть Vim без указания файла, на экране появляется информация о самом текстовом редакторе (рис. 7.6).

В Vim абсолютно *все* действия — начиная от перемещения курсора и заканчивая сохранением и открытием файлов — выполняются с помощью клавиатуры. Это возможно благодаря *режимам* Vim.

В Vim есть четыре режима: обычный режим, режим ввода, командный режим и визуальный режим.

- Обычный режим используется для перемещения по файлу и переключения режимов.
- Режим ввода применяется для ввода текста и внесения изменений в файл.

[1] Готовую виртуальную машину можно найти на http://webdevelopmentrecipes.com/

- В командном режиме мы выполняем специальные команды — например, сохранение и открытие файлов.
- Визуальный режим используется для выделения фрагмента текста, с которым нужно что-либо сделать.

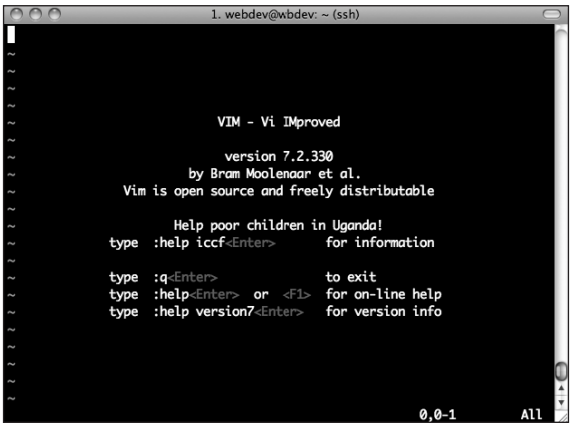

Рис. 7.6. Начальный экран Vim

Когда мы откроем Vim в первый раз, будет выбран обычный режим. Чтобы перейти в режим ввода, нажмите i; после этого в нижней части экрана появится надпись `-- INSERT --`.

В режиме ввода наберите Welcome to Vim и нажмите клавишу Enter, а затем введите Let's have some fun! В результате у вас получится такой файл:

```
Welcome to Vim
Let's have some fun!
```

Чтобы вернуться в обычный режим, нажмите кнопку Esc. В этом режиме можно перемещать курсор с шагом в один символ, используя или стрелки, или клавиши h, j, k и l. Эти клавиши соответствуют начальному расположению пальцев правой руки. Поэтому чтобы эффективно их использовать, нужно всего лишь немного попрактиковаться. Клавиша h передвигает курсов влево, а клавиша l — вправо; клавиша k перемещает курсор вверх на одну строку, клавиша j — вниз. Чтобы это запомнить, попробуйте представить, что буква j похожа на стрелку вниз.

Теперь нам нужно сохранить и закрыть этот файл. Чтобы перейти из обычного режима в командный, нажмите :. Для сохранения используется команда `:w`, которой можно передать имя файла. Поэтому чтобы сохранить наш файл под именем test.txt, достаточно вызвать следующую команду:

```
:w test.txt
```

Чтобы сохранить уже существующий файл во время редактирования, можно использовать команду `:w` без указания имени файла.

Наконец, чтобы выйти из Vim, наберите команду `:q`.

Конечно же, возможности Vim не ограничиваются этим набором простых команд, однако для наших целей — отображения пользовательской страницы с сообщением об ошибке — этого вполне достаточно.

Создание и отображение пользовательской страницы ошибки

Существует несколько способов указать, какую страницу Apache должен отображать в случае ошибки. Можно изменить основной файл конфигурации Apache, можно изменить файл конфигурации нашего сайта, а можно использовать специальный файл .htaccess: с его помощью можно задавать настройки сервера Apache для конкретного каталога, поэтому такой подход является наиболее гибким. Иногда это единственный способ задания настроек сервера, поскольку у нас может не оказаться прав для изменения других файлов конфигурации. Чтобы Apache правильно обрабатывал файлы .htaccess, необходимо задать некоторые настройки.

Сначала нужно подключить расширение Apache mod_rewrite (см. «Рецепт 41. Как переписать URL, сохранив ссылки»). Для этого в командной строке сервера введите следующую команду.

```
$ sudo a2enmod rewrite
```

Далее нам нужно разрешить переопределение параметров конфигурации для нашего сайта. Если этого не сделать, Apache просто проигнорирует все, что мы запишем в файл .htaccess. Чтобы изменить файл конфигурации для сайта по умолчанию, мы снова воспользуемся редактором Vim.

```
$ sudo vim /etc/apache2/sites-enabled/000-default
```

Используя вместо стрелок клавиши навигации (h, j, k и l), мы изменим значение переменной `AllowOverride` для каталога /var/www. Переместите курсор в конец строки `AllowOverride None` и нажмите i, чтобы перейти в режим ввода. Далее замените значение `None` и на `All`. После этого наш файл должен выглядеть так.

```
<Directory /var/www>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order allow,deny
    allow from all
</Directory>
```

Прежде чем сохранить файл, нажмите Esc, чтобы выйти из режима ввода. После этого можно сохранить файл с помощью команды `:w` и выйти из Vim с помощью команды `:q`.

Далее нам нужно создать файл, который будет использоваться в качестве страницы 404. Для этого мы перейдем в корневой каталог нашего сайта и с помощью Vim создадим новую страницу 404 под названием 404.html.

```
$ cd /var/www
$ vim 404.html
```

Эта команда создаст для нас пустой файл. С помощью клавиши i мы перейдем в режим ввода и напишем базовую разметку для нашей страницы.

```
<h1>We're sorry</h1>
<p>
  The page you are looking for can't be found.
  It may have been moved to a new location.
</p>
<p>
  You might be able to find what you're looking for
  <a href="/">here</a>.
</p>
```

После этого мы снова нажмем Esc, чтобы выйти из режима ввода. Чтобы сохранить файл и закрыть Vim одной командой, введите `:wq`.

Теперь, когда у нас есть готовая страница 404, нам нужно сообщить о ней серверу. В каталоге /var/www, где хранятся все остальные веб-файлы, мы создадим файл .htaccess.

```
$ vim .htaccess
```

После этого мы зададим расположение страницы 404 с помощью *правила конфигурации*. Нажмите клавишу i, чтобы перейти в режим ввода, и добавьте следующий текст:

```
ErrorDocument 404 /404.html
```

Расположение файла задается относительно URL сайта, а не относительно расположения файла на диске сервера.

Далее мы выйдем из режима ввода с помощью клавиши Esc, а затем сохраним файл, используя команду `:wq`. Чтобы посмотреть, что у нас получилось, откройте в браузере несуществующую страницу нашего сайта. Теперь на сайте отображается более понятный вариант страницы 404, и у нас есть немного времени, чтобы исправить наше приложение, добавив настоящую страницу 404.

Дополнительные возможности

Если назвать Vim обычным редактором, то с таким же успехом бекон можно назвать обычным мясом. Но бекон — это больше чем мясо, это очень вкусное мясо. Правильный подбор плагинов способен сделать из Vim полноценную и «очень вкусную» интегрированную среду разработки. Поскольку Vim можно установить во всех крупных операционных системах[1], вы всегда сможете использовать его на вашей рабочей машине. А чтобы оптимизировать рабочий процесс, вы сможете установить свой собственный набор плагинов[2].

После того как вы определитесь с набором плагинов, попробуйте использовать Pathogen[3]. Обычно плагины Vim устанавливаются в специальные папки, однако Pathogen предлагает хранить их в одном месте, что существенно облегчает установку обновлений. Идея очень проста: вы сохраняете плагины в папку .vim/bundles, а Pathogen сообщает Vim, что плагины хранятся в этой папке и ее подпапках. Так как самые популярные плагины Vim размещены на GitHub, многие разработчики клонируют их прямо в папку .vim/bundles с помощью Git, а чтобы установить обновления, они просто вызывают команду `git pull`.

Более подробно о том, как использовать Vim для решения самых разнообразных задач, можно узнать на VimCasts[4]. Там вы найдете деморолики, подробно описывающие работу с Vim и его плагинами.

Смотрите также

- Рецепт 37. Как установить виртуальную машину
- Рецепт 39. Как защитить Apache с помощью SSL и HTTPS
- Рецепт 41. Как переписать URL, сохранив ссылки

---

[1] http://www.vim.org/download.php
[2] http://www.vim.org/scripts/script_search_results.php
[3] http://www.vim.org/scripts/script.php?script_id=2332
[4] http://vimcasts.org/

Рецепт 39. Как защитить Apache с помощью SSL и HTTPS

Задача

Если наш сайт или приложение используют информацию о реальных людях, мы обязаны обеспечить ее безопасность. Мы должны позаботиться о том, чтобы эта информация была защищена не только во время хранения на серверах и в базах данных, но и в процессе передачи с компьютера на сервер и обратно. Поэтому мы хотим настроить наш сервер так, чтобы он подключался к браузеру через SSL.

Ингредиенты

- Виртуальная машина для тестирования с установленной системой Ubuntu
- Веб-сервер Apache с поддержкой SSL

Решение

Чтобы обезопасить веб-сервер, необходимо установить SSL-сертификаты. Рабочие сайты используют подписанные SSL-сертификаты, заверенные третьей стороной, поскольку это дает пользователям ощущение защищенности.

Однако подписанные SSL-сертификаты стоят денег, а мы не хотим платить за сертификаты, которые будем использовать только в процессе разработки. Для задач тестирования можно создавать «самоподписанные» сертификаты, заверенные самими разработчиками.

Здесь мы будем использовать виртуальную машину, созданную в рецепте 37 («Как установить виртуальную машину»)[1]. Поэтому когда нам нужно будет настроить рабочую машину, мы будем точно знать, что делать. Все команды, использующиеся в этом рецепте, мы будем запускать из консоли виртуальной машины, то есть *не* на локальной машине.

Создание самоподписанного сертификата для разработки

Процедура получения SSL-сертификата одинакова для заверенных и самоподписанных сертификатов. Сначала мы создаем запрос на сертификат. Этот запрос обычно передается третьей стороне одновременно с оплатой, после чего третья сторона отправляет обратно заверенный SSL-сертификат, который можно установить на сервере. В нашем случае мы будем играть роль обеих сторон.

Чтобы создать запрос, мы запустим виртуальную машину, войдем в консоль и введем следующее:

```
$ openssl req -new -out awesomeco.csr
```

Эта команда создает запрос на сертификат и личный ключ подписи, которому требуется кодовая фраза.

Далее нам нужно задать для нового ключа кодовую фразу. Кроме того, нас попросят ввести название компании и другую информацию. Здесь следует указывать реальные данные, особенно если вы хотите запросить ключ в центре сертификации.

---

[1] Чтобы сэкономить время, вы можете загрузить готовую виртуальную машину на http://www.webdevelopmentrecipes.com/

Такие ключи запрашивают кодовую фразу при каждом их использовании. Поэтому если мы создадим сертификат для этого ключа, нам придется вводить кодовую фразу при каждом запуске веб-сервера. Такой способ является надежным, но неудобным. К тому же в рабочей среде с такими ключами довольно трудно работать. Поэтому мы создадим ключ, не требующий пароля.

```
$ sudo openssl rsa -in privkey.pem -out awesomeco.key
```

Теперь у нас есть запрос, и мы можем сами его подписать, передав в качестве параметров и запрос, и ключ.

```
$ openssl x509 -req -days 364 -in awesomeco.csr \
-signkey awesomeco.key -out awesomeco.crt
```

Этот сертификат можно использовать в течение года.

Нам осталось скопировать сертификат и файл ключа в нужное место.

```
$ sudo cp awesomeco.key /etc/ssl/private
$ sudo cp awesomeco.crt /etc/ssl/certs
```

Теперь давайте изменим настройки сайта Apache, чтобы подключить SSL.

Настройка поддержки SSL в Apache

Нам нужно активировать на нашем сервере модуль Apache для поддержки SSL. Для этого можно либо вручную изменить список установленных модулей, либо просто ввести следующее:

```
$ sudo a2enmod ssl
```

Эта команда сделает всё за нас.

Теперь нам нужно изменить настройки Apache так, чтобы при загрузке страниц использовался SSL.

Давайте создадим для нашего SSL-сайта отдельный файл конфигурации /etc/apache2/sites-available/ssl_example и добавим в него следующий текст:

```
  <VirtualHost *:443>
  ServerAdmin webmaster@localhost
  DocumentRoot /var/www
  <Directory /var/www/>
    Options FollowSymLinks
    AllowOverride None
  </Directory>
  SSLEngine on
  SSLOptions +StrictRequire
  SSLCertificateFile /etc/ssl/certs/server.crt
SSLCertificateKeyFile /etc/ssl/private/server.key
</VirtualHost>
```

Мы создаем новый виртуальный хост на порте 443, который будет подключен ко всем адресам. `DocumentRoot` задает расположение наших веб-страниц, а раздел `directory` определяет права доступа.

Последние несколько строк устанавливают SSL-соединения. Здесь мы активируем поддержку SSL и указываем, что все правила должны строго выполняться. Далее мы проверяем, что нам известно расположение подписанного сертификата и ключа.

Сохранив новый файл конфигурации, мы активируем его и обновляем настройки Apache.

```
$ sudo a2ensite ssl_example
$ sudo /etc/init.d/apache2 restart
```

Теперь мы можем зайти на URL нашего сайта через SSL. Правда, при этом браузер выведет на экран предупреждения, поскольку наш сертификат не является безопасным с точки зрения обычного пользователя. На самом деле это действительно так. Если бы кто угодно мог создавать сертификаты, которые каждый браузер признает надежными, такая защита имела бы мало смысла. По-настоящему надежный сертификат можно получить только при помощи третьей стороны — и здесь в игру вступает поставщик сертификатов.

Как работать с поставщиком сертификатов

Чтобы наши пользователи не думали, что мы пытаемся украсть информацию об их кредитной карте или сделать еще что-то нехорошее, нам нужно получить надежный сертификат. Сначала мы сгенерируем запрос и ключ точно так же, как мы это делали для самоподписанного сертификата. Затем мы отправим этот запрос в центр сертификации одновременно с оплатой и получим сертификат, готовый к установке, а также кое-какие инструкции.

Некоторые центры сертификации не ограничиваются выдачей сертификата, убирающего сообщение об ошибке, в обмен на деньги. Иногда они действительно проверяют, является ли ваша организация законной. Если пользователь откроет сведения о сертификате в браузере, он увидит эту информацию и с большей вероятностью сочтет этот сертификат надежным. Конечно, такая защита потребует дополнительных затрат, но иногда она действительно имеет смысл.

Существует много разных центров сертификации. Наиболее известными и надежными из них являются Thawte[1] и VeriSign[2], однако мы рекомендуем вам рассмотреть разные варианты и выбрать из них наиболее подходящий. Если вы работаете с хост-провайдером, можно воспользоваться его услугами.

Дополнительные возможности

На самом деле существует несколько типов SSL-сертификатов. Действие обычного сертификата распространяется на один сервер, а специальный сертификат «wildcard» можно установить на всех серверах одного домена. Wildcard-сертификаты гораздо дороже обычных.

Третий тип сертификатов — SNI (Server Name Indication, «указание имени сервера») — наиболее дешевый вариант, однако он работает только в самых современных браузерах и операционных системах. Такие сертификаты хорошо подходят для внутреннего использования, то есть только в том случае, когда мы можем сами решать, в каких браузерах будут работать наши пользователи. Во всех остальных случаях лучше ограничиться более традиционными хост- или IP-сертификатами.

Смотрите также

- Рецепт 37. Как установить виртуальную машину.

[1] http://www.thawte.com/

[2] https://www.verisign.com/

Рецепт 40. Как обезопасить контент

Задача

Иногда требуется ограничить доступ к определенным файлам и папкам, хранящимся на сервере, и нам нужно найти простой способ это сделать. При этом мы хотим максимально упростить процесс аутентификации пользователей, имеющих доступ к этому содержимому.

Ингредиенты

- ❑ Сервер разработки с установленным Apache

Решение

Когда мы размещаем данные на сервере, они доступны всем желающим. Но если мы не хотим открывать всему миру важные документы, нам придется защитить их, добавив аутентификацию пользователя. С помощью специальных файлов конфигурации Apache позволяет указать, какие файлы и папки нельзя передавать без аутентификации. В этом рецепте мы расскажем о том, как создавать такие файлы конфигурации.

Простая HTTP-аутентификация

При передаче файлов Apache всегда ищет файл .htaccess. Этот файл содержит настройки для данной конкретной папки. С его помощью можно защитить файл паролем, запретить передачу файла заданному типу пользователей, настроить перенаправления и сообщения об ошибках, а также многое другое.

Мы начнем с создания файла, требующего аутентификации. Если у вас еще нет сервера для тестирования, установите его, следуя указаниям из рецепта 37 («Как установить виртуальную машину»). После того как вы войдете на сервер разработки, проверьте, что Apache запущен.

```
$ sudo service apache2 restart
```

Теперь мы можем перейти к добавлению аутентификации. Для обычной HTTP-аутентификации нам потребуется файл с именами пользователей и паролями, для которых доступ к файлу открыт. С помощью команды `htpasswd` мы можем сгенерировать имя пользователя с зашифрованным паролем. Давайте создадим пользователя и сохраним получившийся файл в начальном каталоге.

```
$ htpasswd -c ~/.htpasswd webdev
New password:
Re-type new password:
Adding password for user webdev
```

При вызове `htpasswd` мы указываем расположение файла и наше имя пользователя; нам также нужно ввести пароль для шифровки. Флаг `-c` означает, что файл нужно создать, если он не был создан ранее. С помощью команды `cat` можно посмотреть, что сейчас находится в этом файле.

```
$ cat .htpasswd
webdev:mT8fQuzEhguRg
```

ВОПРОС/ОТВЕТ. ГДЕ НУЖНО ХРАНИТЬ ФАЙЛЫ .HTPASSWD?

На большинстве виртуальных хостов вы можете работать только в вашем начальном каталоге. Это значит, что корневой каталог документа для Apache выглядит примерно так: /home/webdev/mywebsite.com/public_html. Поскольку разработчикам часто приходится размещать на одном хосте несколько сайтов, очень удобно хранить эти сайты в отдельных папках. Поэтому надежнее всего хранить каждый отдельный файл .htpasswd в папке того сайта, к которому он относится. Например, чтобы сгенерировать такой файл для mywebsite.com, нужно выполнить следующую команду?

```
$ htpasswd -c ~/mywebsite.com/.htpasswd webdev
```

Так информация о пользователях разных сайтов будет храниться в разных местах.

После создания пользователя мы можем заняться настройкой доступа к нашим папкам. Давайте перейдем в корневой каталог документа и создадим там файл .htpasswd.

```
$ cd /var/www
$ touch .htaccess
```

Теперь откроем наш файл в текстовом редакторе и добавим несколько директив, ограничивающих доступ к корневой папке:

```
AuthUserFile /home/webdev/.htpasswd
AuthType Basic
AuthName Our secure section
Require valid-user
```

Создав этот файл в самом верхнем каталоге, мы ограничили доступ ко всем документам нашего сервера. Попробуйте открыть в браузере http://192.168.1.100/. Вы увидите модальное диалоговое окно аутентификации (рис. 7.7).

Таким образом, благодаря HTTP-аутентификации Apache мы можем легко защитить контент, хранящийся на сервере.

Рис. 7.7. Диалоговое окно HTTP-аутентификации

Запрет на использование изображений другими сайтами

Если хостинг-план обходится нам в копеечку, мы просто обязаны подумать о пропускной способности и нагрузке на сервер. Кроме того, мы не хотим, чтобы кто-то

использовал наши изображения, не имея на это прав. К счастью, в файл .htaccess можно добавить специальное правило, запрещающее использование наших изображений на других сайтах.

Сначала нам нужно подключить модуль Apache mod_rewrite (более подробно об этом рассказывается в рецепте 41 («Как переписать URL, сохранив ссылки»)). С его помощью мы заменим настоящее изображение на специальную картинку, сообщающую об отсутствии изображения.

```
$ sudo a2enmod rewrite
```

Теперь нам нужно добавить правило, которое будет переписывать URL, отправляя запрашивающей стороне отсутствующее изображение. Давайте откроем файл .htaccess и добавим в него следующий текст:

```
RewriteEngine on
RewriteCond %{HTTP_REFERER} !^http://(www\.)?mywebsite.com/.*$ [NC]
RewriteRule \.(jpg|png|gif)$ - [F]
```

Первая строка подключает mod_rewrite. Далее следует условие, согласно которому правило замены будет применяться только к тем сайтам, URL которых не совпадает с нашим. В последней строке мы создаем правило замены: оно будет находить все запросы с расширением изображения на конце. Флаг `[F]` сообщает Apache о том, что такие URL являются запрещенными.

Теперь при любом запросе на изображение сервер будет возвращать картинку, сообщающую об отсутствии изображения, вместо самого изображения.

Дополнительные возможности

Когда нам нужно ограничить доступ к серверу, существует много способов скрыть информацию и контент. Кроме парольной защиты и правил замены можно также блокировать пользователей по IP-адресу или даже сайту, с которого они обращаются к нашему контенту. С помощью файлов конфигурации Apache мы можем защитить наш контент самыми разными способами. Более подробно о том, как работают замены, рассказывается в Рецепте 41 «Как переписать URL, сохранив ссылки»). Можно также заглянуть в инструкцию Apache по .htaccess[1].

Смотрите также

- Рецепт 38. Изменение файлов конфигурации веб-сервера с помощью Vim
- Рецепт 41. Как переписать URL, сохранив ссылки
- Рецепт 37. Как установить виртуальную машину
- Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake

[1] http://httpd.apache.org/docs/current/howto/htaccess.html

Рецепт 41. Как переписать URL, сохранив ссылки

Задача

Мы хотим переделать наш сайт, используя новую CMS. Понятно, что при этом изменятся все наши URL. Наши страницы пользуются большой популярностью на внешних сайтах, и мы не хотим потерять этих пользователей. Составить список сайтов, ссылающихся на наши страницы, и попросить их обновить ссылки — далеко не самая лучшая идея, к тому же это займет достаточно много времени. Идея оставить старые страницы, добавив в них ссылки на новые, ничуть не лучше. Нам нужно найти способ перенаправить пользователей на новые URL с минимальным количеством затрат.

Ингредиенты

- Сервер Apache
- mod_rewrite

Решение

С помощью сервера Apache и mod_rewrite мы можем заставить сервер отправлять заданный файл вместо другого файла, на который поступает запрос. Это дает нам возможность самим решать, что будут загружать посетители нашего сайта. А чтобы нам не пришлось создавать отдельное правило замены для каждой страницы, мы можем использовать регулярные выражения. Кроме того, мы можем добавлять заголовки, по которым поисковые системы будут направлять пользователей по правильному адресу.

В этом рецепте мы будем работать на виртуальной машине с установленным Apache (см. «Рецепт 37. Как установить виртуальную машину»). Если вы пользуетесь услугами хостинг-компании, вам, возможно, придется обратиться к ее администратору с просьбой подключить mod_rewrite.

Сначала нам нужно проверить, что в системе установлен mod_rewrite. Для этого проще всего создать страницу phpinfo.php, добавив в нее одну строку кода.

```
<?php phpinfo(); ?>
```

Этот файл нужно сохранить на сервере в одной папке с остальными веб-страницами, а затем открыть в браузере. На экране появится вся информация о среде, в которой мы работаем. Модуль mod_rewrite мы будем искать в строке Loaded Modules раздела apache2handler (рис. 7.8). (Если вы выполняете такую проверку на рабочем сервере, лучше сразу же удалить этот файл, так как он содержит ценную информацию о конфигурации сервера, которую следует держать в секрете.) Если этот модуль окажется в списке, можно двигаться дальше. Если нет, необходимо подключиться к серверу по SSH и установить его, запустив в терминале команду `sudo a2enmod rewrite`. После этого нужно открыть /etc/apache2/sites-available/default и заменить строку `AllowOverRide None` в разделе `<Directory /var/www/>` на `AllowOverRide All`. После этого перезапустите Apache с помощью приведенной ниже команды, и mod_rewrite будет готов к использованию:

```
sudo /etc/init.d/apache2 restart
```

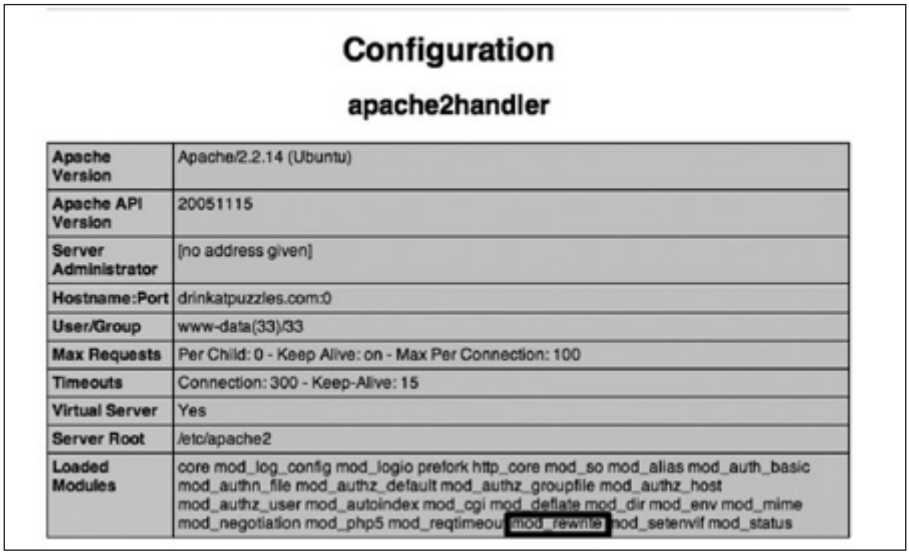

Рис. 7.8. Просмотр списка подключенных модулей с помощью phpinfo()

Чтобы знать, как обрабатывать запросы и куда их перенаправлять, модуль mod_rewrite использует файл .htaccess.

```
RewriteEngine on
RewriteRule ^pages/page-2.html$ pages/2
```

Этот файл .htaccess описывает замену URL для одной страницы, однако этого вполне достаточно, чтобы проверить, как работает перенаправление. В первой строке мы активируем `RewriteEngine`, чтобы можно было использовать mod_rewrite. Во второй строке мы создаем правило замены (`RewriteRule`), состоящее из трех частей. Сначала мы создаем новое правило замены, а затем с помощью регулярного выражения задаем общий вид запросов, для которых необходимо выполнить замену. Наконец, мы указываем, что нужно будет отправить в ответ на данный запрос. Это правило сообщает Apache о том, что по запросу pages/page-2.html нужно передать контент со страницы pages/2. При этом пользователь будет уверен, что он все еще на странице pages/page-2.html. Таким образом, вместо старой страницы (рис. 7.9) мы передаем новую (рис. 7.10), и в итоге посетители нашего сайта видят новую страницу со старым URL (рис. 7.11).

Рис. 7.9. Старая страница до применения mod_rewrite

Рис. 7.10. Новая страница

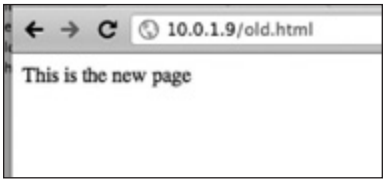

Рис. 7.11. Новая страница, отображаемая по старому URL с помощью mod_rewrite

Итак, перенаправление работает. Но разве нам хочется создавать отдельное правило для каждой страницы? Чтобы избавить себя от такой необходимости, мы воспользуемся регулярными выражениями. Предположим, что мы внедряем новую версию сайта. Раньше наши URL были устроены по принципу pages/page-2.html, однако новая CMS использует формат pages/2.

```
RewriteRule pages/page-(\d+) pages/$1 [L]
```

ВОПРОС/ОТВЕТ. ПОЧЕМУ Я НЕ ВИЖУ ФАЙЛА .HTACCESS?

То, что вы не видите файла .htaccess, вовсе не означает, что его нет. Файлы, начинающиеся с «.», могут быть скрыты, так как обычно они являются системными файлами или файлами конфигурации. Если включить в браузере отображение скрытых файлов, вы будете их видеть. Чтобы просмотреть полный список файлов, наберите команду `ls -la` в терминале OS X или Linux или команду `dir /a` в Windows.

Следуя этому правилу, сервер будет находить в URL первую последовательность цифр, следующую после pages/page-, и с ее помощью определять путь к новой странице. При получении запроса на страницу pages/page-3 сервер передаст pages/3, а в ответ на запрос pages/678.html он передаст сам файл pages/678.html, так как этот запрос не подходит под регулярное выражение. Последний параметр — [L] — отменяет использование других правил в случае, если совпадение было найдено.

Теперь новый контент отображается по старому URL. Но на самом деле это можно сделать по обоим адресам — pages/page-2.html и pages/2. Такой вариант нам не очень нравится, поскольку непонятно, на какую страницу должны указывать ссылки; к тому же это вызывает трудности с обновлением страниц. Поэтому мы хотим полностью перенаправить браузер на новые URL и сделать так, чтобы роботы поисковых систем обновили свои данные о страницах.

Для этого мы снова откроем .htaccess и добавим к правилу замены параметр `R=301`. Теперь при получении запроса по старому URL Apache будет отправлять заголовок 301, таким образом сообщая, что запрашиваемый ресурс был перемещен навсегда. Вместе с ним будет передаваться новый URL (заданный в .htaccess), чтобы браузеры и поисковые роботы могли перейти на новый адрес и получить доступ к искомой информации.

```
RewriteRule pages/page-(\d+) pages/$1 [R=301,L]
```

Теперь при переезде на новый сайт мы можем легко отказаться от старой структуры контента, не опасаясь, что наши страницы будут недоступны по старым ссылкам. Для этого нам нужно просто написать несколько регулярных выражений и правил замены.

Дополнительные возможности

А как с помощью mod_rewrite и .htaccess перенаправить запросы на новый домен? Учитывая, что в правиле замены можно задать полный URL, как будет выглядеть правило, перенаправляющее пользователей с a.com на b.com? А что, если один из разделов сайта переедет на субдомен?

Что, если мы захотим изменить язык сервера с PHP на Ruby on Rails? Что нужно будет сделать, чтобы сохранить все наши URL /display.php?term=foo&id=123, загружая контент из term/foo и term/123? Если нам удастся это сделать, никто даже не узнает, что мы изменили серверную часть.

Смотрите также

- Рецепт 38. Изменение файлов конфигурации веб-сервера с помощью Vim
- Рецепт 37. Как установить виртуальную машину
- Рецепт 39. Как защитить Apache с помощью SSL и HTTPS

Рецепт 42. Автоматизированное внедрение статического сайта с помощью Jammit и Rake

Задача

В процессе создания статических сайтов разработчики обычно используют инструменты для передачи веб-страниц и связанных с ними документов на рабочий сервер; одним из таких инструментов является FTP. Для маленьких сайтов этот метод работает хорошо, однако при большом количестве файлов мы уже не можем выполнять все необходимые действия вручную. Иначе велика вероятность того, что мы просто забудем скопировать какой-нибудь файл или скопируем его не туда, куда нужно. Кроме того, в последнее время все более популярной становятся идея объединения ресурсов в пакеты — например, объединения нескольких JavaScript-файлов в один сжатый файл. Сейчас такие процессы можно легко автоматизировать. Именно это мы попытаемся сделать в данном рецепте, стараясь создать гибкую и удобную реализацию.

Ингредиенты

- Виртуальная машина, созданная в рецепте 37 («Как установить виртуальную машину»).[1]
- Jammit[2]
- Guard[3]
- Rake[4]

Решение

Как и все веб-разработчики, мы тратим массу времени на автоматизацию процессов, выполняемых нашими пользователями и клиентами. Поэтому вполне разумно потратить немного времени и на автоматизацию наших собственных процессов. Практически в каждой командной оболочке предусмотрен язык сценариев, с помощью которого можно автоматизировать внедрение сайта. Однако мы решили воспользоваться инструментами, основанными на языке Ruby, поскольку они работают как в Windows, так и в OS X и Linux.

Компания AwesomeCo собирается распространить акцию «товар дня» на другие регионы. Поэтому нас попросили разработать простой сайт для сбора электронных адресов пользователей, желающих получить информацию о появлении новой услуги в их регионе. В итоге наш сайт должен выглядеть примерно так, как показано на рис. 7.12.

Мы хотим создать прототип этого сайта. Для объединения и сжатия JavaScript-файлов и таблиц стилей мы будем использовать специальный инструмент под названием Jammit. Далее с помощью инструмента Rake мы напишем универсальный скрипт для передачи на сервер обновленной версии сайта. Для начала давайте посмотрим, как разработать проект с учетом идеи управления ресурсами.

---

[1] Готовую виртуальную машину можно найти на http://webdevelopmentrecipes.com/
[2] http://documentcloud.github.com/jammit/
[3] https://github.com/guard/guard
[4] https://github.com/jimweirich/rake

Рис. 7.12. Наша целевая страница

Как улучшить производительность за счет объединения ресурсов в пакеты

Для отображения веб-страницы с двумя вызовами JavaScript-кода, одной таблицей стилей и одним изображением требуется отправить на сервер пять запросов. Сначала браузер загружает саму страницу, а затем запрашивает у сервера все остальные ресурсы. Некоторые браузеры не могут отправлять на один и тот же сервер более двух запросов одновременно. Вместо того чтобы подключать несколько JavaScript-файлов, мы могли бы объединить их в один. Еще один способ снизить время загрузки — *сократить* этот файл, то есть убрать комментарии и лишние пробелы. Таким образом мы уменьшим объем данных, передаваемых клиенту. После этого наш объединенный и сокращенный файл нужно подключить к веб-странице.

Чтобы сохранить все наши тщательно продуманные отступы, комментарии и структуру файлов, сжатие будет выполняться автоматически в процессе разработки; при этом на рабочий сервер будут отправляться только сокращенные версии файлов. Нечто похожее мы уже делали в рецепте 29 («Как улучшить JavaScript-код с помощью CoffeeScript»). На этот раз всю работу за нас сделает Jammit.

С помощью Jammit уменьшить CSS- и JavaScript-файлы очень просто. Для этого нужно создать файл конфигурации с указанием входных и выходных файлов. Все остальное Jammit сделает сам. Итак, давайте создадим нашу целевую страницу.

Чтобы установить Jammit, мы вызовем из командной строки команду `gem`, которая поставляется вместе с Ruby. Если на вашей машине Ruby еще не установлен, загляните в Приложение 1 «Установка Ruby», прежде чем двигаться дальше. Мы также установим Guard и плагин Jammit; с их помощью мы сможем сделать так, чтобы при изменении основных файлов Jammit обновлял наши таблицы стилей и JavaScript-файлы.

```
$ gem install jammit guard guard-jammit
```

Теперь все необходимые инструменты установлены, и мы можем заняться нашей страницей.

Создание целевой страницы

В папке нашего проекта мы создадим папку для JavaScript-файлов, папку для таблиц стилей и папку `public` для файлов, которые собираемся отправить на рабочий сервер.

```
$ mkdir public
$ mkdir javascripts
$ mkdir stylesheets
```

Вместо того чтобы загружать библиотеку jQuery с сервера Google, мы будем хранить ее вместе с остальными ресурсами. Поэтому сейчас нам нужно загрузить jQuery и поместить ее в папку javascripts.

Для отправки адреса электронной почты пользователя на сервер через Ajax нашей форме понадобится JavaScript-код. Давайте создадим для него файл javascripts/form.js.

Теперь добавим в файл public/index.html базовую разметку для целевой страницы.

static/deploy/public/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>AwesomeCo Deals</title>
    <link rel="stylesheet" href="assets/app.css" type="text/css">
    <script type="text/javascript" src="assets/app.js"></script>
  </head>
  <body>
  </body
</html>
```

Обратите внимание на раздел <head>: в нем мы подключаем CSS- и JavaScript-файлы из папки assets, а не из папок javascripts и stylesheets. Дело в том, что эти файлы Jammit создаст сам, объединив ресурсы из папок javascripts и stylesheets. А нам нужно всего лишь объяснить Jammit, чего мы хотим.

Jammit будет искать файл под названием config/assets.yml, так что давайте добавим его в наш проект. В нем мы зададим выходные файлы и соответствующие им входные.

static/deploy/config/assets.yml

```
stylesheets:
  app:
    - stylesheets/style.css

javascripts:
  app:
    - javascripts/jquery-1.7.min.js
    - javascripts/form.js
```

Далее нам нужно настроить Guard: он должен следить за изменениями файлов в папках javascripts и stylesheets. Для этого мы создадим файл Guardfile со следующим текстом.

static/deploy/Guardfile

```
guard 'jammit' do
  watch(/^stylesheets\/(.*)\.css/)
  watch(/^javascripts\/(.*)\.js/)
end
```

Теперь давайте запустим Guard, чтобы он начал создавать сокращенные версии файлов.

```
$ guard
```

Далее мы добавим в файл index.html разметку для нашей формы.

static/deploy/public/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>AwesomeCo Deals</title>
    <link rel="stylesheet" href="assets/app.css" type="text/css">
    <script type="text/javascript" src="assets/app.js"></script>
  </head>
  <body>
    <div id="container">
    <h1><span class="name">AwesomeCo Deals</span> is coming to your area!</h1>

    <form method="post" action="">
      <p>
        Sign up to be notified when we're ready to launch and be one of the
        first in your area to get in on the action!
      </p>
      <div>
        <label>
          Enter your email
          <input type="email" placeholder="email@example.com">
        </label>
        <input type="submit" value="Sign Me Up">
      </div>
    </form>
  </body
</html>
```

После того как пользователь введет свой адрес электронной почты и отправит форму, мы перехватим это событие и отправим результат через Ajax. После этого мы скроем нашу форму и отобразим сообщение с подтверждением. Сейчас мы не будем подробно разбирать процесс отправки через Ajax. В файл javascripts/form.js мы добавим следующий код.

static/deploy/javascripts/form.js

```
 (function() {
  $(function() {
    return $("form").submit(function(event) {
      var element;
      event.preventDefault();
      element = $("<p>You've been added to the list!</p>");
      element.insertAfter($(this));
      return $(this).hide();
    });
  });
}).call(this);
```

После того как мы сохраним этот файл, Guard запустит Jammit, который обновит public/assets/app.js. Теперь jQuery и код формы будут храниться в одном сжатом файле.

Теперь нам осталось добавить самую простую таблицу стилей в stylesheets/style.css. Сначала мы выровняем страницу по центру и зададим размер шрифтов.

static/deploy/stylesheets/style.css

```
#container {
  width: 960px;
  margin: 0px auto;
  text-align: center;
  box-shadow: 5px 5px 5px #ddd;
  border: 1px solid #ddd;
}
#container h1 {
  font-size: 72px;
}
#container h1 span.name {
  color: #900;
  display: block;
}
#container p {
  font-size: 24px;
}
```

Далее мы изменим границы и размер текста для полей формы.

static/deploy/stylesheets/style.css

```
#container form {
margin-bottom: 20px;
}
#container input, #container label {
height: 50px;
font-size: 36px;
}
#container input {
border: 1px solid #ddd;
}

#container input[type=submit] {
background-color: #900;
color: #fff;
}
```

Открыв страницу index.html в браузере, вы увидите, что все прекрасно работает. Следующим нашим шагом будет внедрение сайта; здесь мы попробуем применить те же принципы, что и при оптимизации процесса разработки.

Автоматизированное внедрение с помощью Rake

Rake — это инструмент командной строки, написанный на Ruby, который прекрасно подходит для решения задач автоматизации. С его помощью мы запакуем наши ресурсы и передадим все необходимые файлы на сервер. В качестве сервера мы будем исполь-

зовать виртуальную машину, созданную в рецепте 37 («Как установить виртуальную машину»).

Так же как и Jammit, мы установим Rake через оболочку.

```
$ gem install rake
```

Задачи мы определим в файле Rakefile. При этом можно использовать любую Ruby-библиотеку и даже обращаться к другим программам командной строки. Обычно задача выглядит примерно так:

```
desc "remove all .tmp files from this folder"
task :cleanup do
  FileUtils.rm_rf "*.tmp"
end
```

Этот код описывает простую задачу под названием `cleanup`, которая удаляет из текущего каталога все файлы с расширением .tmp. Чтобы операция удаления выполнялась на любой платформе, мы использовали библиотеку FileUtils. Таким образом, эта задача будет работать в любой операционной системе, в которой установлен Ruby. Чтобы ее запустить, нужно ввести в оболочке следующее:

```
$ rake cleanup
```

Перед определением задачи располагается строка `desc`, которая формулирует эту задачу. Список задач в Rakefile можно просмотреть с помощью такой команды:

```
$ rake -T
```

В этом списке будут отображаться только те задачи, перед определением которых есть строка `desc`. Поэтому мы рекомендуем вам всегда добавлять к задачам такие строки.

В нашем проекте мы создадим файл Rakefile с двумя задачами. Первая из них будет с помощью простого Ruby-скрипта загружать Guardfile и выполнять все его задачи.

static/deploy/Rakefile

```
task :build do
  require 'guard'
  Guard.setup
  Guard::Dsl.evaluate_guardfile
  Guard::guards.each{|guard| guard.run_all}
end
```

Следующая задача будет копировать папку public в папку /var/www на виртуальной машине. Для передачи файлов мы будем использовать Ruby-библиотеку Net::SCP, которая работает в Windows, OS X и Linux. Будем считать, что наш сервер расположен на 192.168.1.100 и что в качестве имени пользователя и пароля задано слово «webdev»[1].

static/deploy/Rakefile

```
desc "Deploy the web site to the dev server"
task :deploy => :build do
  require 'net/scp'
  server = "192.168.1.100"
```

[1] Чтобы узнать IP-адрес сервера, нужно набрать в его консоли команду `ifconfig`

```
  login = "webdev"

  Net::SCP.start(server, login, {:password => "webdev"} ) do |scp|
    scp.upload! "public", "/var/www", :recursive => true
  end
end
```

Определение этой задачи выглядит несколько иначе. С помощью символа `=>` (любители Ruby называют его «hashrocket») мы указываем, что эта задача зависит от предыдущей. При ее запуске задача `build` будет запускаться автоматически.

```
$ rake deploy
```

Эта команда передаст наш код на сервер. Чтобы посмотреть, как это работает, откройте в браузере страницу http://192.168.1.100/index.html. Позже, когда нам нужно будет отправить этот код на рабочий сервер, мы просто изменим данные, необходимые для входа в систему.

ВОПРОС/ОТВЕТ. КАК НАСЧЕТ ВНЕДРЕНИЯ НА СЕРВЕРАХ WINDOWS?

Для этого можно установить на сервере Windows пакет OpenSSH[1] и использовать те же самые скрипты, которые мы создаем в этом рецепте. Если такой вариант вам не подходит, попробуйте подключить диски сервера к вашей машине в качестве сетевых дисков и просто скопировать на них файлы; тогда вам не придется использовать SCP. Мы проверили оба этих варианта — они прекрасно работают. Помните, что процесс внедрения лучше автоматизировать в любом случае, независимо от платформы.

```
$ rake deploy
```

МОЖНО ЛИ НЕ УКАЗЫВАТЬ В СКРИПТЕ ПАРОЛЬ, ЕСЛИ Я ИСПОЛЬЗУЮ SSH-КЛЮЧ?

В рецепте 30 («Управление файлами с помощью Git») мы говорили о том, как создавать SSH-ключи. Если вы передаете на сервер SSH-ключ, вы можете убрать из скрипта для внедрения ту часть, которая начинается с `password =>`. Если вы не указываете пароль, Ruby-библиотека SCP автоматически использует ваши SSH-ключи. Такой способ является наиболее безопасным.

Дополнительные возможности

Нам удалось организовать процесс внедрения, и теперь можно двигаться дальше. Поскольку мы уже используем Guard, нам не составит труда подключить Sass и CoffeeScript: для этого нужно просто установить одноименные библиотеки и соответствующие им плагины Guard.

```
$ gem install coffee-script guard-coffeescript
$ gem install sass guard-sass
```

С помощью Sass можно создать таблицы стилей и поместить их в файл sass/style.scss. JavaScript-код для формы можно переделать, используя CoffeeScript, и сохранить новый код в coffeescripts/form.coffee. После этого вам нужно будет внести изменения в файл Guardfile, так чтобы он поместил сгенерированные CSS- и JavaScript-файлы во временный каталог и начал следить за их изменениями.

[1] http://sshwindows.sourceforge.net/

static/sassandcoffee/Guardfile

```
guard 'sass', :input => 'sass', :output => 'tmp'
guard 'coffeescript', :input => 'coffeescripts', :output => 'tmp'

guard 'jammit' do
  watch(/^tmp\/(.*)\.css/)
  watch(/^tmp\/(.*)\.js/)
  watch(/^stylesheets\/(.*)\.css/)
  watch(/^javascripts\/(.*)\.js/)
end
```

Наконец, вам нужно будет изменить файл конфигурации Jammit: теперь нам нужно объединить сгенерированную таблицу стилей и JavaScript-файлы.

```
stylesheets:
  app:
    - tmp/style.css

javascripts:
  app:
    - javascripts/jquery-1.7.min.js
    - tmp/form.js
```

Такой подход позволяет объединить обычные CSS и JavaScript с CoffeeScript и Sass. Это значит, что теперь в нашей цепи автоматизированных переходов могут участвовать jQuery, Backbone, Knockout, Skeleton, Jekyll и другие инструменты, о которых говорилось в этой книге. А поскольку конечные файлы всегда оказываются в папке public, нам не придется ничего менять в файле Rakefile.

Следующим шагом на пути оптимизации может стать Capistrano[1] — инструмент, основанный на Ruby, который позволяет создавать инструкции по внедрению сайтов из систем управления версиями (таких как Git). Несмотря на то что Capistrano создавался для внедрения приложений Ruby on Rails, он прекрасно подходит и для внедрения статических сайтов, PHP-приложений и даже пакетов программ.

Смотрите также

- Рецепт 28. Модульные таблицы стилей с помощью Sass
- Рецепт 29. Как улучшить JavaScript-код с помощью CoffeeScript
- Рецепт 27. Простой блог с помощью Jekyll
- Рецепт 30. Управление файлами с помощью Git

[1] https://github.com/capistrano/capistrano/wiki/

Приложение

Установка Ruby

В нескольких рецептах этой книги используется язык программирования Ruby или его интерпретатор. Это мощный кроссплатформенный интерпретируемый язык, который стал известным во многом благодаря фреймворку Ruby on Rails. Язык Ruby пользуется большой популярностью среди веб-разработчиков; на его основе был создан ряд эффективных инструментов, позволяющих ускорить процесс создания приложений, например Cucumber[1] и SassOS X[2]. Чтобы их использовать, необходимо установить интерпретатор Ruby и систему управления пакетами RubyGems. В этом приложении мы расскажем о том, как это сделать в системах Windows, OS X и Ubuntu.

Windows

В системе Windows установка происходит в два этапа. Сначала необходимо загрузить Ruby Installer для Windows[3]. Выберите версию для Ruby 1.9.2. Когда в процессе установки система предложит вам добавить исполняемые файлы к переменной PATH, разрешите ей это сделать. Тогда вы сможете использовать Ruby и установленные Ruby-библиотеки из командной строки независимо от того, в какой папке вы находитесь.

После этого загрузите с той же страницы комплект средств разработки. Хотя в этих рецептах мы не собираемся писать программы на Ruby, нам нужно будет скомпилировать некоторые компоненты, написанные на языке C. Комплект средств разработки включает все необходимые компиляторы.

Комплект средств разработки представляет собой самораспаковывающийся архив, который нужно извлечь в папку `c:\ruby\devkit`. После этого откройте командную строку и выполните следующие команды:

```
c:\> cd \devkit
c:\> ruby dk.rb init
c:\> ruby dk.rb install
```

Чтобы проверить, как это работает, попробуйте установить джем Cucumber, набрав в командной строке следующее:

```
$ gem install cucumber
```

Вот и всё. Теперь вы можете создавать проекты, используя библиотеки Sass и Cucumber, а также многое другое.

Mac OS X и Linux: установка через RVM

В OS X и многих дистрибутивах Linux доступны готовые интерпретаторы Ruby; в OS X такой интерпретатор установлен по умолчанию. Для установки Ruby и работы

[1] http://cukes.info/
[2] http://sass-lang.com/
[3] http://rubyinstaller.org/downloads/

с разными его версиями мы будем использовать RVM (Ruby Version Manager, программа для управления версиями Ruby)[1]. На всех поддерживаемых платформах RVM работает одинаково, но устанавливается по-разному. Ниже мы расскажем об особенностях установки RVM на OS X и Ubuntu.

Установка RVM на OS X

Чтобы использовать RVM в системе OS X, необходимо установить Xcode[2]. Сам Xcode нам не понадобится, однако это самый простой способ получить необходимые компиляторы C. Эту программу можно найти на установочном DVD вашей OS X или на App Store для Mac. Кроме того, вам нужно будет установить Git для OS X (см. рецепт 30 («Управление файлами с помощью Git»)), с помощью которого мы получим нашу RVM[3].

Теперь необходимо вызвать из терминала команду, которая установит RVM.

```
$ bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

Эта команда находит RVM и устанавливает ее в вашем начальном каталоге.

Далее нужно запустить команду, которая сделает RVM и связанные с ней файлы доступными при каждом новом запуске терминала.

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && \
source "$HOME/.rvm/scripts/rvm"' >> ~/.bashrc
```

Закройте и перезапустите сессию терминала, чтобы убедиться, что RVM работает, и переходите к разделу «Установка Ruby через RVM».

Установка RVM на Ubuntu

В системе Ubuntu RVM требует установки нескольких зависимостей. Для этого удобно использовать следующую команду — она устанавливает компиляторы, необходимые модули и Git (см. рецепт 30 («Управление файлами с помощью Git»)):

```
$ sudo apt-get install build-essential bison openssl \
libreadline6 libreadline6-dev curl git-core zlib1g \
zlib1g-dev libssl-dev libyaml-dev libsqlite3-0 \
libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf
```

После завершения установки этих библиотек вызовите из терминала команду, которая установит RVM:

```
$ bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

Эта команда достает RVM и устанавливает ее в вашем начальном каталоге.

Далее нужно запустить команду, которая сделает RVM и связанные с ней файлы доступными при каждом новом запуске терминала:

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && \
source "$HOME/.rvm/scripts/rvm"' >> ~/.bashrc
```

Закройте и перезапустите сессию терминала, чтобы убедиться, что RVM работает.

---

[1] http://rvm.beginrescueend.com

[2] https://developer.apple.com/xcode/

3 http://code.google.com/p/git-osx-installer/

Установка Ruby через RVM

Теперь, когда у нас есть RVM, мы можем установить Ruby 1.9.2, вызвав такую команду:

```
$ rvm install 1.9.2
```

Далее, чтобы использовать эту версию Ruby, наберите следующее:

```
$ rvm use 1.9.2
```

Для рецептов этой книги вам, возможно, захочется сделать Ruby 1.9.2 версией по умолчанию:

```
$ rvm --default use 1.9.2
```

Чтобы проверить, как это работает, давайте установим библиотеку Cucumber, которую мы использовали в рецепте 34 («Тестирование с помощью Selenium и Cucumber»). Для этого вызовем из терминала следующую команду.

```
$ gem install cucumber
```

Ну вот и все! Нам удалось установить Ruby и все необходимые модули, и теперь мы можем разрабатывать наши проекты, используя Cucumber, Sass, Guard, Jekyll и другие инструменты.

Библиография

[Bur11] Trevor Burnham. CoffeeScript: Accelerated JavaScript Development. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

[CADH09] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesшy, Bryan Helmkamp, and Dan North. The RSpec Book. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[CC11] Hampton Catlin and Michael Lintorn Catlin. Pragmatic Guide to Sass. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

[Hog10] Brian P. Hogan. HTML5 and CSS3: Develop with Tomorrow's Standards Today. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2010.

Б. Хоган. HTML5 и CSS3. Веб-разработка по стандартам нового поколения. СПб.: Питер. 2012.

[Swi08] Travis Swicegood. Pragmatic Version Control Using Git. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2008.

[WH11] Matt Wynne and Aslak Hellesшy. The Cucumber Book: Behaviour-Driven Development for Testers and Developers. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.