



Addy Osmani

Developing Backbone.js Applications

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Эдди Османи

Разработка Backbone.js приложений



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

2014

Эдди Османи

Разработка Backbone.js приложений

Серия «Бестселлеры O'Reilly»

Перевел с английского А. Кузнецов

Заведующий редакцией

А. Кривцов

Ведущий редактор

Ю. Сергиенко

Художественный редактор

Л. Адуевская

Корректоры

С. Беляева, В. Листова

Верстка

Л. Родионова

ББК 32.988.02-018 УДК 004.737.5

Османи Э.

О-74 Разработка Backbone.js приложений . — СПб.: Питер, 2014. — 352 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-00962-1

Backbone — это javascript-библиотека для тяжелых фронтэнд javascript-приложений, таких, например, как gmail или twitter. В таких приложениях вся логика интерфейса ложится на браузер, что дает очень значительное преимущество в скорости интерфейса.

Целью этой книги — стать удобным источником информации в помощь тем, кто разрабатывает реальные приложения с использованием Backbone.

Издание охватывает теорию MVC и методы создания приложений с помощью моделей, представлений, коллекций и маршрутов библиотеки Backbone; модульную разработку ПО с помощью Backbone.js и AMD (посредством библиотеки RequireJS), решение таких типовых задач, как использование вложенных представлений, устранение проблем с маршрутизацией средствами Backbone и jQuery Mobile, а также многие другие вопросы.

12+ (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1449328252 англ.

© Authorized Russian translation of the English edition titled
Developing Backbone.js Applications (ISBN 978-1449328252)
© 2013 Adnan Osmani. This translation is published and sold
by permission of O'Reilly Media, Inc., which owns or controls all
rights to publish and sell the same.

ISBN 978-5-496-00962-1

© Перевод на русский язык ООО Издательство «Питер», 2014
© Издание на русском языке, оформление
ООО Издательство «Питер», 2014

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

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

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 22.01.14. Формат 70x100/16. Усл. п. л. 28,380. Тираж 1000. Заказ

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

Содержание

Предисловие.....	11
Целевая аудитория	12
Авторы книги	12
Как читать эту книгу	12
Использование примеров кода	13
Благодарности	13
Глава 1. Введение	15
Что такое MVC?.....	16
Что такое Backbone.js?.....	16
В каких случаях необходим MVC-фреймворк на JavaScript?	18
Чем хороша библиотека Backbone.js?	19
О чем эта книга.....	20
Глава 2. Ключевые понятия	23
Паттерн MVC.....	23
MVC в Smalltalk-80	24
MVC во Всемирной паутине.....	25
MVC на клиентской стороне и односторонние приложения	28
MVC на клиентской стороне: стиль Backbone	29
Особенности реализации.....	32
Модели	33
Представления.....	34
Что нам дает MVC?.....	37
Углубленное изучение MVC.....	37
Выводы.....	38
Дополнительная информация	38
Коротко о главном.....	38
Библиотека Backbone.js.....	38
Примеры использования	39
Глава 3. Основы Backbone	43
Подготовительные действия	43
Модели	44
Инициализация.....	45
Значения по умолчанию	45
Считывание и присваивание значений.....	46
Прослушивание изменений модели.....	48
Валидация	50
Представления.....	51
Создание новых представлений	52
Что такое El?	52

Коллекции.....	58
Добавление и удаление моделей	58
Считывание моделей.....	59
Прослушивание событий.....	60
Перезапись и обновление коллекций.....	62
Вспомогательные функции Underscore.....	64
API цепочек команд.....	68
Сохранение моделей с помощью RESTful API	69
Считывание моделей с сервера.....	69
Сохранение моделей на сервер	69
Удаление моделей с сервера.....	70
Параметры.....	71
События.....	71
on(), off() и trigger().....	72
listenTo() и stopListening().....	75
События и представления	76
Маршрутизаторы.....	77
Backbone.history	79
API синхронизации библиотеки Backbone	82
Переопределение Backbone.sync	83
Зависимости.....	85
Выходы.....	86
Глава 4. Упражнение 1: управление задачами — ваше первое приложение на Backbone.js	87
Статический HTML	88
Заголовок и сценарии	88
HTML-код приложения.....	89
Шаблоны	90
Модель задачи	91
Коллекция задач	92
Представление приложения	93
Представления отдельных задач.....	98
Запуск приложения	100
Приложение в действии.....	101
Завершение и удаление задач	103
Маршрутизация задач	106
Выходы.....	107
Глава 5. Упражнение 2: книжная библиотека — ваше первое RESTful-приложение на Backbone.js	108
Подготовительные действия	108
Создание модели, коллекции, представлений и объекта приложения.....	113
Создание интерфейса.....	115
Добавление моделей.....	116
Удаление моделей.....	117
Создание сервера базы данных.....	117
Установка Node.js, npm и MongoDB.....	118
Установка модулей узлов.....	118
Создание простого веб-сервера	119

Подключение к базе данных	122
Взаимодействие с сервером	129
Выводы	136
Глава 6. Расширения Backbone	137
MarionetteJS (Backbone.Marionette)	137
Вспомогательный код отображения	139
Сокращение вспомогательного кода с помощью класса Marionette.ItemView	141
Управление памятью	141
Управление регионами	144
Приложение для управления задачами на основе Marionette	146
TodoMVC.js	146
Контроллеры	151
CompositeView	151
TodoMVC.TodoList.Views.js	152
Удобнее ли поддерживать приложения на Marionette?	155
Marionette и гибкость	156
Простая версия, автор — Джеррод Оверсон (Jarrod Overson)	156
RequireJS-версия, автор тот же	156
Версия с Marionette-модулями, автор — Дерик Бейли	156
Дополнительная информация	157
Thorax	157
Простейшее приложение	158
Встраивание дочерних представлений	158
Помощники представлений	159
Помощник collection	160
Настраиваемые атрибуты HTML-данных	161
Ресурсы, посвященные Thorax	163
Выводы	164
Глава 7. Типичные проблемы и пути их решения	165
Работа с вложенными представлениями	165
Проблема	165
Решение 1	165
Решение 2	166
Решение 3	167
Решение 4	167
Управление моделями во вложенных представлениях	169
Проблема	169
Решение	169
Отображение родительского представления из дочернего представления	171
Проблема	171
Решение	171
Удаление иерархий представлений	172
Проблема	172
Решение	172
Отображение иерархий представлений	173
Проблема	173
Решение	173

Работа с вложенными моделями и коллекциями	174
Проблема	174
Решение	174
Улучшенная валидация свойств моделей	175
Проблема	175
Решение	175
Backbone.validateAll.....	177
Backbone.Validation.....	179
Классы для валидации отдельных форм.....	179
Предотвращение конфликтов при использовании нескольких версий Backbone	180
Проблема	180
Решение	180
Создание иерархий моделей и представлений.....	180
Проблема	180
Решение	181
Вызов переопределенных методов	182
Backbone-Super.....	184
Агрегаторы событий и посредники	184
Проблема	184
Решение	184
Агрегатор событий	185
Посредник	186
Сходства и различия	187
Когда и какие отношения использовать.....	188
Совместное использование агрегатора событий и посредника.....	190
Язык паттернов: семантика.....	191
Глава 8. Модульная разработка	192
Организация модулей с помощью RequireJS и AMD	193
Проблемы поддержки множества файлов сценариев.....	193
Для чего требуется улучшать управление зависимостями	194
Асинхронное определение модулей (AMD)	194
Создание AMD-модулей с помощью RequireJS	195
Начало работы с RequireJS.....	196
Пример совместного использования Require.js и Backbone	199
Внешнее хранение шаблонов с помощью библиотеки RequireJS и плагина Text	203
Оптимизация Backbone-приложений для рабочей среды с помощью оптимизатора RequireJS	204
Заключение.....	207
Глава 9. Упражнение 3: ваше первое модульное приложение на Backbone и RequireJS	209
Введение	209
Разметка.....	210
Конфигурационные параметры.....	211
Создание модулей из моделей, представлений и коллекций.....	212
Загрузка модулей с использованием маршрутов.....	217
Конфигурация модуля в формате JSON	217
Маршрутизатор загрузчика модулей	218

Обработка pushState с использованием NodeJS.....	219
Пакеты активов – альтернатива управлению зависимостями.....	220
Глава 10. Пагинация запросов и коллекций Backbone.js.....	222
Backbone.Paginator.....	223
Реальные примеры.....	224
Paginator.requestPager	225
Вспомогательные методы	228
Paginator.clientPager	229
Вспомогательные методы	231
Замечания о реализации.....	234
Плагины.....	235
Инициализация.....	236
Стили.....	237
Заключение.....	239
Глава 11. Backbone Boilerplate и Grunt-BBB.....	240
Начало работы	242
Создание нового проекта	242
index.html.....	243
config.js	244
main.js.....	246
app.js	247
Создание модулей Backbone Boilerplate.....	249
router.js	251
Другие полезные инструменты и проекты	252
Yeoman	252
Backbone DevTools	254
Заключение.....	254
Глава 12. Backbone и jQuery Mobile.....	256
Разработка мобильных приложений с помощью jQuery Mobile	256
Принцип прогрессивного улучшения виджетов в jQM.....	257
Система навигации jQuery Mobile	258
Настройка Backbone-приложения для работы с jQuery Mobile	260
Рабочий цикл приложения на основе Backbone и jQueryMobile	263
Переход на страницу с фиксированным представлением, наследование класса BasicView	265
Управление шаблонами мобильных страниц	265
Управление DOM и методом \$.mobile.changePage	268
Использование сложных методов jQM в Backbone	271
Динамическое изменение DOM.....	271
Перехват сообщений jQuery Mobile.....	273
Производительность	274
Эффективная поддержка многоплатформенности	275
Глава 13. Jasmine	281
Разработка через реализацию поведения	281
Тестовые наборы, спецификации и агенты	283
beforeEach() и afterEach()	287

Общая область видимости	289
Подготовка к тестированию	290
Разработка через тестирование с использованием Backbone	291
Модели	291
Коллекции	293
Представления	295
Тестирование представлений	296
Упражнение	304
Дополнительная литература	304
Заключение	304
Глава 14. QUnit	305
Настройка	305
Пример HTML-кода с QUnit-совместимой разметкой	305
Операторы контроля	308
Простой тестовый вариант с использованием функции test	308
Сравнение фактического и ожидаемого результата функции	309
Структурирование операторов контроля	310
Простейшие модули QUnit	310
Использование методов setup() и teardown()	310
Создание и очистка объектов с помощью методов setup() и teardown()	311
Примеры использования операторов контроля	311
Фикстуры	313
Пример с фикстурами	314
Асинхронный код	316
Глава 15. SinonJS	318
Что такое SinonJS?	318
Базовые агенты	319
Наблюдение за существующими функциями	319
Тестовый интерфейс	319
Заглушки и мок-объекты	320
Заглушки	321
Мок-объекты	322
Упражнение	323
Модели	323
Коллекции	324
Представления	325
Приложение	326
Дополнительные источники информации	327
Глава 16. Заключение	328
Приложение А. Темы для дальнейшего изучения	331
Приложение Б. Источники	351

Предисловие

Еще недавно выражение «развитое веб-приложение» звучало как оксюморон. Сегодня такие приложения встречаются повсеместно, и вам необходимо уметь их разрабатывать.

Веб-приложения традиционно возлагали трудоемкую задачу обработки данных на серверы, передававшие в браузеры HTML-код в виде загружаемых страниц. Использование JavaScript на клиентской стороне ограничивалось лишь улучшением пользовательского интерфейса. Теперь взаимоотношения клиента и сервера изменились: клиентские приложения считывают необработанные данные с сервера и отображают их в браузере в нужное время и в нужном месте.

Представьте себе товарную корзину, созданную с помощью AJAX, которая не требует обновления страницы при добавлении в нее товара. Изначально такой механизм реализовывался с помощью библиотеки jQuery; принцип ее действия заключался в том, чтобы посыпать AJAX-запросы и обновлять текст на странице. В таком подходе обнаруживалось наличие неявной модели данных на стороне клиента.

Увеличение объема кода, взаимодействующего с сервером по своему усмотрению, увеличило сложность клиентских приложений. Удачная архитектура клиентской стороны приложения стала играть не второстепенную, а главную роль: невозможно просто связать воедино несколько частей jQuery-кода и успешно наращивать его вместе с функционалом приложения. Результатом такого подхода станет кошмарное переплетение обратных вызовов пользовательского интерфейса с бизнес-логикой, которое будет выкинуто тем несчастным программистом, который унаследует ваш код.

К счастью, появляется все больше JavaScript-библиотек, помогающих улучшить структуру и эксплуатационную технологичность кода, упрощая разработку сложных интерфейсов. Библиотека Backbone.js стала одним из самых популярных решений для таких задач с открытым исходным кодом; в этой книге я подробно расскажу о ней.

Мы начнем с основ, а затем выполним ряд упражнений и научимся создавать легкие в поддержке приложения с четкой структурой. Если вы — разработчик, стремящийся сделать свой код читабельнее, придать ему улучшенную структуру и упростить его расширение, то это руководство поможет вам.

Я считаю важным повышать профессиональный уровень разработчиков, поэтому книга выпущена по лицензии «Атрибуция — некоммерческое использование — на тех же условиях 3.0 непортированная». Это означает, что вы можете приобрести или бесплатно получить копию этой книги, а также принять участие в ее совершенствовании. Внесение дополнений в содержание книги всегда приветствуется, и я надеюсь, что вместе мы сможем предоставить сообществу современный и полезный информационный ресурс.

Я выражают благодарности Джереми Ашкенасу (Jeremy Ashkenas) и компании DocumentCloud за создание библиотеки Backbone.js, а также ряду участников сообщества за помощь, благодаря которой результаты этого проекта превзошли все мои ожидания.

Целевая аудитория

Эта книга ориентирована на начинающих разработчиков и разработчиков среднего уровня, которые хотят научиться улучшать структуру кода клиентской части приложений. Чтобы освоить большую часть материала, требуется знание основ языка JavaScript; но я попытался кратко описать концепции JavaScript там, где это возможно.

Авторы книги

Эта книга не увидела бы свет без других разработчиков и авторов сообщества, которые потратили свое время и усилия на участие в ее создании. Я выражают благодарность Марку Фридману (Marc Friedman), Дерику Бейли (Derick Bailey), Райану Истриджу (Ryan Eastridge), Джеку Франклину (Jack Franklin), Дэвиду Эмэнду (David Amend), Майку Боллу (Mike Ball), Угису Озолсу (Uģis Ozols) и Бьорну Экенгрену (Björn Ekengren), а также всем остальным замечательным участникам этого проекта.

Как читать эту книгу

Я рассчитываю, что ваши знания языка JavaScript не ограничиваются его основами, поэтому опустил ряд тем, таких как объектные константы. Если вам

необходима дополнительная литература о JavaScript, рекомендую следующие публикации:

- «Выразительный JavaScript» (Eloquent JavaScript);
- Дэвид Флэнеген «JavaScript: подробное руководство» (David Flanagan, JavaScript);
- Дэвид Херман «Сила JavaScript», Питер, 2012 (David Herman, Effective JavaScript);
- Дуглас Крокфорд «JavaScript: сильные стороны», Питер, 2012 (Douglas Crockford, JavaScript: The Good Parts);
- Стоян Стефанов «Объектно-ориентированный JavaScript» (Stoyan Stefanov, Object-Oriented JavaScript).

Использование примеров кода

Эта книга поможет в реальной работе. Вы можете использовать код из приведенных примеров в своих программах и документации. Вам не требуется запрашивать у нас разрешение, если вы не воспроизводите существенную часть кода. То есть если вы включаете в свою программу несколько фрагментов кода из этой книги, то разрешение не требуется, но для продажи или распространения компакт-диска с примерами для книги разрешение необходимо получить. Цитирование этой книги и приведение фрагментов кода примеров из нее не требует разрешения, а включение в документацию вашего продукта существенной части примеров из этой книги требует предварительного разрешения.

Указание авторства не обязательно, но приветствуется; оно включает в себя название книги, имя автора, название издательства и ISBN.

Если вы чувствуете, что использование вами примеров кода выходит за рамки добросовестного использования или приведенных ранее разрешений, не стесняйтесь связаться с нами по адресу: permissions@oreilly.com.

Благодарности

Я безмерно благодарен техническим рецензентам, чья великолепная работа помогла улучшить эту книгу. Их знания, энергия и увлеченность помогли создать удачный обучающий материал, и они продолжают быть для меня источником вдохновения. Я адресую свои благодарности Дерику и Марку (еще раз), Джереми Ашкенасу, Сэмюэлю Клэю (Samuel Clay), Мэту Скейлзу (Mat

Scales), Алексу Гролу (Alex Graul), Душану Гледовичу (Dusan Gledovic) и Синдре Сорхуса (Sindre Sorhus).

Я также хочу поблагодарить мою любящую семью за терпение и заботу, проявленную во время работы над книгой, а также моего великолепного редактора Мэри Треселер (Mary Treseler).

От издательства

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

Мы будем рады узнать ваше мнение!

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

1

Введение

Фрэнк Ллойд Райт однажды сказал: «Вы не можете изменить замысел архитектора, но вы можете открыть двери и окна для света, который видите». В этой книге я надеюсь осветить тему улучшения структуры веб-приложений и распахнуть для вас двери в мир программ, которые будет удобно поддерживать и использовать.

Цель любой архитектуры — это построение или создание чего-либо, в нашем случае — создание стабильного кода, который понравится не только нам, но и разработчикам, а именно они будут заниматься его поддержкой, после того как мы «умоем руки». Итак, наша цель — сделать архитектуру одновременно простой и великолепной.

Современные JavaScript-фреймворки и библиотеки позволяют структурировать и организовывать проекты, закладывая уже в самом начале работы удобный в поддержке фундамент. Появление подобных фреймворков — результат экспериментов и мучений разработчиков, барахтающихся в море хаоса обратных вызовов, вроде того, в котором вы находитесь сейчас или, возможно, окажетесь в ближайшем будущем.

Используя только библиотеки *jQuery* для разработки приложения, вы не сумеете изящно структурировать и организовать код, и ваше JavaScript-приложение быстро превратится в причудливое месиво из селекторов и обратных вызовов *jQuery*, отчаянно пытающихся синхронизировать данные между HTML-компонентами пользовательского интерфейса, логикой JavaScript и обращениями к вашему API для работы с данными.

В отсутствие средства, способного ограничить хаос, вы, скорее всего, попытаетесь выстроить приложение из независимых плагинов и библиотек либо напишете его с нуля и будете вынуждены самостоятельно заниматься его поддержкой. Библиотека *Backbone* решает эту проблему, предоставляя возможность аккуратно организовывать код приложения и распределять функции между его отдельными компонентами, которые легко поддерживать.

В этой книге я в компании с другими опытными авторами продемонстрирую, как улучшить структуру веб-приложений с помощью популярной JavaScript-библиотеки *Backbone.js* версии 1.0.

Что такое MVC?

Ряд современных JavaScript-фреймворков позволяет разработчикам легко организовывать свой код с помощью разновидностей паттерна под названием **MVC** (Model-View-Controller, модель-представление-контроллер). MVC делит задачи приложения на три части:

- **Модели** отражают данные приложения и знания, специфичные для его предметной области. Модель можно представить как данные, имеющие реальное воплощение, — например, пользователь, фотография или список дел. Модели могут уведомлять наблюдателей об изменении своего состояния.
- **Представления**, как правило, включают в себя пользовательский интерфейс приложения (например, разметку и шаблоны), хотя это и не обязательно. Они наблюдают за моделями, но не взаимодействуют с ними напрямую.
- **Контроллеры** обрабатывают входные данные (события или действия пользователя) и обновляют модели.

Таким образом, в MVC-приложении данные, вводимые пользователем, обрабатываются контроллерами, которые обновляют модели. Представления наблюдают за моделями и обновляют пользовательский интерфейс, когда происходят изменения.

Тем не менее JavaScript-фреймворки, реализующие паттерн MVC, не всегда строго следуют ему. В одних решениях (в том числе и в Backbone.js) функции контроллера интегрированы в представление, в то время как другие решения содержат дополнительные компоненты.

По этой причине мы называем такие решения *MV*-фреймворками*: в них чаще всего имеются модель и представление, но может не быть отдельного контроллера и не исключено наличие других компонентов.

Что такое Backbone.js?

Backbone.js (рис. 1.1) — это небольшая JavaScript-библиотека, которая структурирует код клиентской стороны приложения. Она упрощает управление задачами и распределение их в приложении, упрощая поддержку вашего кода.

Обычно разработчики используют библиотеки вроде Backbone.js для создания односторонних приложений. Односторонние приложения — это веб-приложения, которые загружаются в браузер и реагируют на изменения данных на клиентской стороне, не требуя при этом полного обновления страницы с сервера.

Библиотека Backbone многофункциональна и популярна: вокруг нее существует активное сообщество разработчиков, а для самой библиотеки имеется множество плагинов и расширений. Backbone используется для создания нестандартных приложений такими компаниями, как Disqus, Walmart, SoundCloud и LinkedIn.



Рис. 1.1. Домашняя страница Backbone.js

Главная цель Backbone — обеспечить удобные методы считывания данных и манипуляции ими, чтобы избавить вас от необходимости заново реализовывать объектную модель JavaScript. Backbone — это, скорее, не фреймворк, а библиотека, — хорошо масштабируемая и эффективно работающая с другими компонентами, от встраиваемых виджетов до полномасштабных приложений.

Поскольку Backbone достаточно компактна, вашим пользователям не потребуется загружать большой объем данных через мобильное или медленное соединение. Весь исходный код библиотеки Backbone можно прочитать и понять всего за несколько часов.

В каких случаях необходим MVC-фреймворк на JavaScript?

Разрабатывая одностраничное JavaScript-приложение со сложным пользовательским интерфейсом или просто программу, сокращающую число HTTP-запросов при обновлении веб-страницы, вы, скорее всего, придете к созданию элементов, многие из которых входят в состав MV*-фреймворка.

В начале разработки вы легко напишете собственный фреймворк, который позволит правильно структурировать код приложения; тем не менее было бы большой ошибкой рассчитывать, что так же легко вы создадите нечто, сопоставимое по возможностям с Backbone.

Структурирование приложения — гораздо более масштабная задача, чем связывание воедино DOM-библиотеки, шаблонов и маршрутизации. Многие MV*-фреймворки содержат не только элементы, которые вы могли бы написать сами, но и решения тех проблем, с которыми вы сталкиваетесь в процессе разработки. Фреймворки экономят ваше время и их не следует недооценивать.

Итак, когда же следует воспользоваться MV*-фреймворком, а когда — нет?

Если вы создаете приложение, в котором отображение представлений и работа с данными происходят в основном в браузере, то MV*-фреймворк окажется для вас полезным. Примерами таких приложений являются Gmail, News-Blur и мобильный клиент LinkedIn.

Подобные приложения обычно загружают единственный элемент, который содержит все сценарии, таблицы стилей и разметку, необходимую для основных пользовательских задач, а затем выполняют многочисленные дополнительные действия в фоновом режиме: например, позволяют легко перейти от чтения электронной почты или документа к его редактированию, не отправляя при этом серверу запрос на обновление страницы.

Если же вы создаете приложение, в котором основная нагрузка по отображению страниц и представлений возлагается на сервер, и пользуетесь JavaScript или jQuery только для обеспечения интерактивности, то MV*-фреймворк может оказаться лишним. Конечно, существуют сложные веб-приложения, в которых одностраничная структура эффективно сочетается с выборочным отображением представлений, но, как правило, вам лучше организовать свое приложение более простым способом.

Зрелость фреймворка определяется не столько тем, как давно он существует, сколько его цельностью, а также, что еще важнее, его последовательным совершенствованием в выбранной роли. Стал ли он эффективнее решать типичные

задачи? Продолжает ли он развиваться, по мере того как разработчики создают все более масштабные и сложные приложения с его помощью?

Чем хороша библиотека Backbone.js?

Библиотека Backbone предоставляет разработчику минимальный набор примитивов для структурирования данных (модели, коллекции) и пользовательских интерфейсов, полезных при создании динамических приложений на JavaScript. Она не накладывает строгих ограничений на разработку и обеспечивает свободу и гибкость в выборе методов создания оптимальных интерфейсов для веб-приложений. Вы можете воспользоваться штатной архитектурой Backbone или расширить ее под свои требования.

Главное в библиотеке Backbone — это не набор виджетов и не альтернативный способ структурирования объектов; библиотека лишь предоставляет приложению инструменты для считывания данных и манипулирования ими. Backbone также не требует, чтобы вы пользовались конкретным шаблонизатором; вы можете использовать микрошаблоны библиотеки *Underscore.js* (или одной из ее зависимостей) и, таким образом, связывать представления с HTML-кодом, созданным с помощью выбранного вами шаблонизатора.

Многочисленные примеры приложений, созданных с помощью библиотеки Backbone, наглядно демонстрируют ее способность к масштабированию. Backbone также успешно работает с другими библиотеками, что позволяет встраивать Backbone-виджеты в приложения на AngularJS, совместно использовать Backbone и TypeScript или же применять отдельные классы Backbone (например, модели) для работы с данными в простых приложениях.

Структурирование приложений с помощью Backbone не ухудшает их производительность. В Backbone не используются циклы, двухстороннее связывание, непрерывный опрос обновлений структур данных, а применяемые механизмы преимущественно просты. С другой стороны, если вы захотите пойти другим путем, то сможете реализовать свой подход поверх библиотеки Backbone — она никоим образом не воспрепятствует этому.

Если вам понадобится функционал, отсутствующий в Backbone, то вы с большой вероятностью найдете подходящий проект в активном сообществе разработчиков плагинов и расширений. Кроме того, Backbone включает хорошо составленную документацию своего исходного кода, с помощью которой любой разработчик легко разберется в том, что происходит «за кулисами».

Библиотека Backbone совершенствуется на протяжении двух с половиной лет и является развитым продуктом, который и в будущем позволит создавать

минималистичные решения для высококлассных веб-приложений. Я регулярно пользуюсь Backbone и надеюсь, что она станет удачным дополнением и к вашему инструментарию.

О чём эта книга

Я желаю, чтобы эта книга стала для вас авторитетным источником информации в области разработки реальных приложений с использованием Backbone. Если вы считаете целесообразным улучшить или расширить какую-либо главу или раздел, то, пожалуйста, напишите об этом на сайте *GitHub*, посвященном данной книге. Это не займет у вас много времени, но поможет другим разработчикам.

Книга охватывает теорию MVC и методы создания приложений с помощью моделей, представлений, коллекций и маршрутов библиотеки Backbone. Я также познакомлю вас с более сложными темами: модульной разработкой ПО с помощью Backbone.js и AMD (посредством библиотеки *RequireJS*), решением таких типовых задач, как использование вложенных представлений, устранение проблем с маршрутизацией средствами Backbone и jQuery Mobile и др.

Ниже приведено краткое содержание глав этой книги.

○ Глава 2. Ключевые понятия

Посвящена истории паттерна проектирования MVC и его реализации в библиотеке Backbone.js и других JavaScript-фреймворках.

○ Глава 3. Основы Backbone

Рассказывает об основных возможностях библиотеки Backbone.js, а также о технологиях и методах, которые нужно знать для ее эффективного использования.

○ Глава 4. Упражнение 1: управление задачами – ваше первое приложение на Backbone.js

Пошаговая разработка простой клиентской части приложения, работающего со списками задач.

○ Глава 5. Упражнение 2: книжная библиотека – ваше первое RESTful-приложение на Backbone.js

Знакомит вас с разработкой приложения-библиотеки, которое сохраняет свою модель на сервере при помощи REST API.

○ *Глава 6. Расширения Backbone*

Описывает фреймворки расширений Backbone.Marionette и Thorax, которые добавляют к Backbone.js функционал, полезный при разработке крупномасштабных приложений.

○ *Глава 7. Типичные проблемы и пути их решения*

Разбирает типовые проблемы, с которыми вы можете столкнуться при использовании библиотеки Backbone.js, и способы их решения.

○ *Глава 8. Модульная разработка*

Посвящена созданию модульной структуры кода с помощью AMD-модулей и библиотеки RequireJS.

○ *Глава 9. Упражнение 3: ваше первое модульное приложение на Backbone и RequireJS*

Переработка приложения, созданного в упражнении 1, с помощью библиотеки RequireJS для придания ему более модульной структуры.

○ *Глава 10. Пагинация запросов и коллекций Backbone.js*

Рассказывает о том, как использовать плагин Backbone.Paginator для постраничного разбиения данных ваших коллекций.

○ *Глава 11. Backbone Boilerplate and Grunt-BBB*

Посвящена мощным инструментам, позволяющим начать работу над Backbone-приложением с помощью «болванки» стандартного кода.

○ *Глава 12. Backbone и jQuery Mobile*

Описывает проблемы, которые возникают при использовании Backbone с jQuery Mobile.

○ *Глава 13. Jasmine*

Поможет выполнить модульное тестирование Backbone-кода с помощью тестового фреймворка Jasmine.

○ *Глава 14. QUnit*

Посвящена модульному тестированию с использованием инструмента QUnit.

○ *Глава 15. SinonJS*

Рассказывает о модульном тестировании Backbone-приложений с использованием библиотеки SinonJS.

○ *Глава 16. Заключение*

Подводит итог знакомству с разработкой на Backbone.js.

○ *Приложение A*

Продолжение обсуждения паттернов проектирования: сравнение паттернов MVC и MVP (Model-View-Presenter, Модель-Представление-Презентатор) и анализ их соотношения с библиотекой Backbone.js. Кроме того, здесь рассматривается написание с нуля библиотеки, аналогичной Backbone, а также ряд других вопросов.

○ *Приложение B*

Содержит ссылки на дополнительные ресурсы, посвященные Backbone.

2

Ключевые понятия

Паттерны проектирования — это проверенные решения типичных задач разработки, помогающие улучшить организацию и структуру приложений. Используя паттерны, мы применяем коллективный опыт квалифицированных разработчиков в решении подобных задач.

Разработчики настольных и серверных приложений уже давно пользуются многочисленными паттернами проектирования, однако в разработке клиентских приложений подобные паттерны стали применяться лишь несколько лет назад.

В этой главе мы изучим развитие паттерна проектирования «Модель-Представление-Контроллер» (Model-View-Controller, MVC) и узнаем, как библиотека Backbone.js использует этот шаблон в разработке клиентского приложения.

Паттерн MVC

MVC представляет собой паттерн проектирования архитектуры, который улучшает структуру приложения путем разделения его задач. Он позволяет изолировать бизнес-данные (модели) от пользовательских интерфейсов (представлений) с помощью третьего компонента (контроллеров), который управляет логикой и вводом пользовательских данных, а также координирует модели и представления. Исходный паттерн был создан Трюгве Ренсгаугом (Trygve Reenskaug) в 1979 году во время его работы над языком Smalltalk-80 и назывался «Модель-Представление-Контроллер-Редактор» (Model-View-Controller-Editor). Паттерн MVC подробно описан в книге «Банды Четырех» (Gang of Four) под названием «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (Design Patterns: Elements of Reusable Object-Oriented Software), вышедшей в 1994 году и способствовавшей популяризации его использования.

MVC в Smalltalk-80

Важно понимать задачи, для решения которых разрабатывался исходный паттерн MVC, поскольку на сегодняшний день круг этих задач существенно изменился. В 1970-е годы графические пользовательские интерфейсы еще не получили широкого распространения. Для отделения объектов, моделировавших предметы реального мира (например, фотографии и людей), от объектов представления, которые выводились на экран пользователя, стал использоваться подход под названием *раздельное отображение*.

Реализация паттерна MVC в языке Smalltalk-80 развila эту концепцию и ее целью стало отделение логики приложения от пользовательского интерфейса. Идея заключалась в том, что разделение приложения на эти части позволит повторно использовать модели в других интерфейсах приложения. В MVC-архитектуре языка Smalltalk-80 есть несколько интересных аспектов, о которых стоит упомянуть:

- Доменный элемент назывался моделью и не имел никакой связи с пользовательским интерфейсом (представлениями и контроллерами).
- За отображение отвечали представление и контроллер, но не в единственном экземпляре: для каждого элемента, отображаемого на экране, требовалась пара «представление–контроллер», и между ними фактически не существовало разделения.
- Роль контроллера в этой паре заключалась в обработке пользовательского ввода, например событий, соответствующих нажатиям клавиш и щелчкам кнопкой мыши, и выполнению каких-либо осозаемых действий над ними.
- Паттерн наблюдателя использовался для обновления представления при изменении модели.

Иногда разработчики удивляются, узнав, что паттерн наблюдателя (который на сегодняшний день обычно реализуется в виде модели публикации/подписки) был частью архитектуры MVC несколько десятилетий назад. В паттерне MVC языка Smalltalk-80 за моделью наблюдают и представление, и контроллер: представление реагирует на каждое изменение модели. Простой пример этого подхода — приложение для фондовой биржи: чтобы оно могло отображать информацию в реальном времени, любое изменение данных в его модели должно приводить к немедленному обновлению представления.

Мартин Фаулер (Martin Fowler) написал отличную работу об истоках MVC; если вас интересует дополнительная информация о развитии MVC в языке Smalltalk-80, я рекомендую прочитать ее.

MVC во Всемирной паутине

Всемирная паутина в значительной степени опирается на протокол HTTP, у которого нет *внутреннего состояния*. Это означает, что не существует постоянно открытого соединения между браузером и сервером: каждый запрос приводит к созданию нового канала связи между ними. Как только сторона, инициировавшая запрос (например, браузер), получает ответ, соединение закрывается. Такой контекст отличается от контекста операционных систем, в котором разрабатывались многие изначальные идеи MVC. Реализация MVC должна соответствовать веб-контексту.

Примером фреймворка для серверных веб-приложений, в котором сделана попытка применения MVC-подхода в контексте Всемирной паутины, является *Ruby on Rails* (рис. 2.1).

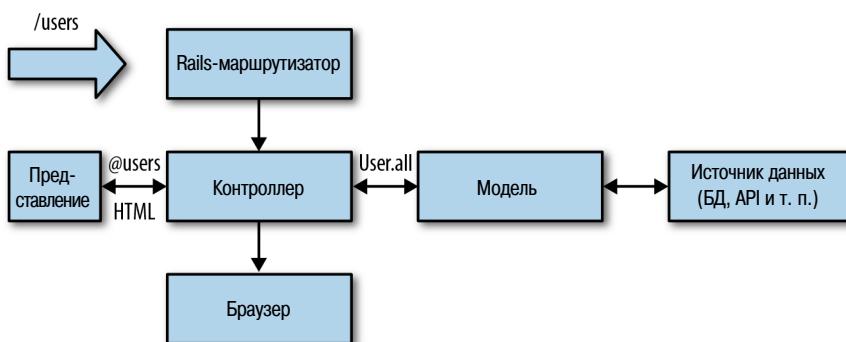


Рис. 2.1. Фреймворк Ruby on Rails

Ядро этого фреймворка составляет архитектура на основе трех знакомых нам MVC-компонентов: модели, представления и контроллера. В Rails они имеют следующий смысл:

- Модели отражают данные приложения и используются для управления правилами взаимодействия со специальной таблицей базы данных. Как правило, одна таблица соответствует одной модели, а модели содержат основную часть бизнес-логики приложения.
- Представления отражают пользовательский интерфейс, часто в виде HTML-кода, который передается в браузер. Они представляют данные приложения стороне, которая запрашивает их.
- Контроллеры являются связующим звеном между моделями и представлениями. Они ответственны за обработку запросов от браузера, обращение

к моделям для считывания данных, а также за передачу этих данных в представления для последующего отображения в браузере.

Несмотря на то что Rails имеет схожее с MVC распределение задач приложения, на самом деле, в Rails используется другой паттерн под названием *Model2*. Это объясняется тем, что в Rails представления не получают уведомлений от моделей, а контроллеры просто передают данные модели напрямую в представление.

Вместе с тем, генерация HTML-страницы в качестве ответа и отделение бизнес-логики от интерфейса дает множество преимуществ даже при приеме URL-запроса серверной стороной приложения. Преимущества, которые предоставляет четкое отделение бизнес-логики от записей базы данных в серверных фреймворках, эквивалентны преимуществам четкого разделения между пользовательским интерфейсом и моделями данных в JavaScript (об этом мы далее поговорим подробнее).

Другие серверные паттерны MVC, например PHP-фреймворк *Zend*, также реализуют паттерн проектирования контроллера запросов. Этот паттерн располагает MVC-стек за единственной точкой входа, благодаря которой все HTTP-запросы (такие, как `http://www.<имя>.com`, `http://www.<имя>.com/<страница>/` и другие) направляются конфигурацией сервера одному и тому же обработчику независимо от URI.

Когда контроллер запросов получает HTTP-запрос, он анализирует его и принимает решение о том, какой класс (контроллер) и метод (действие) вызвать. Выбранное действие контроллера получает управление и взаимодействует с соответствующей моделью для выполнения запроса. Контроллер получает данные от модели, загружает соответствующее представление, вставляет в него данные модели и возвращает ответ браузеру.

Допустим, что у нас есть блог по адресу `www.example.com`, мы хотим отредактировать на нем статью с идентификатором 43 (`id=43`) и посылаем запрос `http://www.example.com/article/edit/43`.

На стороне сервера контроллер запросов проанализирует URL и вызовет контроллер статьи (который соответствует части `/article/` URI) и его редактирующее действие (которое соответствует части `/edit/` URI). Внутри действия будет обращение к модели статей (назовем ее так) и ее методу `Articles::getEntry(43)` (`43` соответствует части `/43` URI). В ответ будут возвращены данные статьи блога для редактирования. Затем контроллер статьи загрузит представление `article/edit`, содержащее логику для вставки данных статьи в форму и позволяющее отредактировать содержание, заголовок статьи и другие (мета)данные. В конце результирующий HTML-ответ будет возвращен в браузер.

Как вы можете догадаться, аналогичная процедура необходима и для обработки запросов POST, когда мы щелкаем по кнопке сохранения в форме. URI для POST-действия будет иметь вид /article/save/43. Запрос будет передан тому же контроллеру, но на этот раз будет вызвано действие сохранения (из-за фрагмента /save/ URI), модель статей сохранит отредактированную статью в базе данных с помощью вызова Articles::saveEntry(43) и браузер будет перенаправлен по URI /article/edit/43 для дальнейшего редактирования статьи.

Наконец, при введении запроса <http://www.example.com/> контроллер запросов вызовет контроллер и действие по умолчанию (например, индексный контроллер и индексное действие). Внутри этого действия будет обращение к модели статей и ее методу Articles::getLastEntries(10), который вернет последние 10 публикаций блога. Контроллер загрузит представление blog/index с базовой логикой отображения публикаций.

На рис. 2.2 показан типичный жизненный цикл запросов/ответов HTTP для серверной части MVC.

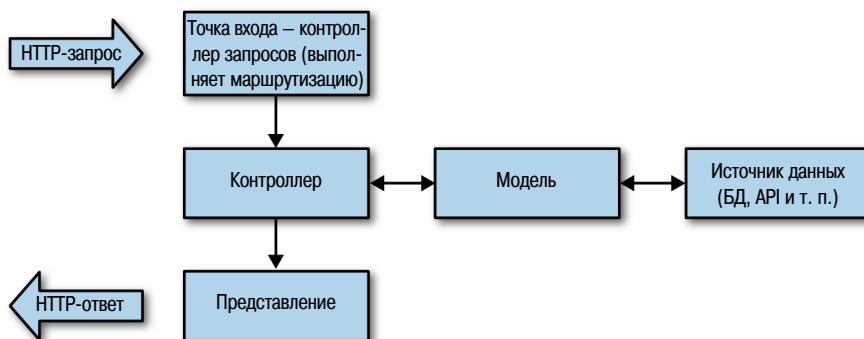


Рис. 2.2. Жизненный цикл запросов/ответов HTTP для серверной части MVC

Сервер получает HTTP-запрос и направляет его через единственную точку входа, на которой контроллер запросов анализирует запрос и вызывает действие соответствующего контроллера. Этот процесс называется *маршрутизацией*. Модель действия получает запрос на возврат и/или сохранение переданных данных. Эта модель взаимодействует с источником данных (например, с базой данных или API). Когда модель заканчивает работу, она возвращает данные контроллеру, который затем загружает соответствующее представление. Это представление выполняет логику отображения (перебирает статьи и печатает их заголовки, содержимое и прочее) с использованием предоставленных данных. В конце HTTP-ответ возвращается браузеру.

MVC на клиентской стороне и одностраничные приложения

Ряд исследований подтвердил, что сокращение задержек при взаимодействии пользователей с сайтами и приложениями положительно влияет на их популярность, а также на степень вовлеченности их пользователей. Это противоречит традиционному подходу к разработке веб-приложений, который сконцентрирован на серверной части и требует полной перезагрузки страницы при переходе с одной страницы на другую. Даже при наличии мощного кэша браузер должен выполнить разбор CSS, JavaScript и HTML, а также отобразить интерфейс на экране.

Такой подход не только приводит к повторной передаче пользователю большого объема данных, но и отрицательно влияет на задержки и общую быстроту взаимодействия интерфейса с пользователем. В последние годы для уменьшения ощущаемых пользователем задержек разработчики зачастую создают одностраничные приложения, которые сначала загружают единственную страницу, а затем обрабатывают навигационные действия и запросы пользователей, не перезагружая страницу целиком.

Когда пользователь переходит к новому представлению, приложение обращается за дополнительным содержимым для этого представления с помощью запроса *XHR* (*XMLHttpRequest*), как правило, к серверному *REST API* или конечной точке. *AJAX* (*Asynchronous JavaScript and XML*, асинхронный JavaScript и XML) обеспечивает асинхронное взаимодействие с сервером, при котором передача и обработка данных происходит в фоновом режиме без вмешательства в работу пользователя с другими частями страницы. Это делает интерфейс более быстрым и удобным.

Одностраничные приложения могут использовать такие возможности браузера, как API журнала для обновления поля адреса при перемещении из одного представления в другое. Эти URL также позволяют пользователям создавать закладки на определенных состояниях приложения и обмениваться ими без необходимости загружать совершенно новые страницы.

Типичное одностраничное приложение содержит небольшие логические элементы с собственными пользовательскими интерфейсами, бизнес-логикой и данными. Хорошим примером является корзина, в которую можно добавлять товары в веб-приложении интернет-магазина. Эту корзину можно отображать в правом верхнем углу страницы (рис. 2.3).

Корзина и ее данные представлены в виде HTML. Данные и связанное с ними представление изменяются с течением времени. Когда-то мы использовали jQuery (или похожую DOM-библиотеку) и множество прямых и обратных AJAX-вызовов для синхронизации данных и представления. В результате

получался плохо структурированный код, который было трудно поддерживать. Ошибки были частыми, а зачастую и неизбежными.

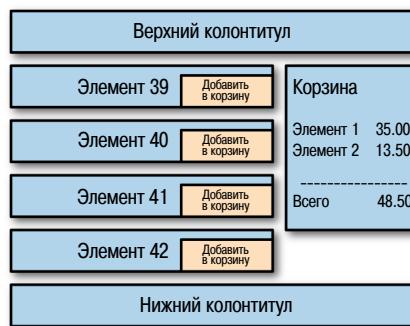


Рис. 2.3. Товарная корзина, формирующая регион одностраничного приложения

Потребность в динамичных и сложных веб-приложениях с быстро реагирующими интерфейсами на основе AJAX потребовала дублирования значительной части описанной логики на стороне клиента, что резко увеличило размер и сложность кода клиентской части приложения. В конечном счете, возникла необходимость реализовать на клиентской стороне MVC или другую схожую архитектуру, чтобы улучшить организацию кода, упростить его поддержку и продлить жизненный цикл приложения.

JavaScript-разработчики воспользовались преимуществами традиционного паттерна MVC и с помощью проб и ошибок постепенно создали несколько MVC-подобных JavaScript-фреймворков, таких как Backbone.js.

MVC на клиентской стороне: стиль Backbone

Давайте рассмотрим, как библиотека Backbone.js обеспечивает преимущества паттерна MVC при разработке клиентской части на примере приложения для управления задачами. Мы будем расширять этот пример в следующих главах по мере изучения возможностей Backbone, однако сейчас сконцентрируем внимание на том, как ключевые компоненты этого приложения соотносятся с MVC.

В данном примере нам потребуется элемент `div`, к которому мы привяжем список задач, а также HTML-шаблон, содержащий место для этого списка, и флажок завершения, который можно устанавливать для отдельных задач. Все перечисленные компоненты создаются при помощи следующего HTML-кода:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
</head>
<body>
    <div id="todo">
    </div>
    <script type="text/template" id="item-template">
        <div>
            <input id="todo_complete" type="checkbox" <%= completed ? 'checked="checked"' : '' %>
            <%- title %>
        </div>
    </script>
    <script src="jquery.js"></script>
    <script src="underscore.js"></script>
    <script src="backbone.js"></script>
    <script src="demo.js"></script>
</body>
</html>
```

В нашем приложении (в файле demo.js) данные каждой задачи хранятся в экземплярах моделей Backbone:

```
// определение модели Todo
var Todo = Backbone.Model.extend({
    // значения атрибутов todo по умолчанию
    defaults: {
        title: '',
        completed: false
    }
});
// Создание экземпляра модели задачи с помощью ее названия,
// атрибут completed имеет значение false по умолчанию
var myTodo = new Todo({
    title: 'Check attributes property of the logged models in the console.'
});
```

Наша модель задачи расширяет модель Backbone.Model и просто определяет значения по умолчанию для двух атрибутов данных. В следующих главах вы увидите, что возможности моделей Backbone гораздо шире, однако этот простой пример иллюстрирует, что модель в первую очередь является контейнером для данных.

Каждая задача будет отображаться на странице с помощью представления TodoView:

```
var TodoView = Backbone.View.extend({
  tagName: 'li',
  // Кэширование функции шаблона для отдельной задачи.
  todoTpl: _.template( $('#item-template').html() ),
  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },
  // Вызывается при первом создании представления
  initialize: function () {
    this.$el = $('#todo');
    // Позже мы рассмотрим вызов
    // this.listenTo(someCollection, 'all', this.render);
    // но вы можете запустить этот пример прямо сейчас,
    // вызвав метод TodoView.render();
  },
  // Повторное отображение заголовков задач.
  render: function() {
    this.$el.html( this.todoTpl( this.model.toJSON() ) );
    // Здесь $el - это ссылка на элемент jQuery,
    // связанный с представлением, todoTpl - ссылка
    // на Underscore-шаблон, а toJSON() возвращает объект,
    // содержащий атрибуты модели.
    // В совокупности этот оператор заменяет HTML-код
    // DOM-элемента на результат создания
    // экземпляра шаблона с атрибутами модели.
    this.input = this$('.edit');
    return this;
  },
  edit: function() {
    // выполняется при двойном щелчке по ярлыку задачи
  },
  close: function() {
    // выполняется, когда задача теряет фокус
  },
  updateOnEnter: function( e ) {
    // выполняется при каждом нажатии клавиши в режиме редактирования задачи,
    // но мы будем ждать нажатия enter, чтобы перейти в действие
  }
});
// создание представления задачи
var todoView = new TodoView({model: myTodo});
```

Мы определяем представление TodoView, расширяя класс Backbone.View, и создаем его экземпляр представления с помощью соответствующей модели. В нашем примере метод render() использует шаблон для формирования HTML-кода задачи внутри элемента li. Каждый вызов функции render() замещает содержимое элемента li данными текущей модели. Таким образом, экземпляр представления отображает содержимое DOM-элемента, используя атрибуты связанной с ним

модели. Позднее мы увидим, как представление может связать свой метод `render()` с событиями изменения модели, которые приводят к повторному отображению представления при изменении модели.

Итак, мы увидели, что с точки зрения паттерна MVC класс `Backbone.Model` реализует модель, а `Backbone.View` — представление. Однако, как мы отмечали ранее, библиотека Backbone отличается от традиционного MVC в том, что касается контроллеров, — класса `Backbone.Controller` не существует!

Вместо него функции контроллера выполняют представление. Вспомните, что контроллеры реагируют на запросы и выполняют соответствующие действия, которые могут привести к изменению модели и обновлению представления. В одностраничном приложении вместо запросов в традиционном смысле этого термина используются события. События могут включать в себя обычные DOM-события (такие, как щелчки) и внутренние события приложения (такие, как изменения модели).

В нашем представлении `TodoView` атрибут `events` играет роль конфигурации контроллера, которая определяет, как события, происходящие внутри DOM-элемента представления, должны передаваться методам обработки событий, определенным в представлении.

Хотя в этом примере события помогают нам связать библиотеку Backbone с паттерном MVC, мы увидим, что они будут играть большую роль в наших одностраничных приложениях. Класс `Backbone.Event` — это фундаментальный компонент Backbone, который используется как в классе `Backbone.Model`, так и в классе `Backbone.View` и предоставляет им богатые возможности управления событиями. Обратите внимание на то, что традиционную роль представления (в стиле языка Smalltalk-80) играет шаблон, а не компонент `Backbone.View`.

На этом наше первое знакомство с фреймворком `Backbone.js` завершается. Ниже мы поговорим о его возможностях, основанных на приведенных здесь простых конструкциях. Перед тем как двигаться дальше, давайте познакомимся с ключевыми возможностями JavaScript-фреймворков на основе паттернов MV*.

Особенности реализации

Одностраничное приложение загружается в браузер с помощью обычного HTTP-запроса и HTTP-ответа. Страница может представлять собой HTML-файл, как в предыдущем примере, или являться представлением, сгенерированным серверной стороной реализации MVC.

Как только одностраничное приложение загружено, маршрутизатор клиентской стороны перехватывает URL и обращается к логике клиента вместо того,

чтобы посыпать новый запрос серверу. На рис. 2.4 показан типичный процесс обработки запроса клиентской стороной MVC в реализации Backbone.

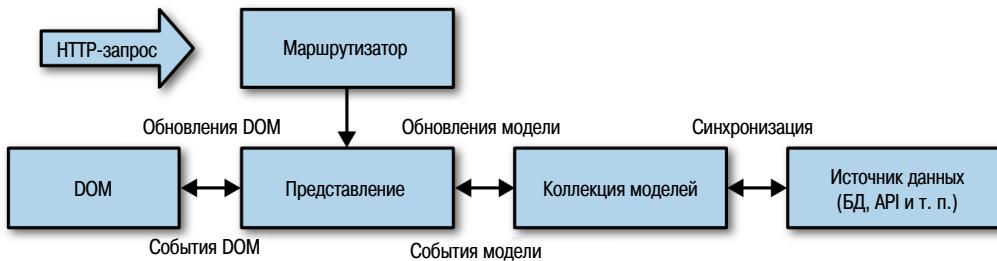


Рис. 2.4. Подход Backbone к обработке запросов

Маршрутизация URL, DOM-события (например, щелчки кнопкой мыши) и события моделей (например, изменения атрибутов) активизируют логику представления, отвечающую за их обработку. Обработчики обновляют DOM и модели, которые могут генерировать дополнительные события. Модели синхронизируются с источниками данных, обращаясь при этом к серверам баз данных.

Модели

- Встроенные возможности моделей изменяются в зависимости от фреймворка; тем не менее модели обычно поддерживают валидацию атрибутов, где атрибуты представляют свойства модели, например ее идентификатор.
- При использовании моделей в реальных приложениях нам, как правило, требуется сохранять их. Это дает возможность редактировать и обновлять модели, зная при этом, что их последние состояния будут сохранены, например, в локальном хранилище веб-браузера или синхронизованы с базой данных.
- За изменениями одной модели могут одновременно наблюдать несколько представлений. Под *наблюдением* я понимаю состояние, в котором представление зарегистрировало свою «заинтересованность» в получении уведомлений об обновлении модели. Это позволяет представлению гарантировать, что информация, отображаемая на экране, соответствует данным, содержащимся в модели. В зависимости от требований вы можете создать как единственное представление, отображающее все атрибуты модели, так и несколько представлений, отображающих различные атрибуты. Здесь важно то, что модель не имеет отношения к тому, как организованы эти

представления; она просто посыпает уведомления о том, что ее данные обновились, через систему событий фреймворка.

- Современные MVC/MV*-фреймворки нередко предоставляют способы группировки моделей. В Backbone такие группы называются *коллекциями*. Управление моделями в группах позволяет создавать логику приложения на основе уведомлений, поступающих от группы, когда модель внутри группы изменяется. Это устраняет необходимость отслеживать состояния отдельных моделей вручную. Ниже мы рассмотрим этот механизм в действии. Коллекции также полезны при выполнении сводных вычислений с участием нескольких моделей.

Представления

- Как правило, пользователи взаимодействуют с представлениями, считывая и редактируя данные модели. Например, в нашем приложении наблюдение за моделью задачи происходит в пользовательском интерфейсе в списке всех задач. Внутри него каждая задача отображается вместе с ее названием и флажком завершения. Редактирование модели осуществляется через представление редактирования, в котором пользователь, выбравший определенный элемент, изменяет его название в форме.
- В нашем представлении мы определяем функцию `render()`, которая отвечает за отображение содержимого модели с помощью механизма шаблонов JavaScript (предоставляемого библиотекой Underscore.js) и обновление содержимого представления, на которое ссылается элемент `this.el`.
- Затем мы добавляем обратный вызов `render()` в качестве подписчика модели, чтобы при изменении модели представление получало сигнал обновиться.
- Возможно, вам не вполне ясно, где здесь происходит взаимодействие приложения с пользователем. Когда пользователь щелкает по задаче (по элементу `todo`), находящейся внутри представления, решение о том, что делать дальше, принимает не представление, а контроллер. Для этого мы добавляем слушателя события к элементу задачи, который делегирует реагирование на щелчок обработчику событий.

Использование шаблонов

Говоря о JavaScript-фреймворках, поддерживающих паттерны MVC/MV*, имеет смысл более подробно познакомиться с шаблонами JavaScript и их отношением к представлениям.

Давно считается плохой (и затратной с точки зрения вычислений) практикой вручную создавать крупные блоки HTML-разметки в памяти путем слияния строк. Разработчики, использующие такую технику, часто перебирают свои данные, упаковывают их во вложенные элементы `div` и вставляют шаблон в объектную модель документа устаревшими методами вроде `document.write`. При таком подходе сценарная разметка часто размещается вместе со стандартной, а это усложняет чтение и поддержку кода, особенно при создании больших приложений.

Шаблонизаторы JavaScript (такие, как *Mustache* или *Handlebars.js*) часто используются для определения шаблонов представлений в виде HTML-разметки, содержащей переменные шаблона. Блоки шаблонов могут храниться вне или внутри тегов `<script>` с нестандартным типом (например, `text/template`). Переменные отделены друг от друга с помощью специального синтаксиса (например, `<%= title %>` в Underscore или `{{title}}` в Handlebars).

Как правило, шаблонизаторы JavaScript принимают данные в различных форматах, в том числе в последовательном формате JSON, который всегда является строкой. Рутинная работа по заполнению шаблонов данными выполняется самим фреймворком. В таком подходе есть несколько преимуществ, особенно если вы храните шаблоны вовне, что позволяет приложениям динамически загружать их по мере необходимости.

Давайте сравним два примера HTML-шаблонов. Один из них реализован с помощью популярной библиотеки Handlebars.js, а в другом используются микрошаблоны библиотеки Underscore.

Handlebars.js

```
<div class="view">
  <input class="toggle" type="checkbox" {{#if completed}} "checked" {{/if}}>
  <label>{{title}}</label>
  <button class="destroy"></button>
</div>
<input class="edit" value="{{title}}>
```

Микрошаблоны Underscore.js

```
<div class="view">
  <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>
  <label><%- title %></label>
  <button class="destroy"></button>
</div>
<input class="edit" value="<%- title %>">
```



Обратите внимание, что в классической веб-разработке навигация между независимыми представлениями требовала обновления страницы. Тем не менее в одностраничных JavaScript-приложениях данные, единожды полученные с сервера через AJAX, могут динамически отображаться в новом представлении на той же странице. Поскольку это не приводит к автоматическому обновлению URL, роль навигации передается маршрутизатору, который помогает управлять состоянием приложения (например, позволяя пользователям создать закладку на представлении, которое они открыли). Поскольку маршрутизаторы не входят в паттерн MVC и не реализованы во всех MVC-подобных фреймворках, я не буду рассматривать их подробнее в этом разделе.

Контроллеры

В нашем приложении для управления задачами контроллер будет отвечать за обработку изменений, которые пользователь вносит в задачи с помощью представления редактирования, и обновление соответствующей модели списка по завершении редактирования.

Большинство JavaScript-фреймворков, реализующих паттерн MVC, отклоняется от его традиционной интерпретации в том, что касается контроллеров. Этому есть ряд причин, но мне кажется, что авторы JavaScript-фреймворков, изучая серверные интерпретации MVC (такие, как Ruby on Rails), приходили к выводу, что их нельзя в точности повторить на клиентской стороне, и по-новому реализовывали контроллеры, чтобы решить проблему управления состоянием приложения. Это разумный подход, однако из-за него разработчикам, впервые использующим MVC, может быть сложно одновременно понять классический паттерн MVC и особую роль контроллеров в других JavaScript-фреймворках.

Так есть ли контроллеры в Backbone.js? На самом деле — нет. В Backbone представления, как правило, содержат логику контроллера, а маршрутизаторы помогают управлять состоянием приложения, однако ни те ни другие не являются контроллерами по канонам MVC.

Следовательно, библиотека Backbone фактически не является MVC-фреймворком, хотя это и противоречит утверждениям в официальной документации и статьям в различных блогах. Правильнее рассматривать Backbone как MV*-фреймворк с особой архитектурой. Разумеется, в этом нет ничего плохого, но чтобы материалы по MVC помогли вам в Backbone-проектах, важно понимать различия между классическим паттерном MVC и MV*-архитектурами.

Что нам дает MVC?

Подведем итог: паттерн MVC помогает разделять логику приложения и пользовательский интерфейс, упрощая модификацию и поддержку и того и другого. Благодаря такому разделению разработчику гораздо легче понять, где вносить изменения в данные, интерфейсы и бизнес-логику приложения и что должны проверять модульные тесты.

Углубленное изучение MVC

Теперь у вас сложилось базовое представление о возможностях паттерна MVC, но для особо любопытных читателей мы немного углубимся в детали.

Авторский коллектив «Банда четырех» (Gang of Four, GoF) рассматривает MVC не как паттерн проектирования, а как набор классов для создания пользовательских интерфейсов. С их точки зрения, MVC представляет собой сочетание трех других классических паттернов проектирования: «Наблюдатель» (публикация/подписка), «Стратегия» и «Компоновщик». В зависимости от особенностей реализации MVC фреймворк также может использовать паттерны «Фабрика» и «Декоратор». Я описал некоторые из этих паттернов в своей книге «Паттерны проектирования на JavaScript для начинающих» (JavaScript Design Patterns for Beginners), которую рекомендую вам для их дальнейшего изучения.

Как мы говорили, модели отражают данные приложения, а представления работают с тем, что пользователь видит на экране. По этой причине ряд ключевых коммуникационных механизмов MVC основан на принципе публикации/подписки (любопытно, что об этом редко говорится в статьях, посвященных паттерну MVC). Когда модель изменяется, она оповещает об этом другие компоненты приложения. После этого подписчик (как правило, контроллер) соответствующим образом обновляет представление. Механизм наблюдения позволяет легко связать несколько представлений с одной и той же моделью.

Для разработчиков, заинтересованных в углубленном изучении распределенной структуры MVC (которая, как мы знаем, зависит от реализации), отметим, что одна из целей этого паттерна — упростить определение отношений «один ко многим» между предметом и его наблюдателями. При каждом изменении предмета его наблюдатели получают обновление. Отношения между представлениями и контроллерами несколько иные: контроллеры упрощают реагирование представлений на ввод различной информации пользователем и являются примером паттерна «Стратегия».

Выводы

Вы изучили классический паттерн MVC и узнали о том, как он позволяет разработчикам распределять функции внутри приложения. Вы также получили представление об отличиях между интерпретациями MVC в различных JavaScript-фреймворках и общих базовых концепциях MVC, на которых эти фреймворки построены.

При изучении нового JavaScript-фреймворка на основе MVC/MV* обратите внимание на то, что может быть полезно сделать шаг назад и разобраться, как в нем реализованы модели, представления, контроллеры и другие концепции. Это поможет лучше понять, как пользоваться этим фреймворком на практике.

Дополнительная информация

Если вы хотите глубже изучить реализацию MVC в библиотеке Backbone.js, то, пожалуйста, обратитесь к разделу «MVP» в приложении А.

Коротко о главном

Библиотека Backbone.js

- Содержит следующие основные компоненты: модели, представления, коллекции и маршрутизаторы. Реализует собственный вариант паттерна MV*.
- Поддерживает коммуникацию на основе событий между представлениями и моделями. Как мы увидим далее, можно легко подключить слушателей событий к любому атрибуту модели, что обеспечивает разработчикам детальный контроль над изменениями в представлениях.
- Поддерживает привязку данных с помощью настраиваемых событий и отдельную библиотеку для наблюдения за парами «ключ-значение» (Key-value observing, KVO).
- Обеспечивает встроенную поддержку RESTful-интерфейсов, благодаря которой модели можно легко связывать с сервером баз данных.
- Обладает развитой системой обработки событий. В Backbone легко добавить поддержку публикации/подписки.
- Создает экземпляры прототипов с помощью ключевого слова `new` — некоторые разработчики предпочитают пользоваться таким методом.

- Может работать с произвольными шаблонизаторами; но по умолчанию доступна библиотека микроБШЛОНОВ Underscore.
- Обеспечивает понятные и гибкие правила структурирования приложений. Backbone не требует использования всех своих компонентов, и разработчик может работать только с теми, которые необходимы его приложению.

Примеры использования

Disqus

Веб-сервис Disqus воспользовался библиотекой Backbone.js в последней версии своего виджета для отправки комментариев к публикациям (рис. 2.5). Такое решение оказалось удачным для распределенного веб-приложения благодаря компактности и простоте расширения Backbone.

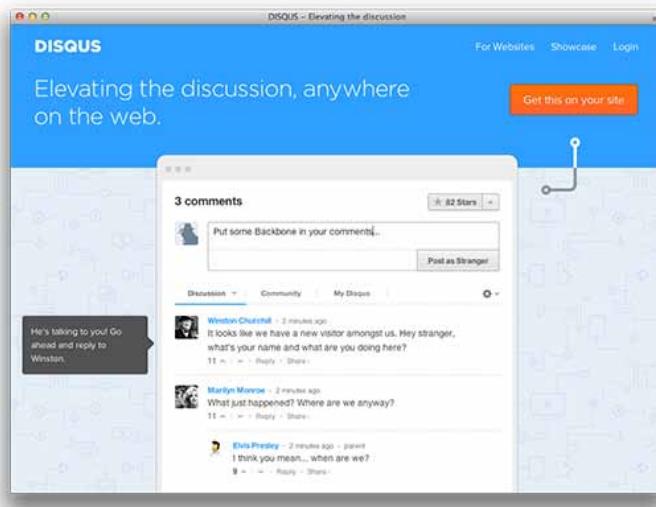


Рис. 2.5. Виджет Disqus для участия в дискуссиях

Khan Academy

Академия Хана предоставляет всем желающим веб-приложение для получения высококлассного бесплатного образования в любой точке мира, а библиотека Backbone позволяет поддерживать код интерфейса этого приложения в модульном и структурированном виде (рис. 2.6).

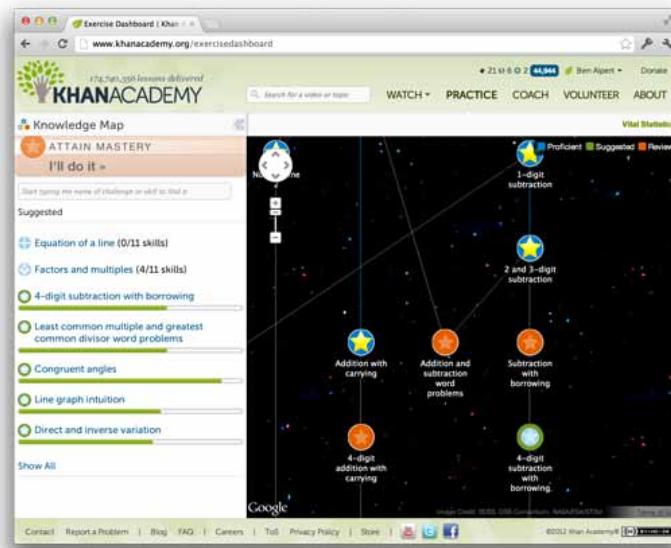


Рис. 2.6. Карта знаний Khan Academy

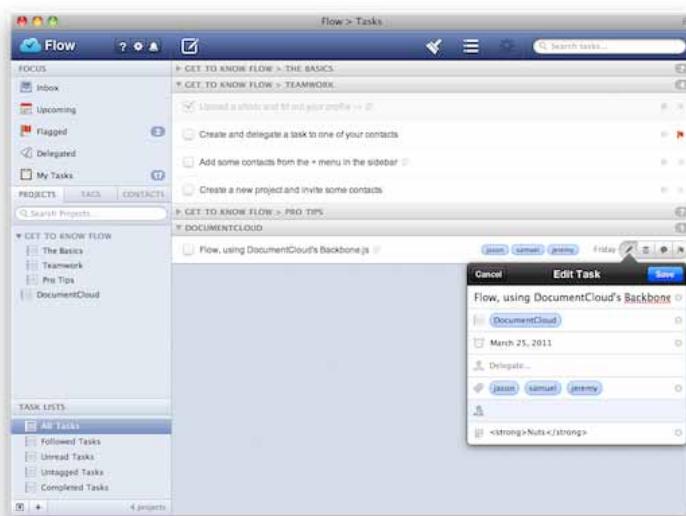


Рис. 2.7. Приложение Flow для онлайнового управления задачами

MetaLab

Компания MetaLab разработала приложение Flow для управления задачами в команде с помощью библиотеки Backbone (рис. 2.7). Рабочая область этого приложения использует Backbone для создания представлений задач, действий, учетных записей, тегов и других компонентов.

Walmart Mobile

Компания Walmart выбрала библиотеку Backbone для разработки своих мобильных веб-приложений (рис. 2.8), а в процессе работы также создала два фреймворка расширения для Backbone — Thorax и Lumbar. Мы рассмотрим оба эти расширения позже в этой книге.

Airbnb

Компания Airbnb (рис. 2.9) разработала с помощью Backbone мобильное приложение, которое теперь используется во многих ее продуктах.



Рис. 2.8. Walmart Mobile



Рис. 2.9. Домашняя страница Airbnb

Code School

Учебное приложение для платформы Code School (рис. 2.10) было разработано с нуля с помощью библиотеки Backbone и задействовало все ее возможности: маршрутизаторы, коллекции, модели и развитую обработку событий.



Рис. 2.10. Учебная среда Code School

3

Основы Backbone

В этой главе мы познакомимся с основами моделей, представлений, коллекций, событий и маршрутизаторов библиотеки Backbone. Материал этой книги не заменяет официальную документацию, однако поможет разобраться во многих ключевых концепциях, лежащих в основе библиотеки Backbone, перед тем как вы начнете разрабатывать приложения с ее помощью.

Подготовительные действия

Перед тем как мы погрузимся в изучение новых примеров кода, давайте создадим небольшую «болванку» с разметкой, в которой будут заданы зависимости, необходимые библиотеке Backbone. Эту «болванку» мы будем многократно использовать с минимальными изменениями (или вообще без них) и без труда запускать код примеров.

Вставьте в ваш текстовый редактор следующий текст и замените закомментированную строку между тегами `<script>` на JavaScript-код любого интересующего примера:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
</script>
<script src="http://documentcloud.github.com/underscore/underscore-min.js">
</script>
<script src="http://documentcloud.github.com/backbone/backbone-min.js">
</script>
<script>
  // Место для вашего кода
```

```
</script>
</body>
</html>
```

Затем сохраните файл и запустите его в каком-либо браузере, например Chrome или Firefox. Если вы предпочитаете работать с онлайновым редактором кода, то воспользуйтесь имеющейся версией этой «болванки» для редактора jsFiddle или jsBin.

Большинство примеров также можно запустить непосредственно из консоли инструментов разработки в браузере, если вы предварительно загрузите HTML-страницу с «болванкой», чтобы обеспечить доступ к библиотеке Backbone и ее зависимостям.

В браузере Chrome инструменты разработчика можно открыть с помощью меню в правом верхнем углу: выберите команду **Tools ▶ Developer Tools** (**Инструменты ▶ Инструменты разработчика**) или воспользуйтесь комбинацией клавиш **Ctrl+Shift+I** в Windows/Linux либо **⌘-Alt-I** в Mac (рис. 3.1).

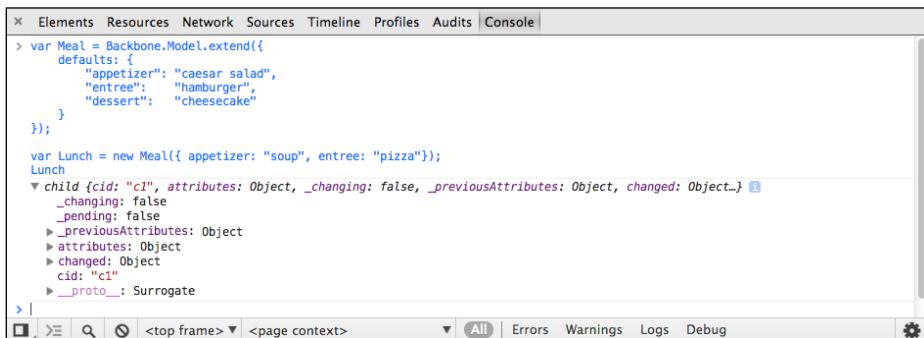


Рис. 3.1. Консоль инструментов разработчика Chrome

Затем перейдите на вкладку **Console**, в которой вы можете ввести и запустить любой фрагмент JavaScript-кода, нажав клавишу **Return**. Вы также можете воспользоваться консолью как многострочным редактором, перемещаясь от конца одной строки к началу другой с помощью клавиш **Shift+Enter** в Windows или **Ctrl+Enter** в Mac.

Модели

Модели Backbone содержат данные приложения, а также логику, работающую с этими данными. Например, мы можем представить в виде модели запланированную к выполнению задачу, включив в модель ее атрибуты — заголовок

(описание того, что нужно сделать) и признак завершения (текущее состояние задачи).

Создавать модели можно, расширяя класс Backbone.Model, как показано ниже:

```
var Todo = Backbone.Model.extend({});  
// Затем мы можем создать собственный экземпляр  
// модели задачи без каких-либо значений:  
var todo1 = new Todo();  
// В журнал: {}  
console.log(JSON.stringify(todo1));  
// или с произвольными данными:  
var todo2 = new Todo({  
    title: 'Check the attributes of both model instances in the console.',  
    completed: true  
});  
// В журнал: {"title":"Check the attributes of both model  
// instances in the console.", "completed":true}  
console.log(JSON.stringify(todo2));
```

Инициализация

Метод initialize() вызывается при создании нового экземпляра модели. Он не является обязательным, но в данном случае его использование — пример хорошего стиля.

```
var Todo = Backbone.Model.extend({  
    initialize: function(){  
        console.log('This model has been initialized.');  
    }  
});  
var myTodo = new Todo();  
// В журнал: This model has been initialized.
```

Значения по умолчанию

Иногда вам нужно, чтобы модель имела набор значений по умолчанию (например, если не все данные вводятся пользователем). Вы можете задать значения по умолчанию для модели с помощью ее свойства defaults.

```
var Todo = Backbone.Model.extend({  
    // Значения атрибутов задачи по умолчанию  
    defaults: {  
        title: '',  
        completed: false  
    }  
});
```

продолжение ↗

```
// Затем мы можем создать собственный экземпляр
// модели задачи без каких-либо значений:
var todo1 = new Todo();
// В журнал: {"title":"","completed":false}
console.log(JSON.stringify(todo1));
// или создать ее экземпляр с какими-либо
// атрибутами (например, title):
var todo2 = new Todo({
    title: 'Check attributes of the logged models in the console.'
});
// В журнал: {}
// in the console.", "completed":false}
console.log(JSON.stringify(todo2));
// или переопределить все атрибуты, заданные по умолчанию:
var todo3 = new Todo({
    title: 'This todo is done, so take no action on this one.',
    completed: true
});
// В журнал: {"title":"This todo is done, so take no action on
// this one.", "completed":true}
console.log(JSON.stringify(todo3));
```

Считывание и присваивание значений

Model.get()

Метод Model.get() обеспечивает простой доступ к атрибутам модели.

```
var Todo = Backbone.Model.extend({
// Значения атрибутов задачи по умолчанию
    defaults: {
        title: '',
        completed: false
    }
});
var todo1 = new Todo();
console.log(todo1.get('title')); // пустая строка
console.log(todo1.get('completed')); // ложь
var todo2 = new Todo({
    title: "Retrieved with model's get() method.",
    completed: true
});
console.log(todo2.get('title')); // Retrieved with model's get() method.
console.log(todo2.get('completed')); // true
```

Если требуется считывать или клонировать все атрибуты данных модели, то воспользуйтесь ее методом toJSON(). Он возвращает копию атрибутов в виде объекта (несмотря на название метода, это не JSON-строка). (Когда методу JSON.

stringify() передается объект, полученный с помощью метода toJSON(), метод JSON.stringify() преобразует в строку результат, возвращаемый toJSON(), а не исходный объект. Этот механизм использовался в предыдущих примерах, где метод JSON.stringify() вызывался для записи в журнал данных об экземплярах модели.)

```
var Todo = Backbone.Model.extend({  
    // Значения атрибутов задачи по умолчанию  
    defaults: {  
        title: '',  
        completed: false  
    }  
});  
var todo1 = new Todo();  
var todo1Attributes = todo1.toJSON();  
// В журнал: {"title":"","completed":false}  
console.log(todo1Attributes);  
var todo2 = new Todo({  
    title: "Try these examples and check results in console.",  
    completed: true  
});  
// Журналы: {"title":"Try these examples and check results in console.",  
// "completed":true}  
console.log(todo2.toJSON());
```

Model.set()

Метод Model.set() устанавливает значения для набора из одного или нескольких атрибутов модели. Когда какой-либо из этих атрибутов изменяет состояние модели, генерируется событие изменения модели. События изменений генерируются и для каждого атрибута и могут быть привязаны к модели (примеры таких событий — change:name и change:age).

```
var Todo = Backbone.Model.extend({  
    // Значения атрибутов задачи по умолчанию  
    defaults: {  
        title: '',  
        completed: false  
    }  
});  
// Задание значений атрибутов модели при создании ее экземпляра  
var myTodo = new Todo({  
    title: "Set through instantiation."  
});  
console.log('Todo title: ' + myTodo.get('title'));  
// Todo title: Set through instantiation.  
console.log('Completed: ' + myTodo.get('completed'));  
// Completed: false
```

продолжение ➔

```
// Задание значения одного атрибута с помощью Model.set():
myTodo.set("title", "Title attribute set through Model.set().");
console.log('Todo title: ' + myTodo.get('title'));
// Todo title: Title attribute set through Model.set().
console.log('Completed: ' + myTodo.get('completed'));
// Completed: false
// Задание набора атрибутов с помощью Model.set():
myTodo.set({
  title: "Both attributes set through Model.set().",
  completed: true
});
console.log('Todo title: ' + myTodo.get('title'));
// Todo title: Both attributes set through Model.set().
console.log('Completed: ' + myTodo.get('completed'));
// Completed: true
```

Прямой доступ

Модели включают в себя атрибут `.attributes`, в котором содержится внутреннее состояние модели. Обычно он имеет вид JSON-объекта, схожего с данными модели на сервере, однако может принимать и другие формы.

При задании значений модели через атрибут `.attributes` события, связанные с моделью, не генерируются.

Передача параметра `{silent:true}` при изменении значения модели не задерживает, а полностью глушит отдельные события `change:attr`:

```
var Person = new Backbone.Model();
Person.set({name: 'Jeremy'}, {silent: true});
console.log(!Person.hasChanged(0));
// true
console.log(!Person.hasChanged(''));
// true
```

Обратите внимание, что по возможности рекомендуется использовать метод `Model.set()` или прямое инстанцирование, как было показано ранее.

Прослушивание изменений модели

Для получения уведомления об изменении модели Backbone вы можете связать с ней слушателя событий, генерируемых при ее изменении. Добавлять слушателей удобно в функции `initialize()`, как показано ниже:

```
var Todo = Backbone.Model.extend({
  // Значения атрибутов задачи по умолчанию
  defaults: {
```

```

        title: '',
        completed: false
    },
    initialize: function(){
        console.log('This model has been initialized.');
        this.on('change', function(){
            console.log('- Values for this model have changed.');
        });
    }
});
var myTodo = new Todo();
myTodo.set('title', 'The listener is triggered whenever an attribute
// value changes.');
console.log('Title has changed: ' + myTodo.get('title'));
myTodo.set('completed', true);
console.log('Completed has changed: ' + myTodo.get('completed'));
myTodo.set({
    title: 'Changing more than one attribute at the same time only triggers
// the listener once.',
    completed: true
});
// Содержимое журнала:
// This model has been initialized.
// - Values for this model have changed.
// Title has changed: The listener is triggered when an attribute value changes.
// - Values for this model have changed.
// Completed has changed: true
// - Values for this model have changed.

```

Можно также прослушивать изменения отдельных атрибутов модели Backbone. В следующем примере мы помещаем в журнал сообщение при каждом изменении определенного атрибута (заголовка задачи).

```

var Todo = Backbone.Model.extend({
// Значения атрибутов задачи по умолчанию
    defaults: {
        title: '',
        completed: false
    },
    initialize: function(){
        console.log('This model has been initialized.');
        this.on('change:title', function(){
            console.log('Title value for this model has changed.');
        });
    },
    setTitle: function(newTitle){
        this.set({ title: newTitle });
    }
});
var myTodo = new Todo();
// Оба следующих изменения приводят к уведомлению слушателя:
myTodo.set('title', 'Check what\'s logged.');

```

продолжение ↗

```
myTodo.setTitle('Go fishing on Sunday.');
// но за таким изменением никто не наблюдает,
// поэтому никто не получает уведомления:
myTodo.set('completed', true);
console.log('Todo set as completed: ' + myTodo.get('completed'));
// Содержимое журнала:
// This model has been initialized.
// Title value for this model has changed.
// Title value for this model has changed.
// Todo set as completed: true
```

Валидация

Backbone обеспечивает валидацию модели с помощью метода `model.validate()`, который позволяет проверять значения атрибутов перед их установкой. По умолчанию валидация выполняется при сохранении модели методом `save()` и при вызове метода `set()` с аргументом `{validate:true}`.

```
var Person = new Backbone.Model({name: 'Jeremy'});
// валидация названия модели
Person.validate = function(attrs) {
  if (!attrs.name) {
    return 'I need your name';
  }
};
// Изменение имени
Person.set({name: 'Samuel'});
console.log(Person.get('name'));
// 'Samuel'
// Удаление атрибута названия, вызов валидации
Person.unset('name', {validate: true});
// ложь
```

Мы также используем метод `unset()`, который удаляет атрибут из внутреннего набора атрибутов модели.

Функции валидации могут быть как простыми, так и сложными. Если переданные атрибуты корректны, то функция `.validate()` не возвращает ничего. Если же атрибуты некорректны, то следует вернуть ошибку.

При возврате ошибки:

- будет сгенерировано событие ошибки, присваивающее свойству модели `validationError` значение, возвращенное этим методом;
- выполнение функции `.save()` прекратится, и атрибуты модели на сервере не будут изменены.

Вот более полный пример валидации:

```
var Todo = Backbone.Model.extend({
  defaults: {
    completed: false
  },
  validate: function(attrs){
    if(attrs.title === undefined){
      return "Remember to set a title for your todo.";
    }
  },
  initialize: function(){
    console.log('This model has been initialized.');
    this.on("invalid", function(model, error){
      console.log(error);
    });
  }
});
var myTodo = new Todo();
myTodo.set('completed', true, {validate: true});
// В журнал: Remember to set a title for your todo.
console.log('completed: ' + myTodo.get('completed')) // completed: ложь
```



Объект `attributes`, переданный функции `validate`, содержит значения атрибутов, которые будут установлены после завершения текущих вызовов `set()` или `save()`. Этот объект отделен от текущих атрибутов модели и от параметров, переданных в операцию. Поскольку он создан путем частичного копирования, невозможно изменить его Number-, String- или Boolean-атрибуты внутри функции, но можно изменить атрибуты во вложенных объектах.

Вы можете ознакомиться с примером, иллюстрирующим вышесказанное, по адресу <http://jsfiddle.net/2NdDY/7/> (автор @fivetanley).

Представления

Представления в Backbone не содержат HTML-разметки приложения; они включают в себя логику отображения данных модели пользователю. Для этой цели представления используют шаблоны JavaScript (например, микрошаблоны Underscore, Mustache, jQuerytmpl и др.). Метод `render()` представления можно связать с событием `change()` модели, это дает представлению возможность немедленно отображать изменения модели без полного обновления страницы.

Создание новых представлений

Создание нового представления относительно несложно и похоже на создание новой модели. Для создания нового представления просто расширьте класс `Backbone.View`. В предыдущей главе мы создали представление `TodoView`; теперь давайте более подробно рассмотрим, как оно работает.

```
var TodoView = Backbone.View.extend({
  tagName: 'li',
  // Кэширование функции шаблона для отдельного элемента.
  todoTpl: _.template( "An example template" ),
  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit': 'close'
  },
  // Повторное отображение заголовка задачи.
  render: function() {
    this.$el.html( this.todoTpl( this.model.toJSON() ) );
    this.input = this$('.edit');
    return this;
  },
  edit: function() {
    // выполняется при двойном щелчке по задаче
  },
  close: function() {
    // выполняется, когда задача теряет фокус
  },
  updateOnEnter: function( e ) {
    // выполняется при каждом нажатии клавиши в режиме редактирования задачи,
    // но мы будем ждать нажатия enter, чтобы попасть в действие
  }
});
var todoView = new TodoView();
// помещаем в журнал ссылку на DOM-элемент,
// соответствующий экземпляру представления
console.log(todoView.el); // в журнале: <li></li>
```

Что такое el?

Ключевым свойством представления является `el` (значение, которое было занесено в журнал в последней строке примера). Что же такое `el` и как оно определяется?

По существу, `el` — это ссылка на DOM-элемент, которая должна содержаться во всех представлениях. С помощью свойства `el` представления могут формировать содержимое своего элемента, а затем одним приемом вставлять его в DOM, что ускоряет отображение, поскольку браузер выполняет минимально необходимое число пересчетов и перерисовок.

Существует два способа связать DOM-элемент с представлением: создать для представления новый элемент и затем добавить его в DOM либо создать ссылку на уже существующий элемент страницы.

Если вы хотите создать новый элемент для вашего представления, задайте любое сочетание из следующих свойств представления: `tagName`, `id` и `className`. Фреймворк создаст новый элемент, ссылка на который будет доступна в свойстве `el`. Если ничего не задано, то `tagName` по умолчанию имеет значение `div`.

В предыдущем примере свойство `tagName` установлено в значение `li`, что приводит к созданию элемента `li`. В следующем примере создается элемент `ul` с атрибутами `id` и `class`:

```
var TodosView = Backbone.View.extend({
  tagName: 'ul', // обязательное свойство, но устанавливается в 'div',
                  // если не задано
  className: 'container', // необязательное свойство; можно присваивать ему
                        // несколько классов, например 'container homepage'
  id: 'todos',         // необязательное свойство
});
var todosView = new TodosView();
console.log(todosView.el); // в журнал: <ul id="todos" class="container"></ul>
```

Приведенный выше код создает DOM-элемент, но не добавляет его в DOM.

```
<ul id="todos" class="container"></ul>
```

Если элемент уже существует на странице, то вы можете установить `el` в качестве CSS-селектора, соответствующего элементу.

```
el: '#footer'
```

В качестве альтернативы можно присвоить атрибуту `el` существующий элемент при создании представления:

```
var todosView = new TodosView({el: $('#footer')});
```

При объявлении представления вы можете определить атрибуты `options`, `el`, `tagName`, `id` и `className` как функции, если хотите, чтобы их значения были заданы на этапе выполнения приложения.

\$el и \$()

Логике представления часто требуется вызывать jQuery- или Zepto-функции для элемента `el` и элементов, вложенных в него. Библиотека Backbone упрощает эту

задачу с помощью определения свойства `$el` и функции `$()`. Свойство `view.$el` эквивалентно `$(view.el)`, а метод `view.$(selector)` эквивалентен `$(view.el).find(selector)`. В методе `render` нашего примера мы видим, что атрибут `this.$el` используется для создания HTML-кода элемента, а метод `this.$()` — для поиска субэлементов класса `edit`.

setElement

Для применения существующего представления Backbone к другому DOM-элементу воспользуйтесь методом `setElement`. Переопределение свойства `this.el` должно одновременно изменять ссылку на DOM и заново привязывать события к новому элементу (и отключать их от старого).

Метод `SetElement` создаст кэшированную ссылку `$el`, переместив переданные события для представления из старого элемента в новый.

```
// Мы создаем два DOM-элемента, соответствующие кнопкам,
// которые бы легко могли быть контейнерами и т. п.
var button1 = $('</button>');
var button2 = $('</button>');
// Определение нового представления
var View = Backbone.View.extend({
  events: {
    click: function(e) {
      console.log(view.el === e.target);
    }
  }
});
// Создание нового экземпляра представления, его
// применение к элементу button1
var view = new View({el: button1});
// Применение представления к элементу button2 с помощью метода setElement
view.setElement(button2);
button1.trigger('click');
button2.trigger('click'); // возвращает true
```

Свойство `el` соответствует лишь разметке представления, которое будет отображено; чтобы представление действительно отобразилось на странице, добавьте ее как новый элемент или присоедините ее к существующему элементу.

```
// Мы также можем передать в setElement "сырую" разметку
// следующим образом (только для примера):
var view = new Backbone.View;
view.setElement('<p><a><b>test</b></a></p>');
view.$('a b').html(); // выводит "test"
```

Изучаем метод render()

`render()` — это дополнительная функция, определяющая логику отображения шаблона. В наших примерах мы будем использовать микрошаблоны библиотеки Underscore, но при желании вы можете пользоваться и другими шаблонизаторами. В нашем примере будет использоваться следующая HTML-разметка:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
  <div id="todo">
    </div>
    <script type="text/template" id="item-template">
      <div>
        <input id="todo_complete" type="checkbox" <%= completed ? 'checked="checked"' : '' %>
        <%= title %>
      </div>
    </script>
    <script src="underscore-min.js"></script>
    <script src="backbone-min.js"></script>
    <script src="jquery-min.js"></script>
    <script src="example.js"></script>
  </body>
</html>
```

Метод `_.template` библиотеки Underscore компилирует JavaScript-шаблоны в функции, которые могут вызываться при выполнении отображения. В представлении `TodoView` я передаю разметку из шаблона с параметром `id`, значение которого равно `item-template`, в метод `_.template()`, чтобы скомпилировать и сохранить ее в свойстве `todoTpl` при создании представления.

Метод `render()` передает этому шаблону атрибуты модели, связанные с представлением, в формате `toJSON()`. Шаблон возвращает свою разметку после оценки выражений, содержащих заголовок модели и флаг завершения задачи. Затем я задаю эту разметку в качестве HTML-содержимого DOM-элемента `el` с помощью свойства `$el`.

Раз-два — и готово! Этот код длиной всего лишь в несколько строк заполняет шаблон, и в результате мы получаем разметку, заполненную данными.

В Backbone в конце функции `render()` принято возвращать указатель `this`. Это удобно по нескольким причинам:

- представления можно многократно использовать в других родительских представлениях;
- можно создать список элементов, не прибегая к отображению и отрисовке каждого элемента в отдельности, а затем единовременно отобразить весь список после его заполнения.

Давайте попробуем реализовать второй пункт. Для простого представления `ListView` напишем следующий метод `render`, не использующий представления `ItemView` для каждого элемента:

```
var ListView = Backbone.View.extend({
  render: function(){
    this.$el.html(this.model.toJSON());
  }
});
```

Как видите, ничего сложного. Предположим, что мы решили усовершенствовать наш список и сформировать его элементы с помощью представления `ItemView`. Напишем следующее представление `ItemView`:

```
var ItemView = Backbone.View.extend({
  events: {},
  render: function(){
    this.$el.html(this.model.toJSON());
    return this;
  }
});
```

Обратите внимание на конструкцию `return this;` в конце функции `render`. Этот типичный прием позволяет повторно использовать представление в качестве подпредставления. С его помощью мы также можем создать пре-визуализацию представления перед тем, как отображать его. Для этого изменим метод отображения в представлении `ListView` следующим образом:

```
var ListView = Backbone.View.extend({
  render: function(){
    // предполагаем, что наша модель публикует элементы,
    // которые мы собираемся вывести в списке
    var items = this.model.get('items');
    // Перебор всех элементов с помощью
    // итератора _.each библиотеки Underscore
    _.each(items, function(item){
      // Создание нового экземпляра ItemView, передача
      // ему конкретного элемента модели
      var itemView = new ItemView({ model: item });
      // DOM-элемент представления itemView добавляется после
      // его отображения. Конструкция
      // 'return this' удобна, когда представление itemView
```

```
// отображает свою модель.
// Затем мы запрашиваем вывод представления ("el")
this.$el.append( itemView.render().el );
},
},
});
```

Набор событий

Набор событий events позволяет подключать слушателей событий к селекторам, связанным с элементом el, либо напрямую к элементу el при отсутствии селектора. Событие имеет вид пары «ключ-значение» 'Имя_события селектор': 'функция_обратного_вызова'. Backbone поддерживает различные типы DOM-событий, в том числе click, submit, mouseover, dblclick и др.

```
// Пример представления
var TodoView = Backbone.View.extend({
  tagName: 'li',
  // с набором событий events, который содержит
  // DOM-события, специфичные для элемента:
  events: {
    'click .toggle': 'toggleCompleted',
    'dblclick label': 'edit',
    'click .destroy': 'clear',
    'blur .edit': 'close'
  },
},
```

Обратите внимание, что Backbone не просто скрыто использует метод .delegate() библиотеки jQuerу, а еще и расширяет его так, что указатель this всегда ссылается на текущий объект представления внутри функций обратного вызова. Единственный нюанс: любой строковый обратный вызов, примененный к атрибуту events, должен иметь соответствующую функцию с таким же именем в области видимости вашего представления.

Используя декларативные, делегируемые события jQuerу, вы можете не беспокоиться о том, был ли определенный элемент отображен в DOM. Обычно в jQuerу вам нужно заботиться об отсутствии или присутствии в DOM все время при привязке событий.

В представлении TodoView обратный вызов редактирования выполняется, когда пользователь совершает двойной щелчок по ярлыку внутри элемента el, функция updateOnEnter вызывается при каждом нажатии клавиши в элементе класса edit, а функция close — когда элемент класса edit теряет фокус. Каждая из этих функций обратного вызова может использовать указатель this, чтобы ссылаться на объект TodoView.

Обратите внимание, что вы также можете сами привязывать методы с помощью функции `_.bind(this.viewEvent, this)`; фактически это эквивалентно тому, что делает значение в паре «ключ-значение» каждого события. Здесь мы используем вызов `_.bind` для повторного отображения представления при изменении модели:

```
var TodoView = Backbone.View.extend({
    initialize: function() {
        this.model.bind('change', _.bind(this.render, this));
    }
});
```

`_.bind` в каждый момент времени работает только с одним методом, однако поддерживает *карринг*; поскольку он возвращает привязанную функцию, метод `_.bind` можно применить к анонимной функции.

Коллекции

Коллекции представляют собой множества моделей и создаются путем расширения класса `Backbone.Collection`.

При создании коллекции требуется также задать свойство, определяющее тип модели, которую будет содержать ваша коллекция, и другие обязательные свойства экземпляра.

В следующем примере мы создадим коллекцию `TodoCollection`, которая будет включать в себя модели задач:

```
var Todo = Backbone.Model.extend({
    defaults: {
        title: '',
        completed: false
    }
});
var TodosCollection = Backbone.Collection.extend({
    model: Todo
});
var myTodo = new Todo({title:'Read the whole book', id: 2});
// передача массива моделей при создании экземпляра коллекции
var todos = new TodosCollection([myTodo]);
console.log("Collection size: " + todos.length); // Collection size: 1
```

Добавление и удаление моделей

В приведенном примере экземпляр коллекции заполнялся во время его создания с помощью массива моделей. После создания коллекции можно добавлять и удалять модели с помощью методов `add()` и `remove()`:

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});
var TodosCollection = Backbone.Collection.extend({
  model: Todo,
});
var a = new Todo({ title: 'Go to Jamaica.'}),
  b = new Todo({ title: 'Go to China.'}),
  c = new Todo({ title: 'Go to Disneyland.'});
var todos = new TodosCollection([a,b]);
console.log("Collection size: " + todos.length);
// В журнал: Collection size: 2
todos.add(c);
console.log("Collection size: " + todos.length);
// В журнал: Collection size: 3
todos.remove([a,b]);
console.log("Collection size: " + todos.length);
// В журнал: Collection size: 1
todos.remove(c);
console.log("Collection size: " + todos.length);
// В журнал: Collection size: 0
```

Обратите внимание, что методы `add()` и `remove()` принимают в качестве аргументов как отдельные модели, так и их списки. Также имейте в виду, что если метод `add()` применяется к коллекции, то передача параметра `{merge: true}` приводит к тому, что атрибуты одинаковых моделей добавляются к существующим моделям, а не игнорируются.

```
var items = new Backbone.Collection;
items.add([{ id : 1, name: "Dog" , age: 3}, { id : 2, name: "cat" , age: 2}]);
items.add([{ id : 1, name: "Bear" }], {merge: true });
items.add([{ id : 2, name: "lion" }]); // merge: false
console.log(JSON.stringify(items.toJSON()));
// [{"id":1,"name":"Bear","age":3}, {"id":2,"name":"cat","age":2}]
```

Считывание моделей

Существует несколько различных способов считывания модели из коллекции. Самый простой способ — с помощью метода `Collection.get()`, который принимает единственный параметр `id`, как показано ниже:

```
var myTodo = new Todo({title:'Read the whole book', id: 2});
// передача массива моделей при создании экземпляра коллекции
var todos = new TodosCollection([myTodo]);
var todo2 = todos.get(2);
// Модели, как объекты, передаются по ссылке
console.log(todo2 === myTodo); // true
```

В клиент-серверных приложениях коллекции содержат модели, считываемые с сервера. При обмене данными между клиентом и сервером вам необходимо уникальным образом идентифицировать модели. Для этого в Backbone используются свойства `id`, `cid` и `idAttribute`.

Каждая модель в Backbone имеет уникальный идентификатор `id`, который является целым числом или строкой (например, UUID). У моделей также есть клиентский идентификатор `cid`, автоматически генерируемый библиотекой Backbone при создании модели. Для считывания модели из коллекций можно использовать любой из идентификаторов.

Основное отличие между этими двумя идентификаторами в том, что `cid` генерируется библиотекой Backbone: это может оказаться удобным при отсутствии настоящего `id`, например если модель еще не сохранена на сервере или вы не храните ее в базе данных.

Атрибут `idAttribute` идентифицирует модель, возвращаемую сервером (то есть `id` в вашей базе данных). Это указывает библиотеке Backbone, какое поле данных сервера должно использоваться для заполнения свойства `id` (можно считать этот атрибут преобразователем). По умолчанию используется поле `id`, однако это можно изменить. Например, если ваш сервер создает уникальный атрибут модели с именем `userId`, то в определении модели вы установите `idAttribute` в значение `userId`.

Значение атрибута `idAttribute` модели должно устанавливаться сервером в момент ее сохранения. После этого вам не нужно задавать это значение вручную, если только не требуется дальнейший контроль.

Класс `Backbone.Collection` содержит массив моделей, упорядоченных по свойству `id`, если оно имеется у экземпляров модели. При вызове `collection.get(id)` проверяется существование в этом массиве модели с соответствующим `id`.

```
// расширяет предыдущий пример
var todoCid = todos.get(todo2.cid);
// как сказано в предыдущем примере,
// модели передаются по ссылке
console.log(todoCid === myTodo); // true
```

Прослушивание событий

Поскольку коллекции представляют собой группы элементов, мы можем прослушивать события `add` и `remove`, которые происходят при добавлении и удалении моделей из коллекции. Например:

```
var TodosCollection = new Backbone.Collection();
TodosCollection.on("add", function(todo) {
```

```

        console.log("I should " + todo.get("title") + ". Have I done it before? "
            + (todo.get("completed") ? 'Yeah!': 'No.' ));
    });
TodosCollection.add([
    { title: 'go to Jamaica', completed: false },
    { title: 'go to China', completed: false },
    { title: 'go to Disneyland', completed: true }
]);
// В журнал:
// I should go to Jamaica. Have I done it before? No.
// I should go to China. Have I done it before? No.
// I should go to Disneyland. Have I done it before? Yeah!

```

Кроме того, с помощью события `change` мы можем прослушивать изменения любых моделей в коллекции.

```

var TodosCollection = new Backbone.Collection();
// запись сообщения в журнал при изменении модели, входящей в коллекцию
TodosCollection.on("change:title", function(model) {
    console.log("Changed my mind! I should " + model.get('title'));
});
TodosCollection.add([
    { title: 'go to Jamaica.', completed: false, id: 3 },
]);
var myTodo = TodosCollection.get(3);
myTodo.set('title', 'go fishing');
// В журнал: Changed my mind! I should go fishing

```

Можно также использовать таблицы событий вида `obj.on({click: action})` в стиле jQuery. Они понятнее, чем три отдельных вызова `.on`, и лучше сочетаются с набором событий, используемых в представлениях:

```

var Todo = Backbone.Model.extend({
    defaults: {
        title: '',
        completed: false
    }
});
var myTodo = new Todo();
myTodo.set({title: 'Buy some cookies', completed: true});
myTodo.on({
    'change:title' : titleChanged,
    'change:completed' : stateChanged
});
function titleChanged(){
    console.log('The title was changed!');
}
function stateChanged(){
    console.log('The state was changed!');
}
myTodo.set({title: 'Get the groceries'});
// Заголовок изменился!

```

События Backbone также поддерживают метод `once()`, который гарантирует, что обратный вызов при получении уведомления сработает только один раз. Этот метод аналогичен методу `once` библиотек Node и jQuery. Он особенно полезен, когда вам нужно сказать: «В следующий раз, когда что-либо произойдет, сделай это».

```
// Определение объекта с двумя счетчиками
var TodoCounter = { counterA: 0, counterB: 0 };
// добавление в Backbone.Events
_.extend(TodoCounter, Backbone.Events);
// инкрементирование счетчика counterA, генерация события
var incrA = function(){
TodoCounter.counterA += 1;
TodoCounter.trigger('event');
};
// инкрементирование счетчика counterB
var incrB = function(){
TodoCounter.counterB += 1;
};
// использование once вместо явного отключения
// нашего слушателя событий
TodoCounter.once('event', incrA);
TodoCounter.once('event', incrB);
// генерация еще одного события
TodoCounter.trigger('event');
// проверка вывода
console.log(TodoCounter.counterA === 1); // true
console.log(TodoCounter.counterB === 1); // true
```

Инкрементирование счетчиков `counterA` и `counterB` должно было произойти только один раз.

Перезапись и обновление коллекций

Вместо добавления и удаления отдельных моделей коллекции вы можете захотеть обновить всю коллекцию одним действием. Метод `Collection.set()` принимает массив моделей и обновляет коллекцию с помощью операций добавления, удаления и изменения моделей.

```
var TodosCollection = new Backbone.Collection();
TodosCollection.add([
    { id: 1, title: 'go to Jamaica.', completed: false },
    { id: 2, title: 'go to China.', completed: false },
    { id: 3, title: 'go to Disneyland.', completed: true }
]);
// мы можем прослушивать события добавления/изменения/удаления
TodosCollection.on("add", function(model) {
    console.log("Added " + model.get('title'));
});
TodosCollection.on("remove", function(model) {
    console.log("Removed " + model.get('title'));
});
```

```

TodosCollection.on("change:completed", function(model) {
    console.log("Completed " + model.get('title'));
});
TodosCollection.set([
    { id: 1, title: 'go to Jamaica.', completed: true },
    { id: 2, title: 'go to China.', completed: false },
    { id: 4, title: 'go to Disney World.', completed: false }
]);
// В журнале:
// Removed go to Disneyland.
// Completed go to Jamaica.
// Added go to Disney World.

```

Для простой замены всего содержимого коллекции воспользуйтесь методом `Collection.reset()`, как показано ниже:

```

var TodosCollection = new Backbone.Collection();
// мы можем прослушивать события перезаписи
TodosCollection.on("reset", function() {
    console.log("Collection reset.");
});
TodosCollection.add([
    { title: 'go to Jamaica.', completed: false },
    { title: 'go to China.', completed: false },
    { title: 'go to Disneyland.', completed: true }
]);
console.log('Collection size: ' + TodosCollection.length); // Collection size: 3
TodosCollection.reset([
    { title: 'go to Cuba.', completed: false }
]);
// В журнал: 'Collection reset.'
console.log('Collection size: ' + TodosCollection.length); // Collection size: 1

```

Еще один полезный совет: чтобы очистить коллекцию, воспользуйтесь операцией `reset` без аргументов. Это удобно, когда вы динамически загружаете новую страницу результатов и хотите очистить текущую страницу.

```
myCollection.reset();
```

Обратите внимание, что при использовании метода `Collection.reset()` события `add` и `remove` не генерируются. Как показано в предыдущем примере, вместо них генерируется событие `reset`. Этот способ можно использовать для сверхоптимизированного отображения в крайних случаях, когда работать с отдельными событиями слишком накладно.

Также имейте в виду, что при прослушивании события `reset` список предыдущих моделей для удобства доступен в элементе `options.previousModels`.

```

var Todo = new Backbone.Model();
var Todos = new Backbone.Collection([Todo])
.on('reset', function(Todos, options) {

```

продолжение ➔

```

        console.log(options.previousModels);
        console.log([Todo]);
        console.log(options.previousModels[0] === Todo); // true
    });
Todos.reset([]);

```

Метод `update()` доступен для коллекций (а также доступен в качестве параметра для выборки) и обеспечивает прекрасные возможности обновления наборов моделей. Этот метод обновляет коллекцию, используя заданный список моделей. Если модель из списка отсутствует в коллекции, то она добавляется в нее. Если модель уже имеется в коллекции, то атрибуты этой модели добавляются к существующей модели. Модели, присутствующие в коллекции, но отсутствующие в списке, удаляются.

```

var theBeatles = new Collection(['john', 'paul', 'george', 'ringo']);
theBeatles.update(['john', 'paul', 'george', 'pete']);
// Генерирует событие `remove` для 'ringo' и событие `add` для 'pete'.
// Обновляет атрибуты john, paul и george, которые могли измениться с течением времени

```

Вспомогательные функции Underscore

Библиотека Backbone в полной мере использует свою тесную связь с библиотекой Underscore, позволяя непосредственно применять многие ее функции к коллекциям.

forEach: перебор элементов коллекций

```

var Todos = new Backbone.Collection();
Todos.add([
    { title: 'go to Belgium.', completed: false },
    { title: 'go to China.', completed: false },
    { title: 'go to Austria.', completed: true }
]);
// перебор моделей коллекции
Todos.forEach(function(model){
    console.log(model.get('title'));
});
// В журнал:
// go to Belgium.
// go to China.
// go to Austria.

```

sortBy(): сортировка коллекции по заданному атрибуту

```

// сортировка коллекции
var sortedByAlphabet = Todos.sortBy(function (todo) {
    return todo.get("title").toLowerCase();
});

```

```
console.log("- Now sorted: ");
sortedByAlphabet.forEach(function(model){
    console.log(model.get('title'));
});
// В журнале:
// go to Austria.
// go to Belgium.
// go to China.
```

map(): создание новой коллекции из списка значений с помощью функции преобразования

```
var count = 1;
console.log(Todos.map(function(model){
    return count++ + ". " + model.get('title'));
}));
// В журнале:
//1. go to Belgium.
//2. go to China.
//3. go to Austria.
```

min()/max(): считывание элемента с минимальным или максимальным значением атрибута

```
Todos.max(function(model){
    return model.id;
}).id;
Todos.min(function(model){
    return model.id;
}).id;
```

pluck(): извлечение заданного атрибута

```
var captions = Todos.pluck('caption');
// возврат списка заголовков
```

filter(): фильтрация коллекции

Фильтрация коллекции с помощью массива идентификаторов моделей.

```
var Todos = Backbone.Collection.extend({
    model: Todo,
    filterById: function(ids){
        return this.models.filter(
            function(c) {
                return _.contains(ids, c.id);
            }
        );
    }
});
```

indexOf(): возврат элемента с заданным индексом внутри коллекции

```
var People = new Backbone.Collection;
People.comparator = function(a, b) {
    return a.get('name') < b.get('name') ? -1 : 1;
};
var tom = new Backbone.Model({name: 'Tom'});
var rob = new Backbone.Model({name: 'Rob'});
var tim = new Backbone.Model({name: 'Tim'});
People.add(tom);
People.add(rob);
People.add(tim);
console.log(People.indexOf(rob) === 0); // true
console.log(People.indexOf(tim) === 1); // true
console.log(People.indexOf(tom) === 2); // true
```

any(): проверка, проходит ли тест истинности итератора хотя бы одно значение коллекции

```
Todos.any(function(model){
    return model.id === 100;
});
// или
Todos.some(function(model){
    return model.id === 100;
});
```

size(): определение размера коллекции

```
Todos.size();
// эквивалентно
Todos.length;
```

isEmpty(): определение, пуста ли коллекция

```
var isEmpty = Todos.isEmpty();
```

groupBy(): помещение коллекции в группу

```
var Todos = new Backbone.Collection();
Todos.add([
    { title: 'go to Belgium.', completed: false },
    { title: 'go to China.', completed: false },
    { title: 'go to Austria.', completed: true }
]);
// создание групп моделей завершенных и незавершенных задач
var byCompleted = Todos.groupBy('completed');
var completed = new Backbone.Collection(byCompleted[true]);
console.log(completed.pluck('title'));
// В журнал: ["go to Austria."]
```

Кроме того, некоторые операции библиотеки Underscore над объектами доступны как методы моделей.

pick(): извлечение набора атрибутов из модели

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.pick('title'));
// В журнал: {title: "go to Austria"}
```

omit(): извлечение из модели всех атрибутов, за исключением заданных

```
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.omit('title'));
// В журнал: {completed: false}
```

keys() и values(): считывание списков имен и значений атрибутов

```
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.keys());
// В журнал: ["title", "completed"]
console.log(todo.values());
// В журнал: ["go to Austria.", false]
```

pairs(): считывание списка атрибутов в виде пар [ключ, значение]

```
var todo = new Todo({title: 'go to Austria.'});
var pairs = todo.pairs();
console.log(pairs[0]);
// в журнале: ["title", "go to Austria."]
console.log(pairs[1]);
// в журнале: ["completed", false]
```

invert(): создание объекта, в котором значения являются ключами, а атрибуты — значениями

```
var todo = new Todo({title: 'go to Austria.'});
console.log(todo.invert());
// в журнале: {go to Austria.: "title", false: "completed"}
```

Полный список возможностей библиотеки Underscore можно найти в ее официальной документации.

API цепочек команд

Еще одна полезная возможность Backbone — поддержка метода `chain()` библиотеки Underscore. Связывание команд в цепочку распространено в объектно-ориентированном программировании; цепочкой команд называется последовательность вызовов методов одного и того же объекта, которые выполняются в рамках одной операции. Поскольку Backbone обеспечивает доступ к операциям библиотеки Underscore над массивами в виде методов объектов коллекций, они не могут быть напрямую объединены в цепочку, так как возвращают массивы, а не исходную коллекцию.

К счастью, метод `chain()` дает вам возможность создавать цепочки из этих методов коллекций.

Метод `chain()` возвращает объект, который содержит операции Underscore над массивами в виде методов, возвращающих этот объект. Цепочка заканчивается вызовом метода `value()`, возвращающим итоговый массив. Если вы не видели «цепочечный» API ранее, то он выглядит следующим образом:

```
var collection = new Backbone.Collection([
  { name: 'Tim', age: 5 },
  { name: 'Ida', age: 26 },
  { name: 'Rob', age: 55 }
]);
var filteredNames = collection.chain()
// начало цепочки, возвращает "обертку" вокруг моделей коллекции
.filter(function(item) { return item.get('age') > 10; })
// возвращает "обернутый" массив без элемента Tim
.map(function(item) { return item.get('name'); })
// возвращает "обернутый" массив с оставшимися именами
.value(); // завершает цепочку и возвращает итоговый массив
console.log(filteredNames); // в журнале: ['Ida', 'Rob']
```

Некоторые методы, специфичные для библиотеки Backbone, возвращают указатель `this`, что также дает возможность объединять их в цепочки:

```
var collection = new Backbone.Collection();
collection
  .add({ name: 'John', age: 23 })
  .add({ name: 'Harry', age: 33 })
  .add({ name: 'Steve', age: 41 });
var names = collection.pluck('name');
console.log(names); // в журнале: ['John', 'Harry', 'Steve']
```

Сохранение моделей с помощью RESTful API

До текущего момента все данные для наших примеров создавались в браузере. В большинстве одностраничных приложений модели создаются из набора данных, находящихся на сервере. Библиотека Backbone значительно упрощает код, который требуется написать для выполнения RESTful-синхронизации с сервером, с помощью простого API моделей и коллекций.

Считывание моделей с сервера

Метод Collections.fetch() считывает множество моделей с сервера в виде JSON-массива, посыпая запрос HTTP GET с URL, заданным в свойстве url коллекции (который может быть и функцией). После получения данных коллекция обновляется с помощью метода set().

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});
var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  url: '/todos'
});
var todos = new TodosCollection();
todos.fetch(); // посылает запрос HTTP GET по адресу /todos
```

Сохранение моделей на сервер

Несмотря на то что библиотека Backbone позволяет одним действием считать всю коллекцию моделей с сервера, обновления моделей выполняются индивидуально с помощью метода save() модели. Когда метод save() вызывается у модели, которая была считана с сервера, он формирует URL, добавляя id модели к URL коллекции, и посыпает серверу запрос HTTP PUT.

Если модель является новым экземпляром, который был создан в браузере (у него нет id), то запрос HTTP POST посыпается по адресу URL коллекции. Можно создать новую модель, добавить ее в коллекцию и отправить на сервер с помощью единственного вызова Collections.create().

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
  }
});
```

продолжение ↗

```

        completed: false
    }
});
var TodosCollection = Backbone.Collection.extend({
    model: Todo,
    url: '/todos'
});
var todos = new TodosCollection();
todos.fetch();
var todo2 = todos.get(2);
todo2.set('title', 'go fishing');
todo2.save(); // посыпает запрос HTTP PUT по адресу /todos/2
todos.create({title: 'Try out code samples'});
// посыпает запрос HTTP POST по адресу /todos и добавляет модель в коллекцию

```

Как говорилось ранее, метод `validate()` модели вызывается автоматически методом `save()` и генерирует событие «ошибка», если валидация оказывается неуспешной.

Удаление моделей с сервера

Модель можно удалить из коллекции и с сервера, вызвав ее метод `destroy()`. В отличие от метода `Collection.remove()`, который только удаляет модель из коллекции, метод `Model.destroy()` также посыпает запрос HTTP `DELETE` в адрес URL коллекции.

```

var Todo = Backbone.Model.extend({
    defaults: {
        title: '',
        completed: false
    }
});
var TodosCollection = Backbone.Collection.extend({
    model: Todo,
    url: '/todos'
});
var todos = new TodosCollection();
todos.fetch();
var todo2 = todos.get(2);
todo2.destroy(); // посыпает запрос HTTP DELETE по адресу /todos/2
                // и удаляет модель из коллекции

```

Вызов `destroy` возвращает значение `false`, если у модели установлен флаг `isNew`:

```

var Todo = new Backbone.Model();
console.log(Todo.destroy());
// false

```

Параметры

Каждый метод RESTful API принимает набор параметров, однако важнее то, что все методы позволяют указать обратные вызовы, обрабатывающие успешные и неуспешные операции, которые можно использовать для настройки обработки ответов сервера.

При передаче параметра `{patch: true}` методу `Model.save()` (то есть при вызове `model.save(attrs, {patch: true})`) он использует запрос HTTP PATCH, чтобы отправить на сервер только измененные атрибуты (такие, как частичные обновления), а не модель целиком.

```
// Частичное сохранение с помощью PATCH
model.clear().set({id: 1, a: 1, b: 2, c: 3, d: 4});
model.save();
model.save({b: 2, d: 4}, {patch: true});
console.log(this.syncArgs.method);
// 'patch'
```

Аналогично, передача параметра `{reset: true}` методу `Collection.fetch()` приводит к тому, что коллекция обновляется методом `reset()`, а не `set()`.

Полные описания поддерживаемых параметров вы найдете в документации Backbone.js.

События

События представляют собой простой механизм передачи управления. Вместо того чтобы одна функция обращалась к другой по имени, вторая функция регистрируется как обработчик, который вызывается при наступлении определенного события.

Теперь «вызывающая» и «вызываемая» часть вашего приложения поменялись местами. Это ключевой фактор, благодаря которому ваша бизнес-логика не обязана знать, как работает пользовательский интерфейс, и самый мощный функционал системы обработки событий в библиотеке Backbone.

Управление событиями — один из самых быстрых способов повысить производительность разработки с помощью Backbone, поэтому давайте подробнее познакомимся с моделью событий Backbone.

Класс `Backbone.Events` входит в состав ряда других классов Backbone:

- Backbone;
- Backbone.Model;

- Backbone.Collection;
- Backbone.Router;
- Backbone.History;
- Backbone.View.

Обратите внимание, что класс `Backbone.Events` является частью объекта `Backbone`. Поскольку этот объект имеет глобальную область видимости, он может использоваться в качестве простой шины событий:

```
Backbone.on('event', function() {
    console.log('Handled Backbone event');
});
```

on(), off() и trigger()

Класс `Backbone.Events` дает любому объекту возможность осуществлять привязку и генерацию произвольных событий. Мы можем легко включить этот модуль в любой объект, при этом события не обязательно объявлять раньше, чем они привязываются к обработчику обратного вызова. Например:

```
var ourObject = {};
// Mixin
_.extend(ourObject, Backbone.Events);
// Добавление настраиваемого события
ourObject.on('dance', function(msg){
    console.log('We triggered ' + msg);
});
// Генерация настраиваемого события
ourObject.trigger('dance', 'our event');
```

Если вы знакомы с настраиваемыми событиями jQuery или концепцией публикации/подписки, то увидите, что класс `Backbone.Events` обеспечивает очень похожий механизм, где метод `on` аналогичен методу `subscribe`, а метод `trigger` — методу `publish`.

Метод `on` привязывает функцию обратного вызова к объекту, как было показано в предыдущем примере с событием `dance`. Обратный вызов выполняется каждый раз при генерации события.

Официальная документация Backbone.js рекомендует использовать пространства имен событий с двоеточиями, если на вашей странице есть хотя бы несколько событий. Например:

```
var ourObject = {};
// примесь
_.extend(ourObject, Backbone.Events);
```

```
function dancing (msg) { console.log("We started " + msg); }
// добавление настраиваемых событий, названных
// с использованием пространства имен
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);
// Генерация настраиваемого события
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");
// Здесь ничего не генерируется, поскольку нет слушателей
ourObject.trigger("dance", "break dancing. Yeah!");
```

Если вы хотите, чтобы уведомления отправлялись при любом событии, про-исходящем на объекте, воспользуйтесь специальным событием `all` (например, для вывода информации о событиях, происходящих в каком-либо месте приложения). Событие `all` можно использовать следующим образом:

```
var ourObject = {};
// Mixin
_.extend(ourObject, Backbone.Events);
function dancing (msg) { console.log("We started " + msg); }
ourObject.on("all", function(eventName){
    console.log("The name of the event passed was " + eventName);
});
// Теперь каждое событие будет перехвачено слушателем события 'all'
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");
ourObject.trigger("dance", "break dancing. Yeah!");
```

Метод `off` удаляет функции обратного вызова, которые ранее были привязаны к объекту. Возвращаясь к нашему сравнению с механизмом публикации/подписки, считайте этот метод аналогом прекращения подписки на настраиваемые события.

Для удаления события `dance`, которое мы связали с объектом `ourObject`, выполним следующее:

```
var ourObject = {};
// примесь
_.extend(ourObject, Backbone.Events);
function dancing (msg) { console.log("We " + msg); }
// добавление настраиваемых событий, названных с использованием пространства имен
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);
// Генерация настраиваемых событий.
// Каждое событие будет перехвачено и обработано.
ourObject.trigger("dance:tap", "started tap dancing. Yeah!");
ourObject.trigger("dance:break", "started break dancing. Yeah!");
// Удаление события, связанного с объектом
ourObject.off("dance:tap");
// Повторная генерация настраиваемых событий,
// но одно из них попадает в журнал.
ourObject.trigger("dance:tap", "stopped tap dancing.");
// не заносится в журнал, так как не прослушивается
ourObject.trigger("dance:break", "break dancing. Yeah!");
```

Чтобы удалить все функции обратного вызова для события, мы передаем имя события (например, `move`) методу `off()` объекта, с которым связано событие. Для удаления конкретного обратного вызова мы передадим его в качестве второго параметра:

```
var ourObject = {};
// примесь
_.extend(ourObject, Backbone.Events);
function dancing (msg) { console.log("We are dancing. " + msg); }
function jumping (msg) { console.log("We are jumping. " + msg); }
// добавление двух слушателей к одному событию
ourObject.on("move", dancing);
ourObject.on("move", jumping);
// Генерация событий. Вызываются оба слушателя.
ourObject.trigger("move", "Yeah!");
// Удаление заданного слушателя
ourObject.off("move", dancing);
// Повторная генерация событий. Один из слушателей отключился.
ourObject.trigger("move", "Yeah, jump, jump!");
```

Наконец, как мы видели в предыдущих примерах, метод `trigger` запускает обратный вызов для заданного события (или списка событий, разделенных пробелами). Например:

```
var ourObject = {};
// примесь
_.extend(ourObject, Backbone.Events);
function doAction (msg) { console.log("We are " + msg); }
// добавление слушателей событий
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);
// единственное событие
ourObject.trigger("dance", 'just dancing.');
// несколько событий
ourObject.trigger("dance jump skip", 'very tired from so much action.');
```

Метод `trigger` может передавать несколько аргументов в функцию обратного вызова:

```
var ourObject = {};
// примесь
_.extend(ourObject, Backbone.Events);
function doAction (action, duration) {
    console.log("We are " + action + ' for ' + duration );
}
// добавление слушателей событий
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
```

```
ourObject.on("skip", doAction);
// передача нескольких аргументов одному событию
ourObject.trigger("dance", 'dancing', "5 minutes");
// передача нескольких аргументов нескольким событиям
ourObject.trigger("dance jump skip", 'on fire', "15 minutes");
```

listenTo() и stopListening()

Если методы `on()` и `off()` добавляют обратные вызовы непосредственно к объекту, за которым ведется наблюдение, то метод `listenTo()` дает одному объекту указание прослушивать события другого объекта и позволяет слушателю следить за тем, какие события он прослушивает. После метода `listenTo()` можно вызвать у слушателя метод `stopListening()`, чтобы он перестал прослушивать события:

```
var a = _.extend({}, Backbone.Events);
var b = _.extend({}, Backbone.Events);
var c = _.extend({}, Backbone.Events);
// добавление к объекту А слушателей событий В и С
a.listenTo(b, 'anything', function(event){
  console.log("anything happened"); });
a.listenTo(c, 'everything', function(event){
  console.log("everything happened"); });
// генерация события
b.trigger('anything'); // в журнал: anything happened
// прекращение прослушивания
a.stopListening();
// А не получает эти события
b.trigger('anything');
c.trigger('everything');
```

Метод `stopListening()` можно также использовать для выборочного прекращения прослушивания, указывая в нем определенное событие, модель или обработчика обратного вызова.

Если вы используете методы `on` и `off` и одновременно удаляете представления и соответствующие им модели, то это, как правило, не вызывает проблем. Тем не менее проблема появляется, когда вы удаляете представление, которое зарегистрировало себя как получателя уведомлений о событиях модели, но при этом не удаляете саму модель и не вызываете метод `off` для удаления обработчика событий представления. Поскольку модель содержит в себе ссылку на функцию обратного вызова представления, «сборщик мусора» JavaScript не может удалить представление из памяти. Такое представление называется *паразитным* и является распространенной формой утечки памяти, поскольку модели, как правило, существуют в приложении дольше, чем соответствующие

представления. Подробную информацию об этой проблеме и ее решении вы найдете в замечательной статье Дерика Бэйли (Derick Bailey, <http://bit.ly/ZN0Sci>).

Каждый вызов `on` объекта требует вызова `off`, чтобы «сборщик мусора» мог выполнить свою работу. Метод `listenTo()` меняет эту практику, позволяя представлениям подключаться к уведомлениям модели и отключаться от них с помощью единственного вызова `stopListening()`.

Реализация метода `View.remove()` по умолчанию вызывает метод `stopListening()`, чтобы отключить всех слушателей, подключенных с помощью метода `listenTo()`, перед уничтожением представления.

```
var view = new Backbone.View();
var b = _.extend({}, Backbone.Events);
view.listenTo(b, 'all', function(){ console.log(true); });
b.trigger('anything'); // в журнале: true
view.listenTo(b, 'all', function(){ console.log(false); });
view.remove(); // неявный вызов stopListening()
b.trigger('anything');
// в журнал ничего не записывается
```

События и представления

В представлении существует два типа событий, которые можно прослушивать: DOM-события и события, сгенерированные API для работы с событиями. Важно понимать различия в том, как представления подключаются к этим событиям, и контекстах, в которых запускаются их обратные вызовы.

Привязка DOM-событий может осуществляться с помощью свойства `events` представления или с помощью вызова `jQuery.on()`. В функциях обратного вызова, подключенных с помощью свойства `events`, указатель `this` ссылается на объект представления; во всех обратных вызовах, подключенных напрямую с помощью библиотеки jQuery, эта библиотека направляет указатель `this` на обрабатывающий DOM-элемент. jQuery передает всем обратным вызовам DOM-событий объект `event`. Более подробную информацию вы найдете в описании метода `delegateEvents()` в документации Backbone.

Привязка событий, созданных с помощью событийного API, осуществляется так, как описано в этом разделе. При подключении события с помощью метода `on()` наблюдаемого объекта вы можете передать контекстный параметр в качестве третьего аргумента. Если вы подключаете событие с помощью метода `listenTo()`, то внутри обратного вызова указатель `this` ссылается на слушателя. Аргументы, передаваемые обратным вызовам событийного API, зависят от типа события. Более подробную информацию вы найдете в разделе «Catalog of Events» («Каталог событий») документации библиотеки Backbone.

Следующий пример иллюстрирует эти различия:

```
<div id="todo">
  <input type='checkbox' />
</div>
var View = Backbone.View.extend({
  el: '#todo',
  // привязка к DOM-событию с помощью свойства event
  events: {
    'click [type="checkbox)": "clicked",
  },
  initialize: function () {
    // привязка к DOM-событию с помощью jQuery
    this.$el.click(this.jqueryClicked);
    // Привязка к API-событию
    this.on('apiEvent', this.callback);
  },
  // 'this' указывает на представление
  clicked: function(event) {
    console.log("events handler for " + this.el.outerHTML);
    this.trigger('apiEvent', event.type);
  },
  // 'this' указывает на обрабатывающий DOM-элемент
  jqueryClicked: function(event) {
    console.log("jQuery handler for " + this.outerHTML);
  },
  callback: function(eventType) {
    console.log("event type was " + eventType);
  }
});
var view = new View();
```

Маршрутизаторы

В Backbone маршрутизаторы позволяют связывать URL (как локальные, так и реальные) с частями вашего приложения. Любая часть приложения, для которой вы хотите сделать возможным создание закладки, кнопки возврата или перессылки, требует URL.

Вот несколько примеров маршрутов, использующих знак #:

```
http://example.com/#about
http://example.com/#search/seasonal-horns/page2
```

Приложение обычно содержит как минимум один URL-маршрут, связанный с функцией, которая определяет, что происходит, когда пользователь переходит по этому пути. Эта связь определяется следующим образом:

```
' маршрут' : 'функция'
```

Давайте определим наш первый маршрутизатор, расширив класс Backbone.Router. Представьте, что вы создаете крупное приложение для управления задачами (нечто вроде персонального органайзера/планировщика), которому требуется сложный маршрутизатор TodoRouter.

Обратите внимание на комментарии в приведенном ниже коде, поскольку они являются продолжением нашего рассмотрения маршрутизаторов.

```
var TodoRouter = Backbone.Router.extend({
    /* определение таблицы маршрутов и функций для этого маршрутизатора*/
    routes: {
        "about" : "showAbout",
        /* Пример использования: http://example.com/#about */
        "todo/:id" : "getTodo",
        /* Это пример использования переменной ":param", которая позволяет
        сравнивать любые компоненты между двумя слэшами URL */
        /* Пример использования: http://example.com/#todo/5 */
        "search/:query" : "searchTodos",
        /* Мы также можем определить несколько маршрутов,
        связанных с одной функцией отображения, в данном случае searchTodos().
        Обратите внимание, как ниже мы дополнительно передаем
        ссылку на номер страницы, если он указан */
        /* Пример использования: http://example.com/#search/job */
        "search/:query:p:page" : "searchTodos",
        /* Как мы видим, URL могут содержать столько элементов ":param",
        сколько мы захотим */
        /* Пример использования: http://example.com/#search/job/p1 */
        "todos/:id/download/*documentPath" : "downloadDocument",
        /* Это пример использования звездочки (*). Звездочки могут соответствовать
        любому числу URL-компонентов и комбинироваться с конструкциями ":param" */
        /* Пример использования: http://example.com/#todos/5/download/todos.doc */
        /* Если вы хотите использовать звездочки за пределами
        маршрутизации по умолчанию, то, скорее всего,
        будет правильно оставить их в конце URL;
        в противном случае у вас может возникнуть необходимость
        выполнить разбор регулярных выражений над вашим фрагментом */
        "*other" : "defaultRoute"
        /* Это маршрут по умолчанию, который также использует звездочку.
        Рассматривайте путь по умолчанию как шаблон для URL, которым
        не найдено соответствия или в которых пользователь совершил опечатку */
        /* Пример использования: http://example.com/# <anything> */,
        "optional(/:item)": "optionalItem",
        "named/optional/(y:z)": "namedOptionalItem"
        /* Маршрутизатор также поддерживает URL с необязательными фрагментами
        в скобках без необходимости использования регулярных выражений. */
    },
    showAbout: function(){
    },
    getTodo: function(id){
        /*
        Обратите внимание на то, что id, найденный в приведенном выше пути,
        будет передан в эту функцию */
        console.log("You are trying to reach todo " + id);
    },
});
```

```

searchTodos: function(query, page){
  var page_number = page || 1;
  console.log("Page number: " + page_number + " of the results for todos
  containing the word: " + query);
},
downloadDocument: function(id, path){
},
defaultRoute: function(other){
  console.log('Invalid. You attempted to reach:' + other);
}
};

/* Теперь, когда мы настроили маршрутизатор, нам нужно создать его экземпляр */
var myTodoRouter = new TodoRouter();

```

Backbone поддерживает функцию `pushState` стандарта HTML5 методом `window.history.pushState`, что позволяет определять маршруты типа <http://backbonejs.org/just/an/example>. Если браузер пользователя не поддерживает `pushState`, то поддержка такой возможности сохраняется, но с автоматическим снижением производительности. Постарайтесь обеспечить поддержку `pushState` и на серверной стороне, хотя реализовать ее несколько сложнее.



Возможно, вы задаетесь вопросом, существует ли максимальное количество маршрутизаторов, которое рекомендуется использовать в приложении. Эндрю де Андраде (Andrew de Andrade) обращает внимание на то, что Document-Cloud, создатель Backbone, в большинстве своих приложений использует единственный маршрутизатор. В ваших проектах вряд ли потребуется больше одного-двух маршрутизаторов; основную часть функций маршрутизации для вашего приложения можно без проблем реализовать с помощью одного маршрутизатора.

Backbone.history

Далее инициализируем класс `Backbone.history`, поскольку он обрабатывает события `hashchange` в нашем приложении. Он будет автоматически обрабатывать определенные маршруты и запускать функции обратного вызова при переходах по этим маршрутам.

Метод `Backbone.history.start()` говорит библиотеке Backbone, что можно начать наблюдение за всеми событиями `hashchange`, как показано ниже:

```

var TodoRouter = Backbone.Router.extend({
  /* определение таблиц маршрутов и функций для этого маршрутизатора */
  routes: {
    "about" : "showAbout",
    "search/:query" : "searchTodos",
    "search/:query/p:page" : "searchTodos"
  },
  продолжение ↗

```

```

showAbout: function(){},
searchTodos: function(query, page){
  var page_number = page || 1;
  console.log("Page number: " + page_number + " of the results for todos
  containing the word: " + query);
}
});
var myTodoRouter = new TodoRouter();
Backbone.history.start();
// перейдите в консоль и проверьте:
// http://localhost/#search/job/p3, в журнале: Page number: 3 of the results for
// todos containing the word: job
// http://localhost/#search/job, в журнале: Page number: 1 of the results for
// todos containing the word: job
// и т. п.

```



Чтобы запустить предыдущий пример, создайте локальную среду разработки и тестовый проект, который мы рассмотрим в главе 4.

Если вы хотите, чтобы URL обновлялся в соответствии с текущим состоянием приложения, используйте метод маршрутизатора `.navigate()`. По умолчанию он просто обновляет фрагмент URL, не генерируя событие `hashchange`:

```

// Представим, что мы хотим обновить определенный
// фрагмент URL (edit), когда пользователь открывает задачу
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },
  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit');
    // обновляет фрагмент, но не генерирует маршрут
  },
  editTodo: function(id) {
    console.log("Edit todo opened.");
  }
});
var myTodoRouter = new TodoRouter();
Backbone.history.start();
// перейдите по адресу http://localhost/#todo/4
//
// URL обновляется на http://localhost/#todo/4/edit
// но функция editTodo() не вызывается, хотя адрес, по которому
// мы перешли, отображен на ней.
//
// в журнале: View todo requested.

```

Метод Router.navigate() также позволяет одновременно сгенерировать маршрут и обновить фрагмент URL — для этого необходимо передать ему параметр trigger:true.



Так поступать не рекомендуется. Предпочтительнее действовать так, как описано ранее: создать фиксированный URL, когда ваше приложение переходит в определенное состояние.

```
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... другие маршруты
  },
  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit', {trigger: true});
    // обновляет фрагмент URL, а также генерирует маршрут
  },
  editTodo: function(id) {
    console.log("Edit todo opened.");
  }
});
var myTodoRouter = new TodoRouter();
Backbone.history.start();
// Перейдите по адресу http://localhost/#todo/4
//
// URL будет обновлен на http://localhost/#todo/4/edit
// в этот раз вызывается функция editTodo().
//
// В журнале:
// View todo requested.
// Edit todo opened.
```

Событие, связанное с маршрутом, генерируется не только на Backbone.history, но и на маршрутизаторе.

```
Backbone.history.on('route', onRoute);
// генерация события 'route' на экземпляре маршрутизатора"
router.on('route', function(name, args) {
  console.log(name === 'routeEvent');
});
location.replace('http://example.com#route-event/x');
Backbone.history.checkUrl();
```

API синхронизации библиотеки Backbone

Ранее мы уже рассмотрели, как библиотека Backbone поддерживает RESTful-интерфейс с помощью методов моделей `fetch()`, `save()` и `destroy()`, а также методов коллекций `fetch()` и `create()`. Теперь мы поближе познакомимся с Backbone-методом `sync`, на котором основаны перечисленные операции.

Метод `Backbone.sync` — неотъемлемая часть библиотеки `Backbone.js`. Он вызывает jQuery-подобный метод `$.ajax()`, поэтому его HTTP-параметры организованы так же, как в API jQuery. Поскольку некоторые устаревшие серверы могут не поддерживать запросы в формате JSON, а также операции `HTTP PUT` и `DELETE`, мы можем настроить Backbone на эмуляцию этих возможностей с помощью двух конфигурационных переменных, значения которых по умолчанию показаны ниже:

```
Backbone.emulateHTTP = false;
// истина, если сервер не поддерживает HTTP PUT или HTTP DELETE
Backbone.emulateJSON = false;
// истина, если сервер не может обрабатывать запросы application/json
```

Встроенный параметр `Backbone.emulateHTTP` должен быть установлен в `true`, если сервер не поддерживает расширенные HTTP-методы. Параметр `Backbone.emulateJSON` должен быть установлен в `true`, если сервер не распознает MIME-тип для JSON.

```
// создание новой коллекции для библиотеки
var Library = Backbone.Collection.extend({
    url : function() { return '/library'; }
});
// определение атрибутов нашей модели
var attrs = {
    title : "The Tempest",
    author : "Bill Shakespeare",
    length : 123
};
// создание нового экземпляра библиотеки
var library = new Library();
// создание нового экземпляра модели внутри нашей коллекции
library.create(attrs, {wait: false});
// обновление только с emulateHTTP
library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
    emulateHTTP: true
});
// проверка параметров ajaxSettings, используемых в нашем запросе
console.log(this.ajaxSettings.url === '/library/2-the-tempest');
// true
console.log(this.ajaxSettings.type === 'POST'); // true
console.log(this.ajaxSettings.contentType === 'application/json');
```

```
// true
// разбор данных запроса для проверки его корректности
var data = JSON.parse(this.ajaxSettings.data);
console.log(data.id === '2-the-tempest'); // true
console.log(data.author === 'Tim Shakespeare'); // true
console.log(data.length === 123); // true
```

Аналогично, мы могли бы просто выполнить обновление с помощью `emulateJSON`:

```
library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
    emulateJSON: true
});
console.log(this.ajaxSettings.url === '/library/2-the-tempest'); // true
console.log(this.ajaxSettings.type === 'PUT'); // истина
console.log(this.ajaxSettings.contentType ===
'application/x-www-form-urlencoded'); // истина
var data = JSON.parse(this.ajaxSettings.data.model);
console.log(data.id === '2-the-tempest');
console.log(data.author === 'Tim Shakespeare');
console.log(data.length === 123);
```

Метод `Backbone.sync` вызывается каждый раз, когда библиотека Backbone пытается считывать, сохранять или удалять модели. Для выполнения подобных запросов этот метод использует jQuery- или Zepto-реализацию метода `$.ajax()`, однако вы можете переопределить его по собственному усмотрению.

Переопределение `Backbone.sync`

Переопределите метод `sync` глобально как `Backbone.sync` или на более низком уровне, добавив функцию `sync` в коллекцию Backbone или в отдельную модель.

Поскольку сохранение целиком выполняется функцией `Backbone.sync`, используем другой уровень сохранения, просто переопределив `Backbone.sync` с помощью функции, имеющей такой же прототип:

```
Backbone.sync = function(method, model, options) {
};
```

Метод `methodMap` используется в стандартной реализации `sync` для привязки параметра метода к HTTP-операции и определяет тип действия, соответствующий каждому аргументу метода:

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'patch': 'PATCH',
```

продолжение ↗

```
'delete': 'DELETE',
'read': 'GET'
};
```

Для замены стандартной реализации sync функцией, которая просто записывает обращения в журнал, выполним следующее действие:

```
var id_counter = 1;
Backbone.sync = function(method, model) {
    console.log("I've been passed " + method + " with " + JSON.stringify(model));
    if(method === 'create'){ model.set('id', id_counter++); }
};
```

Обратите внимание, что мы присваиваем уникальные id всем создаваемым моделям.

Цель переопределения метода Backbone.sync заключается в поддержке сохранения моделей в других базах данных. Встроенный метод привязан к конкретному RESTful JSON API, поскольку библиотека Backbone была изначально заимствована из приложения на Ruby on Rails, в котором HTTP-методы типа PUT использовались аналогичным образом.

Вызов метода sync содержит три параметра:

- method. Имеет одно из значений: create, update, patch, delete и read.
- model. Объект Backbone-модели.
- options. Может включать в себя методы для обработки успешных и неуспешных операций.

Реализуем новый метод sync с помощью следующего паттерна:

```
Backbone.sync = function(method, model, options) {
    function success(result) {
        // обработка успешных результатов из MyAPI
        if (options.success) {
            options.success(result);
        }
    }
    function error(result) {
        // обработка ошибок из MyAPI
        if (options.error) {
            options.error(result);
        }
    }
    options || (options = {});
    switch (method) {
        case 'create':
            return MyAPI.create(model, success, error);
    }
}
```

```
case 'update':
    return MyAPI.update(model, success, error);
case 'patch':
    return MyAPI.patch(model, success, error);
case 'delete':
    return MyAPI.destroy(model, success, error);
case 'read':
    if (model.attributes[model.idAttribute]) {
        return MyAPI.find(model, success, error);
    } else {
        return MyAPI.findAll(model, success, error);
    }
}
};
```

Этот паттерн делегирует вызовы API новому объекту (`MyAPI`), который может принадлежать классу в стиле Backbone, поддерживающему события. Данный паттерн можно смело протестировать отдельно и использовать его с библиотеками, отличными от Backbone.

Существует еще несколько реализаций метода `sync`. Все примеры, перечисленные ниже, доступны на GitHub:

- *Backbone localStorage*. Сохраняет данные в локальном хранилище браузера.
- *Backbone offline*. Поддерживает работу в оффлайновом режиме.
- *Backbone Redis*. Использует хранилище Redis с параметрами «ключ-значение».
- *backbone-parse*. Интегрирует Backbone с Parse.com.
- *backbone-websql*. Сохраняет данные в WebSQL.
- *Backbone Caching Sync*. Использует локальное хранилище как кэш для реализаций других механизмов синхронизации.

Зависимости

Официальная документация Backbone.js содержит следующую информацию.



Единственной жесткой зависимостью для библиотеки Backbone является библиотека Underscore.js (версии 1.4.3 или старше) или Lo-Dash. Для использования RESTful-функций сохранения данных, поддержки истории с помощью Backbone.Router и DOM-манипуляций с применением класса Backbone.View возьмите библиотеку json2.js и одну из библиотек jQuery (версии 1.7.0 или) или Zepto.

Смысл этого абзаца следующий: если вам потребуется работать с чем-то кроме моделей, то необходимо включить в проект DOM-библиотеку, например jQuery или Zepto. Использование библиотеки Underscore обусловлено в первую очередь ее вспомогательными функциями (на которые Backbone в значительной степени опирается), а библиотека json2.js обеспечивает поддержку стандарта JSON в устаревших браузерах при использовании метода Backbone.sync.

Выводы

В этой главе вы узнали о компонентах, которые будут использоваться при разработке приложений с помощью библиотеки Backbone: моделях, представлениях, коллекциях и маршрутизаторах. Мы изучили класс событий, который используется для интеграции механизма подписки/публикации во все компоненты приложения, и познакомились с его применением к различным объектам. Наконец, мы узнали о богатых возможностях манипуляции и сохранения данных, обеспечиваемых с помощью применения API библиотек Underscore.js и jQuery/Zepto к коллекциям и моделям Backbone.

Библиотека Backbone содержит много функций и возможностей, которые остались за рамками этой главы. Эти функции постоянно совершенствуются, для получения о них более подробной информации или данных об их новейших возможностях изучайте официальную документацию Backbone. В следующей главе вы приступите к практической работе с Backbone в ходе нашего знакомства с реализацией первого Backbone-приложения.

4

Упражнение 1: управление задачами – ваше первое приложение на Backbone.js

Теперь, когда мы изучили необходимые основы, давайте напишем первое приложение на Backbone.js. Мы создадим приложение для управления задачами, размещенное на сайте [TodoMVC.com](#). Разработка списка задач – отличный способ попрактиковаться в Backbone (рис. 4.1). Это относительно простое приложение, однако технические нюансы, касающиеся связывания его компонентов, сохранения данных моделей, маршрутизации и отображения шаблонов, позволяют проиллюстрировать ряд ключевых возможностей библиотеки Backbone.

Рассмотрим общую архитектуру нашего приложения. Нам потребуется:

- модель, описывающая отдельные задачи;
- коллекция `TodoList` для хранения задач;
- способ создания задач;
- способ отображения списков задач;
- способ редактирования существующих задач;
- способ помечать задачу как завершенную;
- способ удаления задач;
- способ фильтрации завершенных и незавершенных задач.

В сущности, перечисленные возможности представляют собой классические CRUD-методы (`create/read/update/delete`, создание/чтение/обновление/удаление). Итак, начнем!

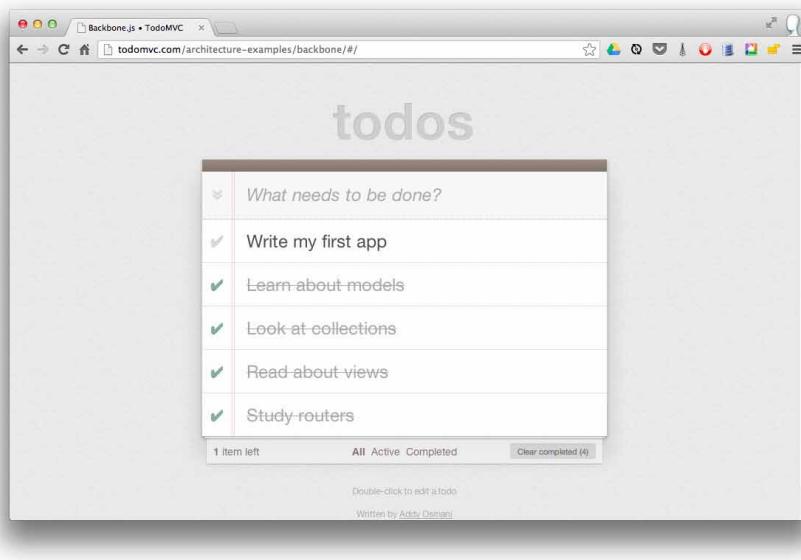


Рис. 4.1. Управление задачами — наше первое приложение на Backbone.js

Статический HTML

Мы поместим весь наш HTML-код в один файл с именем index.html.

Заголовок и сценарии

Сначала мы создадим заголовок и основные зависимости приложения: jQuery, Underscore, Backbone.js и адаптер локального хранилища Backbone.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Backbone.js • TodoMVC</title>
    <link rel="stylesheet" href="assets/base.css">
</head>
<body>
    <script type="text/template" id="item-template"></script>
    <script type="text/template" id="stats-template"></script>
    <script src="js/lib/jquery.min.js"></script>
    <script src="js/lib/underscore-min.js"></script>
```

```
<script src="js/lib/backbone-min.js"></script>
<script src="js/lib/backbone.localStorage.js"></script>
<script src="js/models/todo.js"></script>
<script src="js/collections/todos.js"></script>
<script src="js/views/todos.js"></script>
<script src="js/views/app.js"></script>
<script src="js/routers/router.js"></script>
<script src="js/app.js"></script>
</body>
</html>
```

Обратите внимание, что помимо вышеупомянутых зависимостей также загружается еще несколько файлов, специфичных для приложения. Они организованы в папки, которые отражают их функции в приложении: модели, представления, коллекции и маршрутизаторы. Файл app.js содержит основной код инициализации.

Теперь создайте структуру каталогов, как показано в файле index.html:

1. Поместите файл index.html в каталог верхнего уровня.
2. Загрузите библиотеки jQuery, Underscore, Backbone и Backbone localStorage с соответствующих веб-сайтов и поместите их в каталог js/lib.
3. Создайте каталоги js/models, js/collections, js/views и js/routers.

Вам также понадобятся файлы base.css и bg.png, которые должны находиться в директории assets. Имейте в виду, что демонстрация финальной версии приложения доступна на сайте TodoMVC.com.

Далее мы создадим JavaScript-файлы нашего приложения. Не беспокойтесь о двух элементах text/template script — скоро мы заменим их.

HTML-код приложения

Сформируем тело файла index.html. Нам понадобится тег <input> для создания новых задач, тег <ul id="todo-list" /> для формирования списка задач и нижний колонтитул, в который мы позже вставим статистику и ссылки для выполнения таких операций, как удаление выполненных задач. Мы добавим следующую разметку непосредственно внутрь тега <body> перед элементами script:

```
<section id="todoapp">
  <header id="header">
    <h1>todos</h1>
    <input id="new-todo" placeholder="What needs to be done?" autofocus>
  </header>
  <section id="main">
    <input id="toggle-all" type="checkbox">
```

продолжение ↴

```

<label for="toggle-all">Mark all as complete</label>
<ul id="todo-list"></ul>
</section>
<footer id="footer"></footer>
</section>
<div id="info">
  <p>Double-click to edit a todo</p>
  <p>Written by <a href="https://github.com/addyosmani">Addy Osmani</a></p>
  <p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
</div>

```

Шаблоны

Для завершения создания файла index.html добавим в него шаблоны, с помощью которых мы будем динамически создавать HTML-код, вставляя данные моделей в соответствующие поля. Один из способов включить шаблоны в страницу — использовать настраиваемые теги `<script>`. Они не анализируются браузером, поскольку интерпретируются как простой текст. Затем функции библиотеки Underscore для работы с микрошаблонами могут получить доступ к этим шаблонам и отобразить фрагменты HTML.

Начнем с заполнения шаблона `#item-template`, с помощью которого будут отображаться отдельные задачи.

```

<!-- index.html -->
<script type="text/template" id="item-template">
  <div class="view">
    <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
    <label><%- title %></label>
    <button class="destroy"></button>
  </div>
  <input class="edit" value="<%- title %>">
</script>

```

В приведенной выше разметке теги шаблонов, такие как `<%=` и `<%-`, специфичны для библиотеки Underscore.js, документация о них имеется на сайте Underscore. Используйте в своих приложениях и другие библиотеки шаблонов, например Mustache или Handlebars. Выбирайте библиотеки по своему желанию; Backbone не воспрепятствует этому.

Нам также необходимо определить шаблон `#stats-template`, который будет использоваться для заполнения нижнего колонтитула.

```

<!-- index.html -->
<script type="text/template" id="stats-template">
  <span id="todo-count"><strong><%= remaining %></strong>
  <%= remaining === 1 ? 'item' : 'items' %> left</span>
  <ul id="filters">

```

```
<li>
    <a class="selected" href="#">All</a>
</li>
<li>
    <a href="#/active">Active</a>
</li>
<li>
    <a href="#/completed">Completed</a>
</li>
</ul>
<% if (completed) { %>
<button id="clear-completed">Clear completed (<%= completed %>)</button>
<% } %>
</script>
```

Шаблон `#stats-template` отображает число незавершенных задач и содержит список гиперссылок, с помощью которых можно будет выполнять действия, когда мы реализуем маршрутизатор. Кроме того, в шаблоне имеется кнопка, с помощью которой можно удалить все завершенные задачи.

Теперь, когда у нас есть весь необходимый HTML-код, мы начнем реализовывать наше приложение, вернувшись к основам — модели задачи.

Модель задачи

Модель задачи очень проста. Задача имеет два атрибута: заголовок `title` и флаг `completed`, который указывает, завершена ли задача. Эти атрибуты передаются по умолчанию, как показано ниже:

```
// js/models/todo.js
var app = app || {};
// модель задачи
// -----
// Модель задачи имеет атрибуты 'title', 'order' и 'completed'.
app.Todo = Backbone.Model.extend({
    // Атрибуты по умолчанию определяют, что у каждой созданной задачи будут ключи
    // `title` и `completed`.
    defaults: {
        title: '',
        completed: false
    },
    // переключение состояния задачи `completed`.
    toggle: function() {
        this.save({
            completed: !this.get('completed')
        });
    }
});
```

Кроме того, модель задачи содержит метод `toggle()`, с помощью которого можно задавать и сохранять атрибут завершения задачи.

Коллекция задач

Коллекция `TodoList` предназначена для группирования наших моделей. Она использует адаптер локального хранилища для переопределения Backbone-операции `sync()` по умолчанию; новая операция сохраняет наши задачи в локальном хранилище HTML5. Локальное хранилище хранит задачи между запросами к странице.

```
// js/collections/todos.js
var app = app || {};
// Коллекция задач
// -----
// Коллекция задач сохраняется в локальном хранилище,
// а не на удаленном сервере.
var TodoList = Backbone.Collection.extend({
    // Ссылка на модель этой коллекции.
    model: app.Todo,
    // Сохранение всех задач в пространстве имен `todos-backbone` .
    // Для работы этого кода потребуется, чтобы библиотека Backbone
    // загрузила плагин локального хранилища в вашу страницу. В противном случае
    // при тестировании кода в консоли закомментируйте
    // следующую строку, чтобы не вызвать исключение.
    localStorage: new Backbone.LocalStorage('todos-backbone'),
    // Фильтрация завершенных задач списка.
    completed: function() {
        return this.filter(function( todo ) {
            return todo.get('completed');
        });
    },
    // Фильтрация незавершенных задач списка.
    remaining: function() {
        // метод apply позволяет определить контекст указателя this
        // в области видимости нашей функции
        return this.without.apply( this, this.completed() );
    },
    // Мы поддерживаем последовательный порядок задач, хотя сохранение в базе
    // данных происходит по неупорядоченному GUID.
    // This генерирует следующий порядковый номер для новых элементов.
    nextOrder: function() {
        if ( !this.length ) {
            return 1;
        }
        return this.last().get('order') + 1;
    },
    // Задачи сортируются в порядке их ввода.
    comparator: function( todo ) {
```

```
        return todo.get('order');
    }
});
// Создание глобальной коллекции задач **Todos**.
app.Todos = new TodoList();
```

Методы `completed()` и `remaining()` этой коллекции возвращают массивы завершенных и незавершенных задач соответственно.

Метод `nextOrder()` реализует генератор последовательных чисел, а метод `comparator()` сортирует элементы в порядке их ввода.



Методы `this.filter`, `this.without` и `this.last` входят в состав библиотеки Underscore и включены в класс `Backbone.Collection`. Эта информация поможет читателю получить дополнительные данные об этих методах.

Представление приложения

Давайте изучим ключевую логику приложения, содержащуюся в его представлениях. Каждое представление поддерживает функции типа «редактирования на месте» и поэтому содержит существенный объем логики. Чтобы структурировать эту логику, мы воспользуемся паттерном контроллера элементов. Этот паттерн состоит из двух представлений, одно из которых управляет коллекцией элементов, а второе работает с отдельными элементами.

В нашем примере представление `AppView` займется созданием новых задач и отображением начального списка задач. Экземпляры представления `TodoView` будут связаны с отдельными задачами и смогут выполнять их редактирование, обновление и удаление.

Чтобы наш пример был простым и кратким, не будем реализовывать все возможности приложения в этой главе; мы рассмотрим лишь то, что вам потребуется для начала работы. Тем не менее представление `AppView` требует развернутых комментариев, поэтому разделим дальнейшее рассмотрение на две части.

```
// js/views/app.js
var app = app || {};
// наше приложение
// -----
// Представление AppView – верхний уровень пользовательского интерфейса.
app.AppView = Backbone.View.extend({
    // вместо того чтобы генерировать новый элемент, мы подключаемся
    // к существующему скелету приложения, имеющемуся в HTML.
```

продолжение ➔

```

el: '#todoapp',
// шаблон строки статистики в нижней части приложения.
statsTemplate: _.template( $('#stats-template').html() ),
// при инициализации мы делаем привязку
// к соответствующим событиям коллекции `Todos`
// при добавлении и изменении событий.
initialize: function() {
    this.allCheckbox = this.$('#toggle-all')[0];
    this.$input = this.$('#new-todo');
    this.$footer = this.$('#footer');
    this.$main = this.$('#main');
    this.listenTo(app.Todos, 'add', this.addOne);
    this.listenTo(app.Todos, 'reset', this.addAll);
},
// Добавление в список единственной задачи путем создания
// представления для нее и добавления ее элемента в `<ul>`.
addOne: function( todo ) {
    var view = new app.TodoView({ model: todo });
    $('#todo-list').append( view.render().el );
},
// Одновременное добавление всех элементов в коллекцию Todos.
addAll: function() {
    this.$('#todo-list').html('');
    app.Todos.each(this.addOne, this);
}
});

```

Начальная версия представления AppView содержит несколько возможностей, заслуживающих внимания, в том числе шаблон statsTemplate, метод initialize, который неявно вызывается при создании экземпляра представления, а также несколько методов, специфичных для представлений.

Свойство el (элемент) хранит селектор, указывающий на DOM-элемент с элементом id, равным todoapp. В нашем приложении el указывает на соответствующий элемент <section id="todoapp" /> в файле index.html.

Вызов _.template создает объект statsTemplate из шаблона #stats-template с помощью библиотеки Underscore. Воспользуемся этим шаблоном позже при отображении нашего представления.

Теперь обратимся к функции инициализации. Во-первых, она использует библиотеку jQuery для кэширования в локальных свойствах тех элементов, которые она будет использовать (вспомните, что this.\$() находит элементы, имеющие отношение к this.\$el). Во-вторых, эта функция подключается к двум событиям коллекции задач: add и reset. Поскольку мы делегируем обработку операций обновления и удаления представлению TodoView, нам не нужно работать с этими событиями в данном коде. Логика состоит из двух частей:

- Когда происходит событие add, вызывается метод addOne(), который получает новую модель. Метод addOne() создает экземпляр представления TodoView, отображает его и добавляет результирующий элемент в список задач.

- Когда происходит событие `reset` (мы обновляем коллекцию целиком при загрузке задач из локального хранилища), вызывается метод `addAll()`, который перебирает все задачи, имеющиеся в коллекции на текущий момент, и вызывает метод `addOne()` для каждой из них.

Обратите внимание, что мы использовали указатель `this` внутри вызова `addAll()` для ссылки на представление, поскольку метод `listenTo()` неявно перевел контекст обратного вызова на представление при создании привязки.

Теперь добавим еще немного логики, чтобы завершить создание представления `AppView`:

```
// js/views/app.js
var app = app || {};
// Наше приложение
// -----
// Представление AppView - верхний уровень пользовательского интерфейса.
app.AppView = Backbone.View.extend({
    // вместо того чтобы генерировать новый элемент,
    // мы подключаемся к существующему скелету
    // приложения, имеющемуся в HTML.
    el: '#todoapp',
    // шаблон для строки статистики в нижней части приложения.
    statsTemplate: _.template( $('#stats-template').html() ),
    // Новый код
    // Делегированные события для создания новых задач и удаления завершенных.
    events: {
        'keypress #new-todo': 'createOnEnter',
        'click #clear-completed': 'clearCompleted',
        'click #toggle-all': 'toggleAllComplete'
    },
    // При инициализации мы делаем привязку к соответствующим событиям
    // коллекции `Todos`,
    // когда ее элементы добавляются или изменяются. Мы начинаем
    // с загрузки существующих задач, которые могли быть сохранены
    // в локальное хранилище.
    initialize: function() {
        this.allCheckbox = this.$('#toggle-all')[0];
        this.$input = this.$('#new-todo');
        this.$footer = this.$('#footer');
        this.$main = this.$('#main');
        this.listenTo(app.Todos, 'add', this.addOne);
        this.listenTo(app.Todos, 'reset', this.addAll);
        // Новое
        this.listenTo(app.Todos, 'change:completed', this.filterOne);
        this.listenTo(app.Todos, 'filter', this.filterAll);
        this.listenTo(app.Todos, 'all', this.render);
        app.Todos.fetch();
    },
    // Новый код
    // Повторное отображение приложения означает лишь обновление статистики.
    // Остальная часть приложения не изменяется.
    render: function() {
```

продолжение ↗

```
var completed = app.Todos.completed().length;
var remaining = app.Todos.remaining().length;
if ( app.Todos.length ) {
    this.$main.show();
    this.$footer.show();
    this.$footer.html(this.statsTemplate({
        completed: completed,
        remaining: remaining
    }));
    this.$('#filters li a')
        .removeClass('selected')
        .filter(['[href="#/' + ( app.TodoFilter || '' ) + '"]')
        .addClass('selected');
} else {
    this.$main.hide();
    this.$footer.hide();
}
this.allCheckbox.checked = !remaining;
},
// Добавление одной задачи в список путем создания представления для нее
// и добавления ее элемента в `<ul>` .
addOne: function( todo ) {
    var view = new app.TodoView({ model: todo });
    $('#todo-list').append( view.render().el );
},
// Одновременное добавление всех элементов в коллекцию Todos.
addAll: function() {
    this.$('#todo-list').html('');
    app.Todos.each(this.addOne, this);
},
// Новое
filterOne : function (todo) {
    todo.trigger('visible');
},
// Новое
filterAll : function () {
    app.Todos.each(this.filterOne, this);
},
// Новое
// Генерация атрибутов для новой задачи.
newAttributes: function() {
    return {
        title: this.$input.val().trim(),
        order: app.Todos.nextOrder(),
        completed: false
    };
},
// Создание новой задачи и ее сохранение
// в локальном хранилище при нажатии return.
createOnEnter: function( event ) {
    if ( event.which !== ENTER_KEY || !this.$input.val().trim() ) {
        return;
    }
}
```

```
    app.Todos.create( this.newAttributes() );
    this.$input.val('');
},
// Новое
// Удаление всех завершенных задач уничтожением их моделей.
clearCompleted: function() {
  _.invoke(app.Todos.completed(), 'destroy');
  return false;
},
// Новое
toggleAllComplete: function() {
  var completed = this.allCheckbox.checked;
  app.Todos.each(function( todo ) {
    todo.save({
      'completed': completed
    });
  });
}
});
```

Мы добавили в приложение логику создания новых задач, их редактирования и фильтрации по состоянию завершенности.

Мы определили компонент `events`, включающий декларативные обратные вызовы для наших DOM-событий. Он связывает эти события со следующими методами:

○ `createOnEnter()`

Создает новую модель задачи и сохраняет ее в локальном хранилище, когда пользователь нажимает `Enter`, находясь в поле `<input>`. Этот метод также сбрасывает значение главного поля `<input>`, чтобы подготовить его к следующему вводу. Модель заполняется с помощью метода `newAttributes()`, который возвращает объект, составленный из заголовка, порядка и состояния завершения нового элемента. Обратите внимание, что указатель `this` ссылается на представление, а не на DOM-элемент, поскольку обратный вызов был привязан к компоненту `events`.

○ `clearCompleted()`

Удаляет из списка задачи, помеченные как завершенные, когда пользователь щелкает по флажку удаления завершенных задач (этот флажок будет находиться в нижнем колонтитуле, который формируется с помощью шаблона `#stats-template`).

○ `toggleAllComplete()`

Дает пользователю возможность пометить все элементы в списке задач как завершенные, щелкнув по флажку переключения состояния задач.

○ initialize()

Мы связали функции обратного вызова с некоторыми дополнительными событиями:

- Обратный вызов `filterOne()` коллекции задач связан с событием `change:completed`. Он прослушивает события изменения флага `completed` у любой из моделей коллекции. Модель, в которой произошло изменение, передается обратному вызову, который генерирует у нее настраиваемое событие `visible`.
- Мы связали с событием `filter` обратный вызов `filterAll()`, принцип действия которого напоминает `addOne()` и `addAll()`. Его цель — определять, какие задачи являются видимыми, в зависимости от текущего фильтра, выбранного в графическом интерфейсе (все задачи, только завершенные задачи или только незавершенные задачи), с помощью вызовов `filterOne()`.
- Мы использовали особое событие `all`, чтобы связать любое событие, генерируемое коллекцией задач, с методом отображения представления (который мы рассмотрим чуть позже).

В завершение метод `initialize()` считывает задачи, ранее сохраненные в локальном хранилище.

○ render() выполняет несколько действий:

1. Секции `#main` и `#footer` отображаются или скрываются в зависимости от наличия или отсутствия задач в списке.
2. Колонтитул заполняется HTML-кодом, который генерируется в результате создания экземпляра шаблона `statsTemplate`, и содержит информацию о количестве завершенных и незавершенных задач.
3. HTML-код, созданный на предыдущем шаге, содержит список ссылок фильтра. Значение `app.TodoFilter`, которое будет задано нашим маршрутизатором, используется для применения выбранного класса к ссылке, соответствующей выбранному в текущий момент фильтру. В результате этого действия к данному фильтру применяется соответствующий стиль CSS.
4. Обновление атрибута `allCheckbox` происходит в зависимости от наличия незавершенных задач.

Представления отдельных задач

Рассмотрим представление `TodoView`. Оно отвечает за отдельную задачу и обновляется при изменении ее свойств. Чтобы включить этот функционал

в работу, добавим в представление слушателей событий, происходящих в HTML-отображениях отдельных задач.

```
// js/views/todos.js
var app = app || {};
// представление задачи
// -----
// DOM-элемент задачи представляет собой...
app.TodoView = Backbone.View.extend({
    //... тег списка.
    tagName: 'li',
    // Кэширование функции шаблона для отдельного элемента.
    template: _.template( $('#item-template').html() ),
    // DOM-события, специфичные для элемента.
    events: {
        'dblclick label': 'edit',
        'keypress .edit': 'updateOnEnter',
        'blur .edit': 'close'
    },
    // представление TodoView прослушивает изменения своей модели
    // и выполняет повторное отображение. Поскольку в этом приложении
    // **Todo** и **TodoView** соотносятся как 1 к 1,
    // для удобства мы устанавливаем прямую ссылку на модель.
    initialize: function() {
        this.listenTo(this.model, 'change', this.render);
    },
    // Повторно отображает заголовки задачи.
    render: function() {
        this.$el.html( this.template( this.model.toJSON() ) );
        this.$input = this.$('.edit');
        return this;
    },
    // Переключение этого представления в режим редактирования,
    // отображение поля ввода.
    edit: function() {
        this.$el.addClass('editing');
        this.$input.focus();
    },
    // Закрытие режима редактирования, сохранение изменений в задаче.
    close: function() {
        var value = this.$input.val().trim();
        if ( value ) {
            this.model.save({ title: value });
        }
        this.$el.removeClass('editing');
    },
    // Если вы нажмете `enter`, то редактирование элемента завершится.
    updateOnEnter: function( e ) {
        if ( e.which === ENTER_KEY ) {
            this.close();
        }
    }
});
```

В конструкторе `initialize()` мы задаем слушателя, который наблюдает за событием `change` задачи. В результате при обновлении задачи приложение заново отображает представление и визуализирует изменения. Обратите внимание, что модель, переданная представлением `AppView` в наборе аргументов, автоматически доступна в виде элемента `this.model`.

В методе `render()` мы отображаем Underscore-шаблон `#item-template`, который ранее был скомпилирован в элемент `this.template` методом `_template()` библиотеки Underscore. Он возвращает HTML-фрагмент, заменяющий содержимое элемента представления (элемент `li` был неявно создан с помощью свойства `tagName`). Другими словами, отображенный шаблон теперь присутствует в элементе `this.el` и может быть добавлен в список задач в пользовательском интерфейсе. Метод `render()` завершается запоминанием элемента `input`, содержащегося в экземпляре шаблона, в переменную `this.input`.

Элемент `events` включает в себя три обратных вызова:

○ `edit()`

Переключает текущее представление в режим редактирования при двойном щелчке мышью по существующему элементу списка задач. Это позволяет пользователю изменить текущее название задачи.

○ `updateOnEnter()`

Проверяет, нажал ли пользователь клавишу `Return/Enter`, и вызывает функцию `close()`.

○ `close()`

Вырезает текущий текст из поля `<input/>` и отменяет его дальнейшую обработку, если поле не содержит текста (например, `"`). Если переданное значение корректно, то мы сохраняем изменения в модель текущей задачи и выходим из режима редактирования, удалив соответствующий класс CSS.

Запуск приложения

Итак, теперь у нас есть два представления: `AppView` и `TodoView`. Экземпляр первого представления нужно создать при загрузке страницы, чтобы ее код начал исполняться. Это можно сделать с помощью функции `ready()` библиотеки `jQuery`, которая запускает функцию при загрузке DOM.

```
// js/app.js
var app = app || {};
var ENTER_KEY = 13;
$(function() {
```

```
// начинаем с создания **App**.  
new app.AppView();  
});
```

Приложение в действии

Давайте возьмем паузу и убедимся, что сделанная к настоящему моменту работа приводит к нужным результатам.

Откройте ссылку `file:///*path*/index.html` в браузере и понаблюдайте за его консолью. Если все в порядке, то вы не должны видеть каких-либо JavaScript-ошибок, за исключением тех, которые относятся к еще не созданному нами файлу `router.js`. Список задач должен быть пустым, поскольку пока мы не создали ни одной задачи. Нам предстоит выполнить еще некоторую дополнительную работу, прежде чем наш пользовательский интерфейс заработает полностью.

Тем не менее потестируем некоторые его функции с помощью консоли JavaScript.

Добавьте в этой консоли новую задачу: `window.app.Todos.create({ title: 'My first Todo items'})`; и нажмите `Return` (рис. 4.2).



Рис. 4.2. Добавление новой задачи через консоль JavaScript

Если все работает правильно, то добавленная задача помещается в коллекцию. Кроме того, задача сохраняется в локальном хранилище и должна быть доступной при обновлении страницы.

Метод `window.app.Todos.create()` вызывает метод коллекции `Collection.create(attributes, [options])`, который создает экземпляр новой модели с типом, указанным в определении коллекции, — в нашем случае `app.Todo`:

```
// из файла js/collections/todos.js
var TodoList = Backbone.Collection.extend({
  model: app.Todo // тип модели, используемый методом collection.create()
  // для создания новой модели в коллекции
  ...
});
```

Запустите в консоли следующие команды, чтобы проверить его:

```
var secondTodo = window.app.Todos.create({ title: 'My second Todo item'});
secondTodo instanceof app.Todo // возвращает истину
```

Теперь обновите страницу — пред вами плоды упорной работы.

Задачи, добавленные через консоль, отображаются в списке, поскольку они считаются из локального хранилища. Кроме того, имеется возможность создать новую задачу, введя ее заголовок и нажав `Enter` (рис. 4.3).

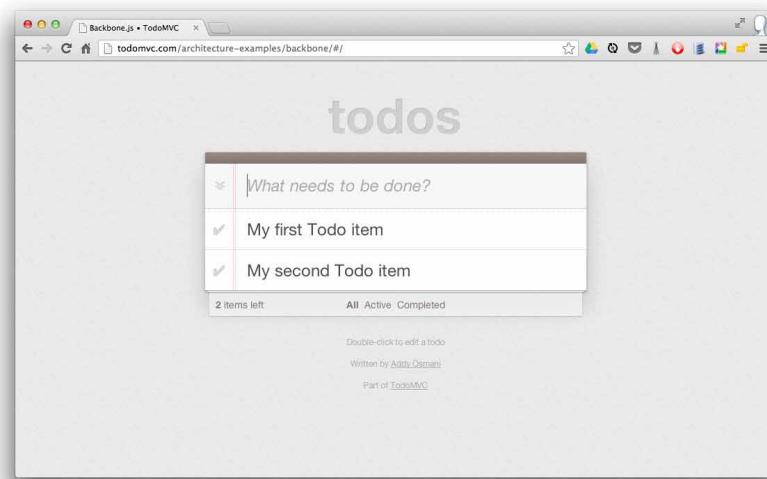


Рис. 4.3. Добавление новых задач

Отлично! Мы здорово продвинулись вперед, но как обстоят дела с завершением и удалением задач?

Завершение и удаление задач

Вторая часть нашего руководства посвящена завершению и удалению задач. Эти два действия специфичны для каждой задачи, поэтому мы должны включить данный функционал в представление TodoView. Для этого мы добавим методы `togglecompleted()` и `clear()` вместе с соответствующими записями в набор events.

```
// js/views/todos.js
var app = app || {};
// Представление задачи
// -----
// DOM-элемент задачи представляет собой...
app.TodoView = Backbone.View.extend({
    //... тег списка.
    tagName: 'li',
    // Кэширование функции шаблона для отдельного элемента.
    template: _.template( $('#item-template').html() ),
    // DOM-события, специфичные для элемента.
    events: {
        'click .toggle': 'togglecompleted', // Новый код
        'dblclick label': 'edit',
        'click .destroy': 'clear',           // Новый код
        'keypress .edit': 'updateOnEnter',
        'blur .edit': 'close'
    },
    // представление TodoView прослушивает изменения своей модели
    // и выполняет ее повторное отображение. Поскольку в этом приложении
    // **Todo** и **TodoView** соотносятся 1 к 1,
    // для удобства мы устанавливаем прямую ссылку на модель.

    initialize: function() {
        this.listenTo(this.model, 'change', this.render);
        this.listenTo(this.model, 'destroy', this.remove); // НОВОЕ
        this.listenTo(this.model, 'visible', this.toggleVisible); // НОВОЕ
    },
    // Повторное отображение заголовков задачи.
    render: function() {
        this.$el.html( this.template( this.model.toJSON() ) );
        this.$el.toggleClass( 'completed', this.model.get('completed') );
            // Новый код
        this.toggleVisible(); // Новый код
        this.$input = this$('.edit');
        return this;
    },
    // НОВОЕ: переключает видимость элемента
    toggleVisible : function () {
        this.$el.toggleClass( 'hidden', this.isHidden());
    },
},
```

продолжение ↗

```
// НОВОЕ – определяет, должен ли элемент быть скрытым
isHidden : function () {
    var isCompleted = this.model.get('completed');
    return ( // только для скрытых
        (!isCompleted && app.TodoFilter === 'completed')
        || (isCompleted && app.TodoFilter === 'active')
    );
},
// НОВОЕ: переключает состояние completed модели.
toggleCompleted: function() {
    this.model.toggle();
},
// Переключение этого представления в режим редактирования
// и отображение поля ввода.
edit: function() {
    this.$el.addClass('editing');
    this.$input.focus();
},
// Закрытие режима редактирования, сохранение изменений в задаче.
close: function() {
    var value = this.$input.val().trim();
    if ( value ) {
        this.model.save({ title: value });
    } else {
        this.clear(); // НОВОЕ
    }
    this.$el.removeClass('editing');
},
// Если вы нажмете `enter`, редактирование элемента завершится.
updateOnEnter: function( e ) {
    if ( e.which === ENTER_KEY ) {
        this.close();
    }
},
// НОВОЕ – удаление элемента, уничтожение модели
// в локальном хранилище и удаление ее представления
clear: function() {
    this.model.destroy();
}
});
```

Ключевыми фрагментами этого кода являются два добавленных обработчика событий: событие `toggleCompleted` флагжка задачи и событие `click` кнопки задачи `<button class="destroy" />`.

Давайте рассмотрим события, которые происходят при щелчке по флагжку задачи.

1. Вызывается функция `toggleCompleted()`, которая запускает метод `toggle()` в модели задачи.
2. Метод `toggle()` изменяет состояние завершения задачи и вызывает метод `save()` модели.

3. При сохранении генерируется событие `change` модели, связанное с методом `render()` представления `TodoView`. Мы добавили в метод `render()` оператор, который изменяет класс элемента в зависимости от состояния завершения модели. Соответствующая CSS изменяет цвет заголовка задачи и зачеркивает его текст, когда задача завершена.
4. При сохранении также генерируется событие `change:completed` модели, которое обрабатывается методом `filterOne()` представления `AppView`. Если мы еще раз посмотрим на представление `AppView`, то увидим, что метод `filterOne()` генерирует событие `visible` модели. Оно используется совместно с фильтрацией в наших маршрутах и коллекциях для того, чтобы элемент отображался только в случае, если состояние его завершения соответствует текущему фильтру. В обновленном представлении `TodoView` мы связали событие `visible` модели с методом `toggleVisible()`. Этот метод использует новый метод `isHidden()` для того, чтобы определить, следует ли отобразить задачу, и обновляет ее соответствующим образом.

Теперь давайте рассмотрим, что происходит при щелчке по кнопке удаления задачи.

1. Вызывается метод `clear()`, который запускает метод `destroy()` модели задачи.
2. Задача удаляется из локального хранилища, и генерируется событие `destroy`.
3. В обновленном представлении `TodoView` мы связали событие `destroy` модели с унаследованным методом `remove()` представления. Этот метод удаляет представление и автоматически удаляет связанный с ним DOM-элемент. Поскольку мы использовали метод `listenTo()`, чтобы связать слушателей представления с его моделью, метод `remove()` отключает от модели обратные вызовы, выполняющие прослушивание, чтобы избежать утечки памяти.
4. Метод `destroy()` также удаляет модель из коллекции задач, что приводит к генерации события `remove` в коллекции.
5. Поскольку метод `render()` представления `AppView` связан с событием `all` коллекции задач, представление `AppView` отображается, а статистика в колонтитуле обновляется.

Вот и всё!

Если вы хотите увидеть пример того, о чём мы говорили, изучите полный исходный код приложения.

Маршрутизация задач

Наконец, рассмотрим механизм маршрутизации, который позволяет легко отфильтровывать завершенные и незавершенные задачи (рис. 4.4). Мы будем поддерживать следующие маршруты:

```
#/ (all - default)
#/active
#/completed
```

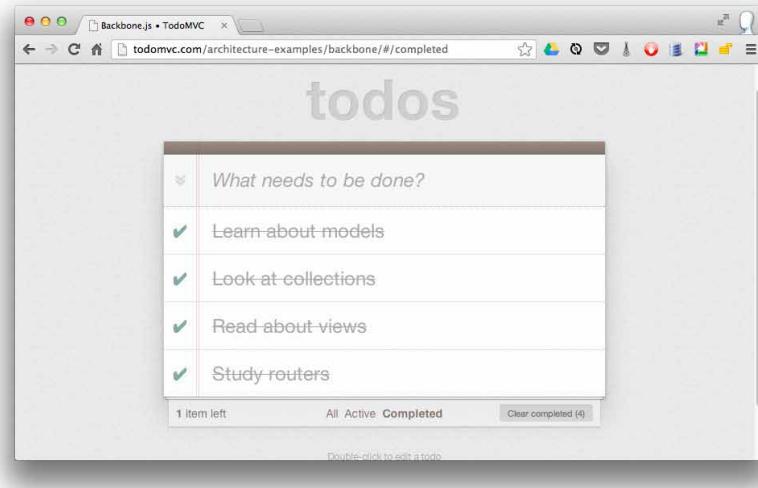


Рис. 4.4. Отфильтрованный список завершенных задач

При изменении маршрута список задач будет фильтроваться на уровне модели, и класс, выбранный с помощью ссылок фильтрации в колонтитуле страницы, будет изменяться так, как было описано. Если элемент обновляется при активном фильтре, то обновление происходит с учетом настроек фильтра (например, если фильтр включен и задача помечается как завершенная, то ее элемент скрывается). Активный фильтр сохраняется при перезагрузке страницы.

```
// js/routers/router.js
// маршрутатор задач
// -----
var Workspace = Backbone.Router.extend({
  routes:{
    '*filter': 'setFilter'
  },
  setFilter: function( param ) {
```

```
// задание текущего фильтра
// генерация события filter коллекции, вызывающего
// скрытие/отображение задач
window.app.Todos.trigger('filter');
}
});
app.TodoRouter = new Workspace();
Backbone.history.start();
```

Наш маршрутизатор использует звездочку (*) для задания маршрута по умолчанию, который передает строку после символов #/ в URL методу `setFilter()`, присваивающему эту строку переменной `window.app.TodoFilter`.

Как показано в строке `window.app.Todos.trigger('filter')`, после установки фильтра он используется к коллекции задач, чтобы поменять местами наборы отображаемых и скрываемых элементов. Вспомните, что метод `filterAll()` представления `AppView` связан с событием фильтрации коллекции и любое событие коллекции приведет к повторному отображению представления `AppView`.

Наконец, мы создаем экземпляр маршрутизатора и вызываем метод `Backbone.history.start()` для маршрутизации начального URL при загрузке страницы.

Выводы

Мы создали наше первое законченное приложение с использованием библиотеки Backbone.js. Вы найдете самую последнюю и полную версию этого приложения, а также ознакомитесь с его исходными текстами на сайте TodoMVC.

В главе 8 мы узнаем о том, как сделать структуру этого приложения более модульной с помощью библиотеки RequireJS, реализовать сохранение моделей в базу данных и провести модульное тестирование приложения с использованием нескольких тестовых фреймворков.

5

Упражнение 2: книжная библиотека — ваше первое RESTful-приложение на Backbone.js

Несмотря на то что первый пример дал нам хорошее представление о разработке приложений с помощью библиотеки Backbone.js, на практике многие приложения должны взаимодействовать с сервером баз данных. Давайте расширим наши знания с помощью еще одного примера — мы создадим *RESTful API*, к которому сможет обращаться наше приложение.

В этом примере мы разработаем с помощью Backbone приложение-библиотеку для управления электронными книгами. Мы будем хранить название, автора, дату издания и набор ключевых слов для каждой книги, а также отображать ее обложку.

Подготовительные действия

Сначала создадим структуру папок для нашего проекта. Чтобы отделить внутренние и внешние функции приложения друг от друга, мы создадим папку под названием *site* для нашего клиента в корневом каталоге проекта. Внутри этой папки мы создадим каталоги *css*, *img* и *js*.

Как и в последнем примере, мы разделим наши JavaScript-файлы по их функциям; для этого в каталоге *js* создайте подкаталоги с названиями *models*, *collections* и *views*. Ваша иерархия каталогов должна выглядеть следующим образом:

```
site/
    css/
    img/
    js/
```

```
collections/
lib/
models/
views/
```

Скачайте библиотеки Backbone, Underscore и jQuery и скопируйте их в папку js/lib. Нам также потребуется картинка, играющая роль книжных обложек. Сохраните картинку, изображенную на рис. 5.1, в папке site/img.



Рис. 5.1. Рисунок, который будет использоваться в качестве книжных обложек

Как и в предыдущем примере, нам потребуется загрузить все зависимости из файла site/index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <title>Backbone.js Library</title>
    <link rel="stylesheet" href="css/screen.css">
  </head>
  <body>
    <script src="js/lib/jquery.min.js"></script>
    <script src="js/lib/underscore-min.js"></script>
    <script src="js/lib/backbone-min.js"></script>
    <script src="js/models/book.js"></script>
    <script src="js/collections/library.js"></script>
    <script src="js/views/book.js"></script>
    <script src="js/views/library.js"></script>
    <script src="js/app.js"></script>
  </body>
</html>
```

Добавим HTML-код для пользовательского интерфейса. Мы создадим форму для добавления новой книги с помощью приведенного ниже кода, который следует поместить внутрь элемента body:

```
<div id="books">
  <form id="addBook" action="#">
    <div>
      <label for="coverImage">CoverImage: </label>
      <input id="coverImage" type="file" />
```

продолжение ↗

```
<label for="title">Title: </label><input id="title" type="text" />
<label for="author">Author: </label><input id="author" type="text" />
<label for="releaseDate">Release date: </label>
<input id="releaseDate" type="text" />
<label for="keywords">Keywords: </label>
<input id="keywords" type="text" />
<button id="add">Add</button>
</div>
</form>
</div>
```

Воспользуемся шаблоном для отображения каждой книги, его следует поместить перед тегами `<script>`:

```
<script id="bookTemplate" type="text/template">
  
  <ul>
    <li><%= title %></li>
    <li><%= author %></li>
    <li><%= releaseDate %></li>
    <li><%= keywords %></li>
  </ul>
  <button class="delete">Delete</button>
</script>
```

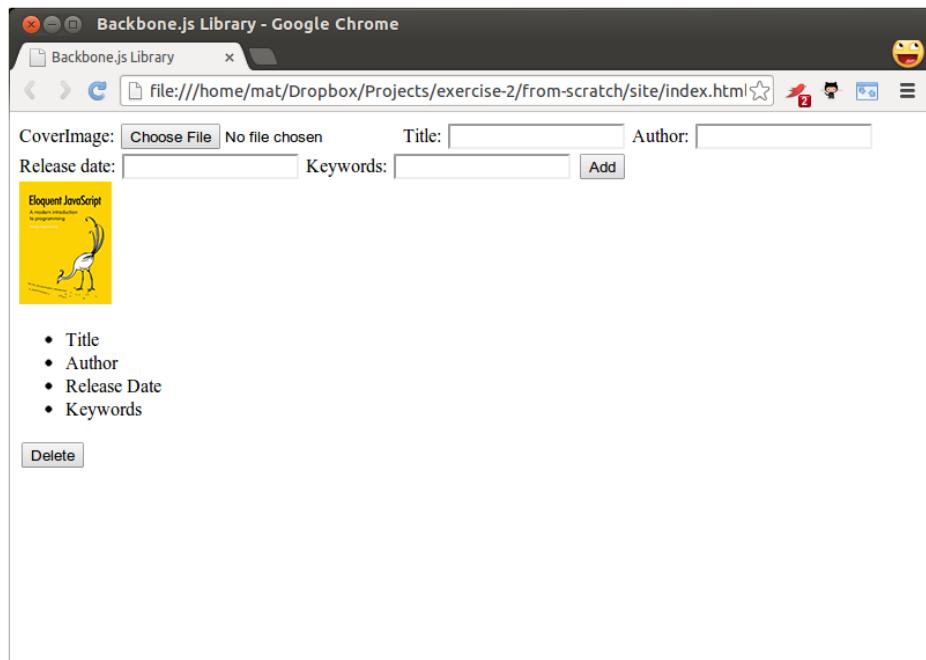


Рис. 5.2. Исходный вид приложения

Чтобы увидеть, как этот шаблон выглядит с введенными в него данными, вручную добавьте книгу с помощью div-элемента books.

```
<div class="bookContainer">
  
  <ul>
    <li>Title</li>
    <li>Author</li>
    <li>Release Date</li>
    <li>Keywords</li>
  </ul>
  <button class="delete">Delete</button>
</div>
```

Когда вы откроете этот файл в браузере, он должен выглядеть так, как показано на рис. 5.2.

Не слишком впечатляет... Конечно, мы не имеем дела с руководством по CSS, но нам все-таки нужно немного отформатировать элементы интерфейса. Создайте файл с именем screen.css в папке site/css:

```
body {
  background-color: #eee;
}
.bookContainer {
  outline: 1px solid #aaa;
  width: 350px;
  height: 130px;
  background-color: #fff;
  float: left;
  margin: 5px;
}
.bookContainer img {
  float: left;
  margin: 10px;
}
.bookContainer ul {
  list-style-type: none;
  margin-bottom: 0;
}
.bookContainer button {
  float: right;
  margin: 10px;
}
#addBook label {
  width: 100px;
  margin-right: 10px;
  text-align: right;
  line-height: 25px;
}
#addBook label, #addBook input {
  display: block;
  margin-bottom: 10px;
```

продолжение ↗

```
    float: left;
}
#addBook label[for="title"], #addBook label[for="releaseDate"] {
    clear: both;
}
#addBook button {
    display: block;
    margin: 5px 20px 10px 10px;
    float: right;
    clear: both;
}
#addBook div {
    width: 550px;
}
#addBook div:after {
    content: "";
    display: block;
    height: 0;
    visibility: hidden;
    clear: both;
    font-size: 0;
    line-height: 0;
}
```

Теперь интерфейс выглядит несколько лучше, это видно на рис. 5.3.

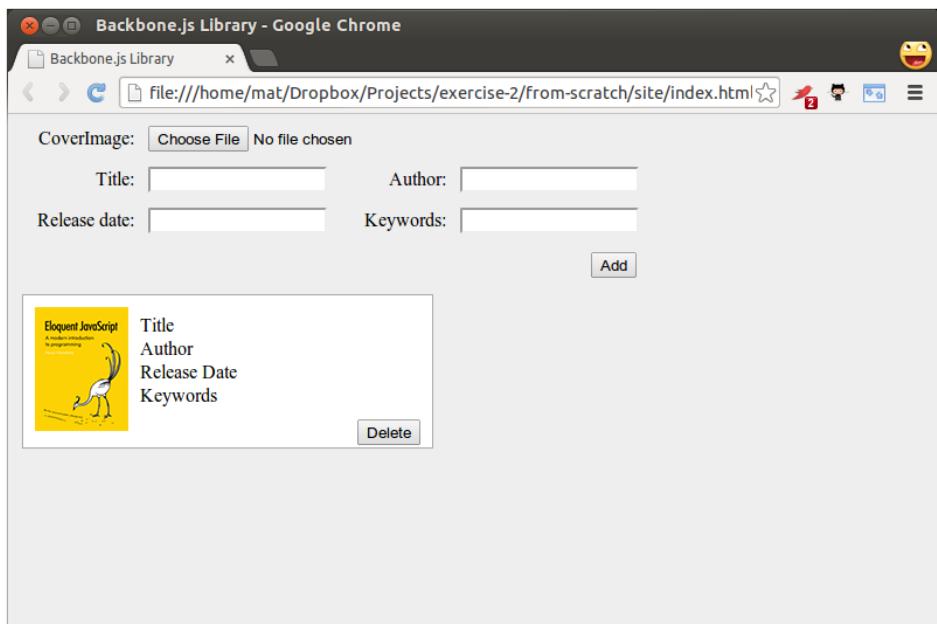


Рис. 5.3. Улучшенный пользовательский интерфейс нашего приложения

Новый результат выглядит неплохо, а теперь добавим больше книг. Скопируйте элемент `bookContainer` `div` несколько раз, чтобы оценить внешний вид интерфейса с несколькими отображенными книгами. Теперь мы готовы приступить к разработке самого приложения.

Создание модели, коллекции, представлений и объекта приложения

В первую очередь нам потребуется модель книги и коллекция для хранения списка книг. Обе они очень просты; в модели объявляются лишь несколько значений по умолчанию:

```
// site/js/models/book.js
var app = app || {};
app.Book = Backbone.Model.extend({
  defaults: {
    coverImage: 'img/placeholder.png',
    title: 'No title',
    author: 'Unknown',
    releaseDate: 'Unknown',
    keywords: 'None'
  }
});
// site/js/collections/library.js
var app = app || {};
app.Library = Backbone.Collection.extend({
  model: app.Book
});
```

Для отображения книг нам понадобится представление:

```
// site/js/views/book.js
var app = app || {};
app.BookView = Backbone.View.extend({
  tagName: 'div',
  className: 'bookContainer',
  template: _.template( $('#bookTemplate').html() ),
  render: function() {
    // tmpl - это функция, которая принимает JSON-объект и возвращает html
    // мы определили this.el в tagName. Используйте $el для доступа
    // к jQuery-функции html()
    this.$el.html( this.template( this.model.toJSON() ) );
    return this;
  }
});
```

Нам также потребуется представление для самого списка:

```
// site/js/views/library.js
var app = app || {};
app.LibraryView = Backbone.View.extend({
  el: '#books',
  initialize: function( initialBooks ) {
    this.collection = new app.Library( initialBooks );
    this.render();
  },
  // отображение библиотеки посредством вывода каждой книги из коллекции
  render: function() {
    this.collection.each(function( item ) {
      this.renderBook( item );
    }, this );
  },
  // отображение книги с помощью создания представления BookView
  // и добавления отображаемого элемента в элемент библиотеки
  renderBook: function( item ) {
    var bookView = new app.BookView({
      model: item
    });
    this.$el.append( bookView.render().el );
  }
});
```

Обратите внимание, что в функции инициализации принимается массив данных и передается конструктору `app.Library`. С его помощью мы заполним коллекцию пробными данными, чтобы убедиться, что все работает правильно. Наконец, у нас есть точка входа в код и данные для проверки:

```
// site/js/app.js
var app = app || {};
$(function() {
  var books = [
    { title: 'JavaScript: The Good Parts', author: 'Douglas Crockford',
      releaseDate: '2008', keywords: 'JavaScript Programming' },
    { title: 'The Little Book on CoffeeScript', author: 'Alex MacCaw',
      releaseDate: '2012', keywords: 'CoffeeScript Programming' },
    { title: 'Scala for the Impatient', author: 'Cay S. Horstmann',
      releaseDate: '2012', keywords: 'Scala Programming' },
    { title: 'American Psycho', author: 'Bret Easton Ellis',
      releaseDate: '1991', keywords: 'Novel Splatter' },
    { title: 'Eloquent JavaScript', author: 'Marijn Haverbeke',
      releaseDate: '2011', keywords: 'JavaScript Programming' }
  ];
  new app.LibraryView( books );
});
```

Наше приложение передает проверочные данные новому экземпляру представления `app.LibraryView`, который оно создает. Поскольку конструктор `initialize()` представления `LibraryView` вызывает его метод `render()`, все книги в библиотеке

выводятся на экран. Так как мы передаем точку входа библиотеке jQuery в качестве обратного вызова (в виде ее псевдонима \$), эта функция выполнится в момент готовности DOM.

Если вы откроете документ index.html в браузере, то увидите изображение, аналогичное рис. 5.4.

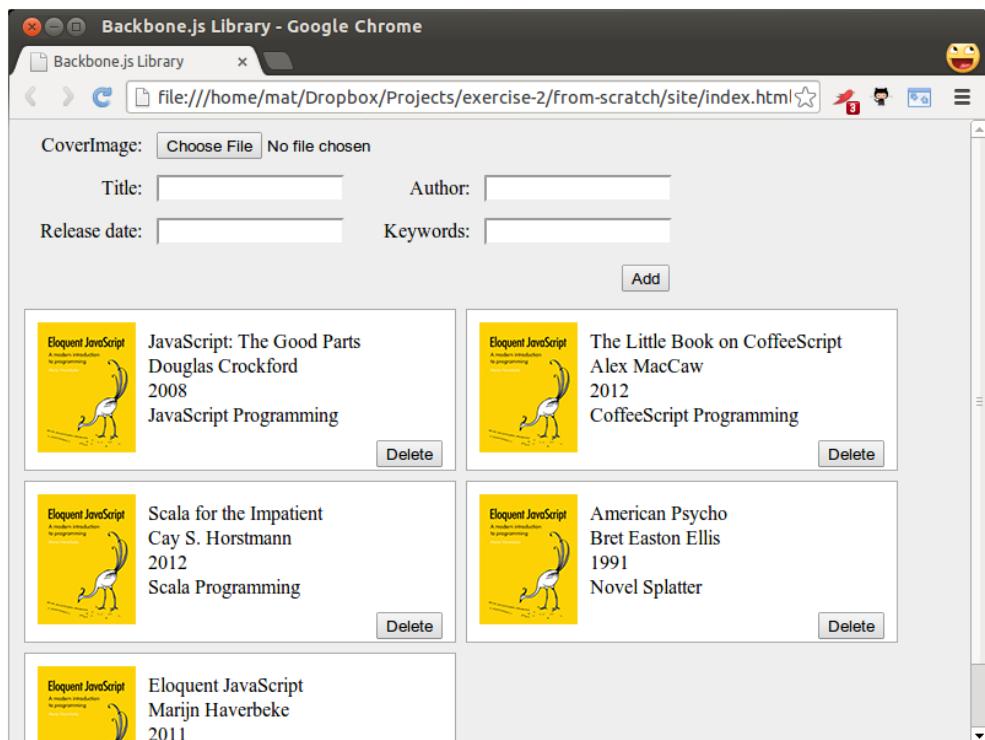


Рис. 5.4. Заполнение Backbone-приложения пробными данными

Итак, перед нами законченное Backbone-приложение, которое, правда, пока не делает ничего интересного.

Создание интерфейса

Теперь добавим функционал к форме в верхней части интерфейса, а также кнопку удаления для каждой книги.

Добавление моделей

При нажатии пользователем кнопки добавления книги мы должны принять данные формы и использовать их для создания новой модели. Добавим в представление `LibraryView` обработчик события щелчка:

```
events: {
  'click #add': 'addBook'
},
addBook: function( e ) {
  e.preventDefault();
  var formData = {};
  $( '#addBook div' ).children( 'input' ).each( function( i, el ) {
    if ( $( el ).val() != '' ) {
      formData[ el.id ] = $( el ).val();
    }
  });
  this.collection.add( new app.Book( formData ) );
},
```

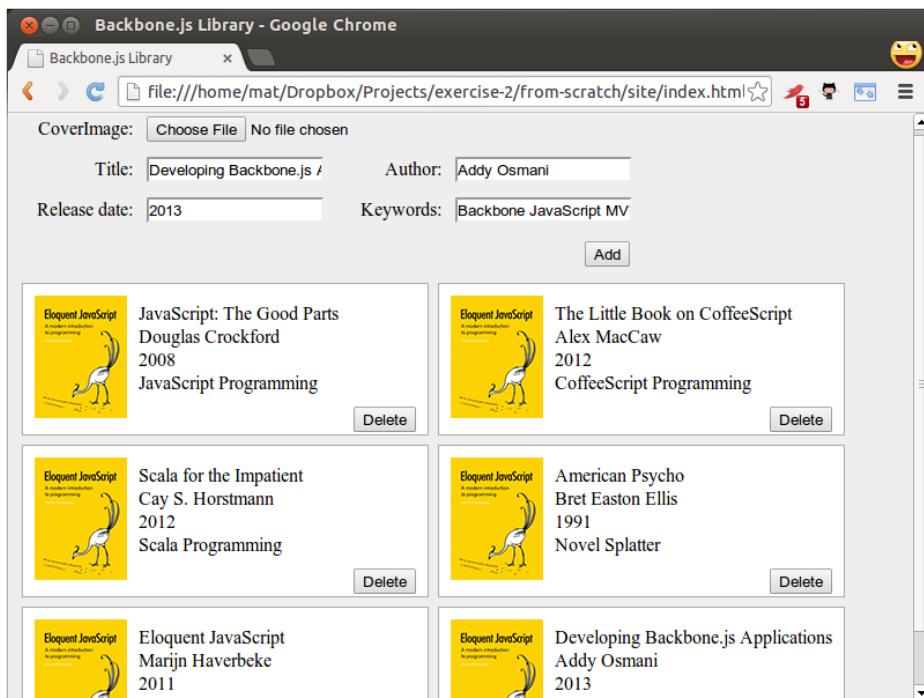


Рис. 5.5. Отображение представления при добавлении новой модели в коллекцию

Выбираем все элементы ввода формы, которые содержат данные, и перебираем их с помощью оператора `each` библиотеки `jQuery`. Поскольку мы использовали одни и те же имена для `id` нашей формы и ключей модели книги, сохраним их непосредственно в объекте `formData`. Затем создадим новую книгу с помощью данных и добавим ее в коллекцию. Мы пропускаем поля, не содержащие данных, чтобы были использованы значения по умолчанию.

Библиотека Backbone передает объект события в качестве параметра функции обработки события. В данном случае это устраивает нас, поскольку мы не хотим, чтобы форма выполняла отправку и перезагружала страницу. Вызов `preventDefault` события, добавленный в функцию `addBook`, заботится об этом.

Теперь заново отобразим представление при добавлении новой модели (рис. 5.5). Для этого мы помещаем следующую строку в функцию `initialize` представления `LibraryView`:

```
this.listenTo( this.collection, 'add', this.renderBook );
```

Теперь вы готовы заставить приложение «крутиться».

Возможно, вы увидите, что ввод файла с изображением обложки не работает, но я предлагаю вам решить эту проблему самостоятельно.

Удаление моделей

Далее создадим кнопку удаления. Настройте обработчика событий в представлении `BookView`:

```
events: {
  'click .delete': 'deleteBook'
},
deleteBook: function() {
  // удаление модели
  this.model.destroy();
  // удаление представления
  this.remove();
},
```

Теперь у вас должна появиться возможность добавлять и удалять книги из библиотеки.

Создание сервера базы данных

Давайте немного отклонимся от темы и создадим сервер с REST API. Поскольку эта книга посвящена JavaScript, для создания сервера мы воспользуемся

JavaScript-библиотекой Node.js. Если вы хотите создавать REST-сервер на другом языке, то помните, что он должен обеспечивать следующий интерфейс.

url	HTTP Method	Operation
/api/books	GET	Считывание массива книг
/api/books/:id	GET	Считывание книги с идентификатором :id
/api/books	POST	Добавление новой книги и ее возврат с добавленным атрибутом :id
/api/books/:id	PUT	Обновление книги с идентификатором :id
/api/books/:id	DELETE	Удаление книги с идентификатором :id

План наших действий в этом разделе выглядит следующим образом:

- установить *Node.js*, *npm* и *MongoDB*;
- установить модули узлов;
- создать простой веб-сервер;
- подключиться к базе данных;
- создать REST API.

Установка Node.js, npm и MongoDB

Скачайте и установите библиотеку Node.js с сайта Nodejs.org. Вместе с ней будет установлен менеджер пакетов узлов (node package manager, npm).

Скачайте и установите библиотеку MongoDB с сайта mongodb.org. На этом сайте вы найдете подробные инструкции по ее установке.

Установка модулей узлов

Создайте файл с названием `package.json` в корневой папке вашего проекта. Он должен выглядеть следующим образом:

```
{
  "name": "backbone-library",
  "version": "0.0.1",
  "description": "A simple library application using Backbone",
  "dependencies": {
    "express": "~3.1.0",
    "path": "~0.4.9",
    "mongoose": "~3.5.5"
  }
}
```

Этот файл выполняет ряд действий, в том числе указывает менеджеру пакетов узлов зависимости для нашего проекта. Находясь в корневой папке проекта, введите в командной строке следующую команду:

```
npm install
```

Вы должны увидеть, как менеджер пакетов узлов получает зависимости, перечисленные в файле package.json, и сохраняет их в папке с именем node_modules.

Структура ваших папок должна выглядеть следующим образом:

```
node_modules/
  .bin/
  express/
  mongoose/
  path/
site/
  css/
  img/
  js/
  index.html
  package.json
```

Создание простого веб-сервера

В корневой папке проекта создайте файл с именем server.js, содержащий следующий код:

```
// Зависимости модулей.
var application_root = __dirname,
    express = require( 'express' ), //Web framework
    path = require( 'path' ), //Utilities for dealing with file paths
    mongoose = require( 'mongoose' ); //MongoDB integration
// Создание сервера
var app = express();
// Конфигурирование сервера
app.configure( function() {
    //разбор тела запроса и заполнение request.body
    app.use( express.bodyParser() );
    //проверка request.body на переопределение HTTP-методов
    app.use( express.methodOverride() );
    //поиск маршрута по URL и HTTP-методу
    app.use( app.router );
    //где сохранить статическое содержимое
    app.use( express.static( path.join( application_root, 'site') ) );
    //показать все ошибки в разработке
    app.use( express.errorHandler({ dumpExceptions: true, showStack: true }));
});
//запуск сервера
var port = 4711;
app.listen( port, function() {
```

продолжение ↗

```
    console.log( 'Express server listening on port %d in %s mode',
      port, app.settings.env );
});
```

Начнем с загрузки модулей, необходимых для данного проекта: *Express* для создания HTTP-сервера, *Path* для обработки файловых путей и *mongoose* для подключения к базе данных.

Затем создадим Express-сервер и конфигурируем его с помощью анонимной функции. Это вполне стандартная конфигурация, и вызов *methodOverride* в нашем приложении на самом деле не нужен. Он используется для передачи HTTP-запросов **PUT** и **DELETE** непосредственно из формы, поскольку обычно формы поддерживают только методы **GET** и **POST**.

Наконец, мы запускаем сервер, вызывая функцию *listen*. В этом примере используется порт с номером **4711**, однако вы можете использовать любой свободный порт системы. Я выбрал порт **4711**, поскольку вероятность его использования другими программами очень невелика. Теперь мы готовы к запуску нашего первого сервера:

```
node server.js
```

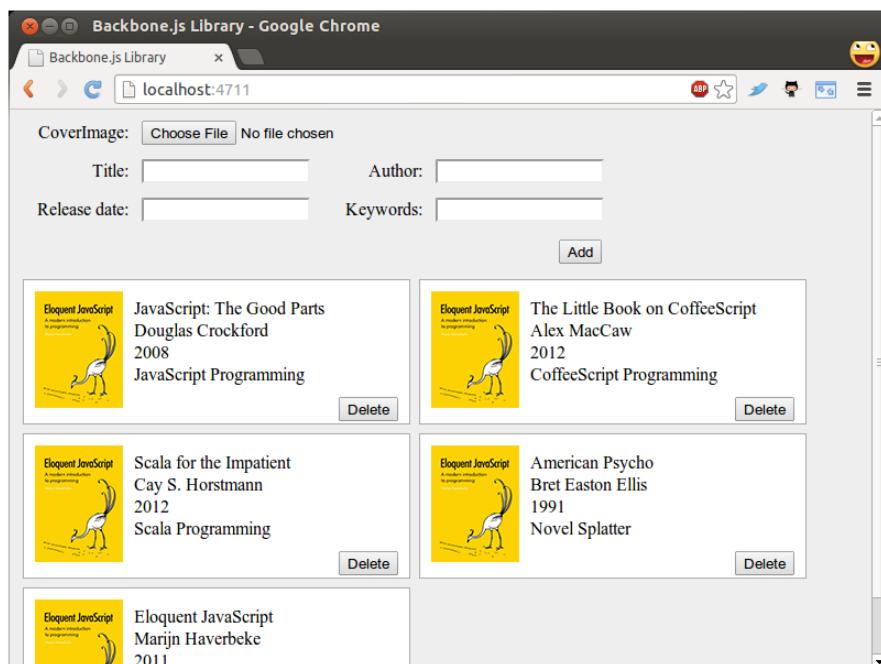


Рис 5.6. Наше Backbone-приложение, запущенное на сервере Express

Если вы откроете в браузере ссылку `http://localhost:4711`, то увидите изображение, аналогичное рис. 5.6.

Теперь наше приложение работает на сервере, а не запускается напрямую из файлов. Отлично! Начнем определять маршруты (URL), на которые сервер должен реагировать, — это будет наш REST API. Мы определяем маршруты с помощью объекта `app` и его методов `get`, `put`, `post` и `delete`, соответствующих одноименным HTTP-запросам. Вернемся к файлу `server.js` и определим простой маршрут:

```
// маршруты
app.get( '/api', function( request, response ) {
    response.send( 'Library API is running' );
});
```

Функция `get` принимает URL в качестве первого аргумента и функцию в качестве второго. В нашем случае функция вызывается с объектами запроса и ответа. Перезапустите библиотеку Node и перейдите по указанному URL (рис. 5.7).

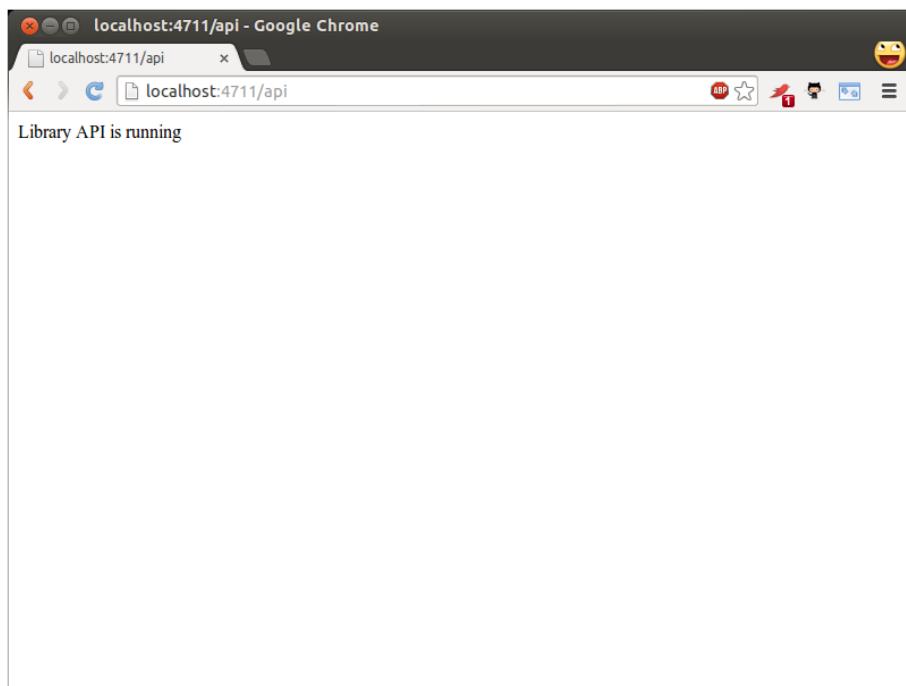


Рис. 5.7. Начальный ответ, полученный от нашего RESTful API

Подключение к базе данных

Фантастика! Теперь, поскольку мы хотим сохранить наши данные в базе данных MongoDB, нам необходимо определить схему. Добавьте в файл `server.js` следующий код:

```
//подключение к базе данных
mongoose.connect( 'mongodb://localhost/library_database' );
//схемы
var Book = new mongoose.Schema({
    title: String,
    author: String,
    releaseDate: Date
});
//модели
var BookModel = mongoose.model( 'Book', Book );
```

Как видите, определения схем вполне просты. Они могут быть более сложными, но для нас достаточно и этого. Я позаимствовал модель (`BookModel`) из СУБД Mongo. Вот то, с чем нам придется работать. Далее, мы определяем в REST API операцию GET, которая будет возвращать все книги:

```
//получение списка всех книг
app.get( '/api/books', function( request, response ) {
    return BookModel.find( function( err, books ) {
        if( !err ) {
            return response.send( books );
        } else {
            return console.log( err );
        }
    });
});
```

Функция `find` модели `BookModel` имеет определение `function find(conditions, fields, options, callback)`, но, поскольку мы хотим, чтобы она возвращала все книги, нам нужен только параметр обратного вызова. В обратный вызов будет передан объект ошибки и массив найденных объектов. При отсутствии ошибки мы возвращаем клиенту массив объектов с помощью функции `send` объекта ответа; в противном случае мы выводим сообщение об ошибке на консоль.

Для тестирования API нам потребуется ввести несколько команд в консоль JavaScript. Перезапустите библиотеку Node и перейдите по ссылке `localhost:4711` в вашем браузере. Откройте консоль JavaScript. Если вы используете Google Chrome, выберите в меню команду `View ▶ Developer ▶ JavaScript Console` (Инструменты ▶ Консоль JavaScript). Если вы используете Firefox, установите Firebug и воспользуйтесь командой `View ▶ Firebug` (Веб-разработка ▶ Firebug). Большинство других браузеров содержат аналогичную консоль.

В консоли введите следующие команды:

```
jQuery.get( '/api/books/' , function( data, textStatus, jqXHR ) {
    console.log( 'Get response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});
```

и нажмите Enter. Вы должны увидеть изображение, аналогичное рис. 5.8.

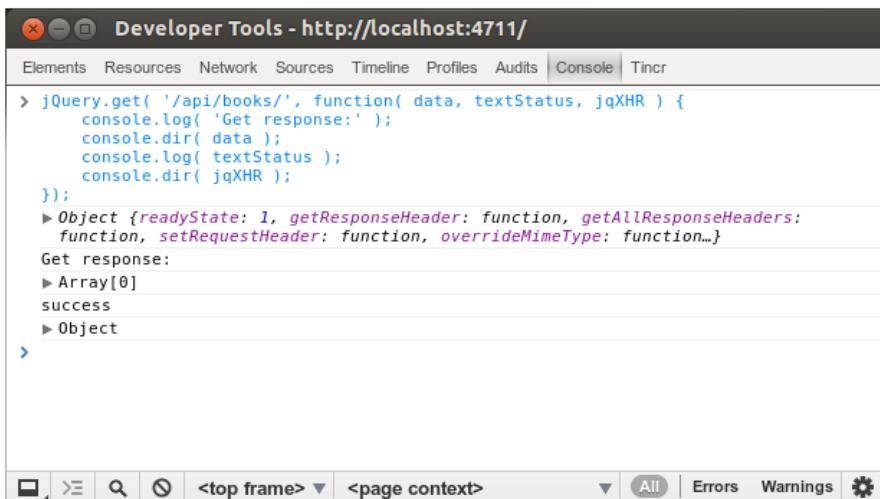


Рис. 5.8. Обращение к REST API с помощью библиотеки jQuery

Я использовал jQuery для того, чтобы обратиться к нашему REST API, поскольку он уже загружен на страницу. Разумеется, возвращенный массив пуст, поскольку мы еще ничего не поместили в базу данных. Теперь создадим POST-маршрут, который позволяет добавлять новые элементы в server.js:

```
//добавление новой книги
app.post( '/api/books' , function( request, response ) {
    var book = new BookModel({
        title: request.body.title,
        author: request.body.author,
        releaseDate: request.body.releaseDate
    });
    book.save( function( err ) {
        if( !err ) {
            return console.log( 'created' );
        } else {
```

продолжение ↗

```

        return console.log( err );
    }
});
return response.send( book );
});

```

Начнем с того, что создадим новую модель BookModel, передавая объект с атрибутами title, author и releaseDate. Получение данных происходит от объекта request.body. Это означает, что при любом вызове этой операции в API необходимо предоставить JSON-объект, содержащий атрибуты title, author и releaseDate. На самом деле, инициатор вызова может опускать любые атрибуты (в том числе все), поскольку мы не сделали ни один из них обязательным.

Затем мы вызываем функцию save модели BookModel, передавая обратный вызов так же, как в предыдущем маршруте get. В конце мы возвращаем сохраненную модель BookModel. Причина, по которой мы возвращаем объект BookModel, а не просто признак успешного завершения операции, в том, что при сохранении BookModel получает от базы данных MongoDB атрибут id, который необходим клиенту при обновлении и удалении определенной книги. Запустим приложение еще раз: для этого перезапустите Node, вернитесь в консоль и введите следующий текст:

```

jQuery.post( '/api/books', {
    'title': 'JavaScript the good parts',
    'author': 'Douglas Crockford',
    'releaseDate': new Date( 2008, 4, 1 ).getTime()
}, function(data, textStatus, jqXHR) {
    console.log( 'Post response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});

```

и затем:

```

jQuery.get( '/api/books/', function( data, textStatus, jqXHR ) {
    console.log( 'Get response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});

```

Теперь вы должны получить от сервера одноэлементный массив. Возможно, вам непонятна следующая строка:

```
'releaseDate': new Date(2008, 4, 1).getTime()
```

База данных MongoDB принимает даты в UNIX-формате (число миллисекунд с полуночи 1 января 1970 г. по Гринвичу), поэтому даты необходимо преобразовывать перед отправкой. Тем не менее возвращаемый объект содержит в себе JavaScript-объект Date. Также обратите внимание на атрибут `_id` возвращенного объекта (рис. 5.9).

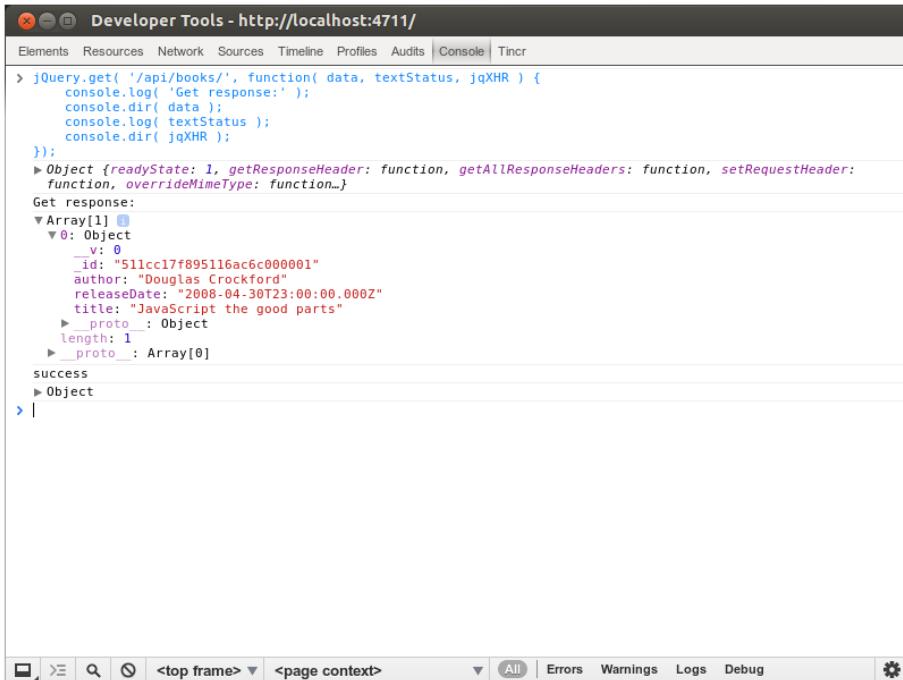


Рис. 5.9. Вид структуры возвращенной модели BookModel

Теперь давайте создадим в файле `server.js` запрос GET, который получает одну книгу:

```
//получение одной книги по id
app.get( '/api/books/:id', function( request, response ) {
  return BookModel.findById( request.params.id, function( err, book ) {
    if( !err ) {
      return response.send( book );
    } else {
      return console.log( err );
    }
  });
});
```

Двоеточие (:id) указывает серверу Express, что эта часть маршрута является динамической. Мы также используем функцию `findById` модели `BookModel`, чтобы получить один результат. Перезапустив библиотеку Node, можно получить в ответ единственную книгу, добавив ранее возвращенный `id` к URL следующим образом:

```
jQuery.get( '/api/books/4f95a8cb1baa9b8a1b000006',
  function( data, textStatus, jqXHR ) {
    console.log( 'Get response:' );
    console.dir( data );
    console.log( textStatus );
    console.dir( jqXHR );
});
```

Теперь создадим PUT-функцию (обновление):

```
//Обновление книги
app.put( '/api/books/:id', function( request, response ) {
  console.log( 'Updating book ' + request.body.title );
  return BookModel.findById( request.params.id, function( err, book ) {
    book.title = request.body.title;
    book.author = request.body.author;
    book.releaseDate = request.body.releaseDate;
    return book.save( function( err ) {
      if( !err ) {
        console.log( 'book updated' );
      } else {
        console.log( err );
      }
      return response.send( book );
    });
  });
});
```

Этот пример длиннее, чем предыдущие, но тоже совсем несложный: мы находим книгу по `id`, обновляем ее свойства, сохраняем ее и отсылаем обратно клиенту.

Чтобы протестировать этот код, воспользуемся более общей jQuery-функцией `ajax`. В следующих примерах снова замените `id` на идентификатор элемента в вашей базе данных:

```
jQuery.ajax({
  url: '/api/books/4f95a8cb1baa9b8a1b000006',
  type: 'PUT',
  data: {
    'title': 'JavaScript The good parts',
    'author': 'The Legendary Douglas Crockford',
    'releaseDate': new Date( 2008, 4, 1 ).getTime()
  },
});
```

```

        success: function( data, textStatus, jqXHR ) {
            console.log( 'Post response:' );
            console.dir( data );
            console.log( textStatus );
            console.dir( jqXHR );
        }
    });

```

Наконец, создадим маршрут для удаления:

```

//удаление книги
app.delete( '/api/books/:id', function( request, response ) {
    console.log( 'Deleting book with id: ' + request.params.id );
    return BookModel.findById( request.params.id, function( err, book ) {
        return book.remove( function( err ) {
            if( !err ) {
                console.log( 'Book removed' );
                return response.send( '' );
            } else {
                console.log( err );
            }
        });
    });
});

```

и проверим результат:

```

jQuery.ajax({
    url: '/api/books/4f95a5251baa9b8a1b000001',
    type: 'DELETE',
    success: function( data, textStatus, jqXHR ) {
        console.log( 'Post response:' );
        console.dir( data );
        console.log( textStatus );
        console.dir( jqXHR );
    }
});

```

Итак, наш REST API готов — мы обеспечили поддержку всех четырех HTTP-методов. Что дальше? До настоящего момента я не касался ключевых слов, связанных с книгой. Работать с ключевыми словами несколько сложнее, поскольку книга может иметь несколько ключевых слов, и мы хотим представлять их не как одну строку, а в виде массива строк. Для этого нам понадобится другая схема.

Добавим схему `Keywords` непосредственно перед схемой `Book`:

```

//схемы
var Keywords = new mongoose.Schema({
    keyword: String
});

```

Чтобы добавить подсхему в существующую схему, используем квадратные скобки:

```
var Book = new mongoose.Schema({
  title: String,
  author: String,
  releaseDate: Date,
  keywords: [ Keywords ] // НОВОЕ
});
```

Также обновим методы POST и PUT:

```
// добавление новой книги
app.post( '/api/books' , function( request, response ) {
  var book = new BookModel({
    title: request.body.title,
    author: request.body.author,
    releaseDate: request.body.releaseDate,
    keywords: request.body.keywords // НОВОЕ
  });
  book.save( function( err ) {
    if( !err ) {
      return console.log( 'created' );
    } else {
      return console.log( err );
    }
  });
  return response.send( book );
});

// обновление книги
app.put( '/api/books/:id' , function( request, response ) {
  console.log( 'Updating book ' + request.body.title );
  return BookModel.findById( request.params.id, function( err, book ) {
    book.title = request.body.title;
    book.author = request.body.author;
    book.releaseDate = request.body.releaseDate;
    book.keywords = request.body.keywords; // НОВОЕ
    return book.save( function( err ) {
      if( !err ) {
        console.log( 'book updated' );
      } else {
        console.log( err );
      }
      return response.send( book );
    });
  });
});
```

Готово! Мы сделали все, что требовалось. Опробуем результат в консоли:

```
jQuery.post( '/api/books' , {
  'title': 'Secrets of the JavaScript Ninja',
```

```
'author': 'John Resig',
'releaseDate': new Date( 2008, 3, 12 ).getTime(),
'keywords':[  
    { 'keyword': 'JavaScript' },  
    { 'keyword': 'Reference' }  
]  
, function( data, textStatus, jqXHR ) {  
    console.log( 'Post response:' );  
    console.dir( data );  
    console.log( textStatus );  
    console.dir( jqXHR );  
});
```

Теперь у вас есть полнофункциональный REST-сервер, к которому можно подключиться из внешнего интерфейса приложения.

Взаимодействие с сервером

В этом разделе мы рассмотрим подключение нашего Backbone-приложения к серверу с использованием REST API.

Как я упоминал в главе 3, мы можем считывать модели с сервера с помощью метода `collection.fetch()`, присваивая переменной `collection.url` URL конечной точки API. Давайте обновим коллекцию нашей библиотеки, добавив в нее эту возможность:

```
var app = app || {};  
app.Library = Backbone.Collection.extend({  
    model: app.Book,  
    url: '/api/books' // НОВОЕ  
});
```

В результате мы получим реализацию `Backbone.sync` по умолчанию, считая, что API выглядит следующим образом.

URL	HTTP-метод	Операция
/api/books	GET	Считывание массива всех книг
/api/books/:id	GET	Считывание книги с идентификатором :id
/api/books	POST	Добавление новой книги, возврат книги с добавленным атрибутом id
/api/books/:id	PUT	Обновление книги с идентификатором :id
/api/books/:id	DELETE	Удаление книги с идентификатором :id

Чтобы наше приложение считывало модели книг с сервера при загрузке страницы, обновим представление `LibraryView`. Документация библиотеки Backbone рекомендует добавлять все модели при генерации страницы на сервере, а не запрашивать их с клиентской стороны после загрузки страницы. Поскольку эта глава нацелена на то, чтобы дать как можно более полную картину взаимодействия с сервером, мы проигнорируем эту рекомендацию.

Перейдите к объявлению представления `LibraryView` и измените функцию `initialize` следующим образом:

```
initialize: function() {
  this.collection = new app.Library();
  this.collection.fetch({reset: true}); // НОВОЕ
  this.render();
  this.listenTo( this.collection, 'add', this.renderBook );
  this.listenTo( this.collection, 'reset', this.render ); // НОВОЕ
},
```

Теперь, когда мы заполняем библиотеку из базы данных с помощью метода `this.collection.fetch()`, функция `initialize()` больше не принимает набор проверочных данных в качестве аргумента и ничего не передает конструктору `app.Library`. Удалите пробные данные из файла `site/js/app.js`, сократив его до единственного оператора, который создает представление `LibraryView`:

```
// site/js/app.js
var app = app || {};
$(function() {
  new app.LibraryView();
});
```

Мы также добавили слушателя события `reset`. Это необходимо сделать потому, что после отображения страницы моделичитываются асинхронно. После завершения считывания библиотека Backbone генерирует событие очистки согласно параметру `reset: true`, и наш слушатель заново отображает представление. Перезагрузите страницу, и вы увидите все книги, которые хранятся на сервере, как показано на рис. 5.10.

Обратите внимание: дата публикации книги и ключевые слова выглядят странно. Дата, полученная с сервера, преобразуется в JavaScript-объект `Date`, и, когда она применяется к шаблону Underscore, он использует функцию `toString()` для ее отображения. В языке JavaScript отсутствует развитая поддержка форматирования дат, поэтому для решения этой проблемы воспользуемся jQuery-плагином `dateFormat`. Скачайте и поместите его в папку `site/js/lib`. Измените шаблон книги так, чтобы дата отображалась с помощью следующей конструкции:

```
<li><%= $.format.date( new Date( releaseDate ), 'MMMM yyyy' ) %></li>
```

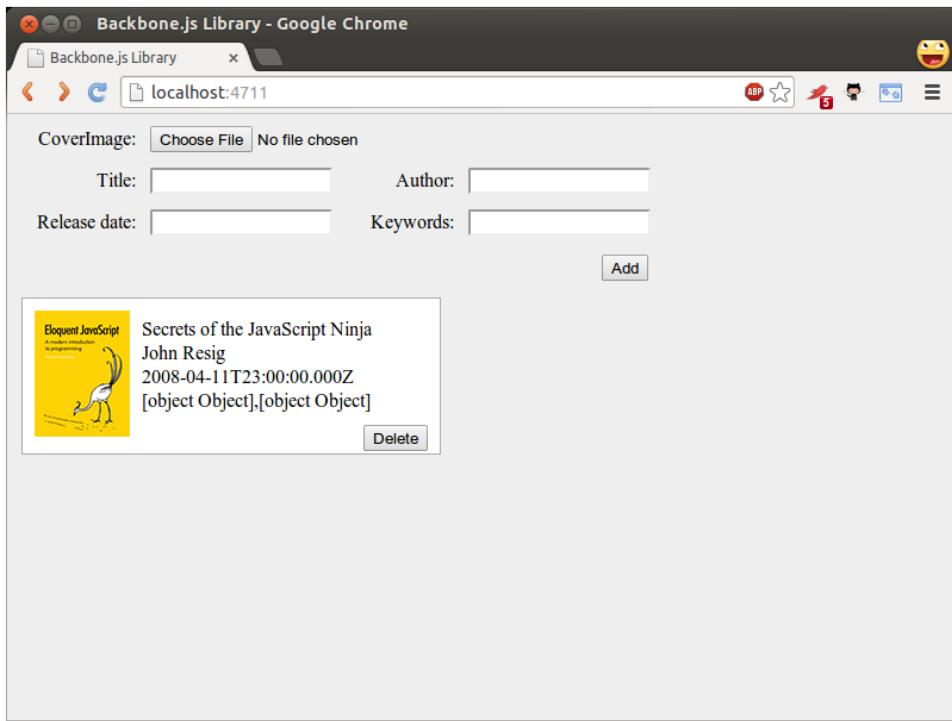


Рис. 5.10. При перезагрузке страницы отображаются книги, хранящиеся на сервере

Добавьте элемент `script` для плагина:

```
<script src="js/lib/jquery-dateFormat-1.0.js"></script>
```

Теперь дата на странице должна выглядеть несколько лучше. А что у нас с ключевыми словами? Поскольку мы получаем их в виде массива, необходимо выполнить код, который генерирует строку разделенных между собой ключевых слов. Для этой цели опустите знак равенства в теге шаблона, это позволит выполнить код, который ничего не отображает:

```
<li><% _.each( keywords, function( keyobj ) {%
<%= keyobj.keyword %><% } ); %></li>
```

Здесь я перебираю ключевые слова из массива с помощью Underscore-функции `each` и печатаю каждое слово. Обратите внимание, что при отображении ключевых слов используется синтаксис библиотеки Underscore — ключевые слова отображаются с пробелами между ними.

После перезагрузки страницы ее внешний вид должен улучшиться, как показано на рис. 5.11.

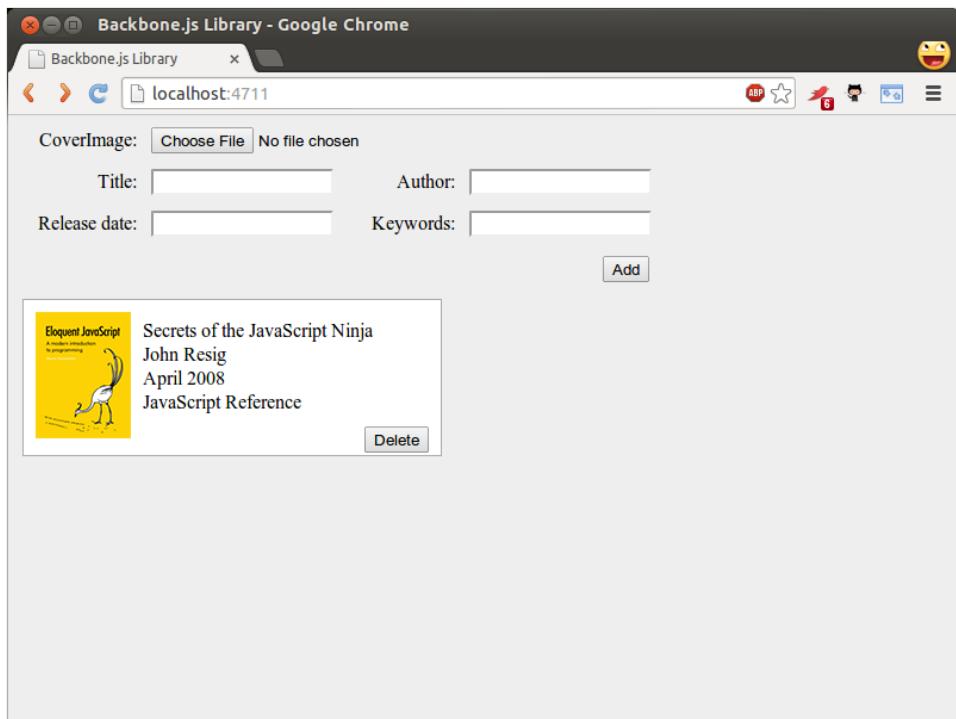


Рис. 5.11. Улучшенное форматирование даты

Теперь удалите книгу и перезагрузите страницу.

Сюрприз! Удаленная книга появилась снова. Это совсем не то, что нам надо; почему так произошло?

Когда мы загружаем модели книг с сервера, у них есть атрибут `_id` (обратите внимание на символ подчеркивания), однако Backbone ждет атрибута `id` (без символа подчеркивания). Поскольку такой атрибут отсутствует, Backbone рассматривает модель как новую, а при удалении новой модели не требуется синхронизация.

Для исправления этой ошибки воспользуемся функцией `parse` объекта `Backbone.Model`. Функция `parse` дает возможность редактировать ответ сервера до того, как он передается конструктору модели. Добавьте функцию `parse` в модель `BookModel`:

```
parse: function( response ) {
  response.id = response._id;
  return response;
}
```

Просто скопируйте значение `_id` в нужный атрибут `id`. При перезагрузке страницы вы увидите, что при щелчке на кнопке удаления модели уничтожаются на сервере.

Другой, более простой способ заставить библиотеку Backbone распознать `_id` как уникальный идентификатор — установить атрибут `idAttribute` модели равным `_id`.

Если теперь вы попробуете добавить новую книгу с помощью формы, то увидите ту же ситуацию, что наблюдалась при удалении: модели не сохраняются на сервер. Это объясняется тем, что метод `Backbone.Collection.add` не выполняет синхронизацию автоматически, однако эту проблему легко устранить. В представлении `LibraryView`, которое находится в файле `views/library.js`, замените строку

```
this.collection.add( new Book( formData ) );
```

на строку

```
this.collection.create( formData );
```

Теперь создаваемые книги будут сохраняться на сервере. На самом деле, они могут и не сохраниться, если вы введете дату. Сервер ждет дату в формате штампа времени UNIX (количество миллисекунд с 1 января 1970 г.). Кроме того, никакие ключевые слова, которые вы вводите, не сохраняются на сервере, поскольку сервер ожидает массив объектов с атрибутом ключевого слова.

Начнем с того, что исправим ошибку с датой. На самом деле, мы не хотим, чтобы пользователи вручную вводили дату в определенном формате, поэтому воспользуемся стандартным календарем `datepicker` из пользовательского интерфейса jQuery. Загрузите конфигурацию пользовательского интерфейса jQuery, содержащую `datepicker`. Добавьте css-тему в папку `site/css/` и JavaScript в папку `site/js/lib`. Создайте ссылки на них в файле `index.html`:

```
<link rel="stylesheet" href="css/cupertino/jquery-ui-1.10.0.custom.css">
```

(*cupertino* — это имя стиля, который я выбрал при загрузке пользовательского интерфейса jQuery).

JavaScript-файл должен загружаться после jQuery.

```
<script src="js/lib/jquery.min.js"></script>
<script src="js/lib/jquery-ui-1.10.0.custom.min.js"></script>
```

Свяжите datepicker с нашим полем releaseDate в файле app.js:

```
var app = app || {};
$(function() {
    $('#releaseDate').datepicker();
    new app.LibraryView();
});
```

Теперь у вас должна появиться возможность выбрать дату, когда вы щелкаете по полю releaseDate, как показано на рис. 5.12.

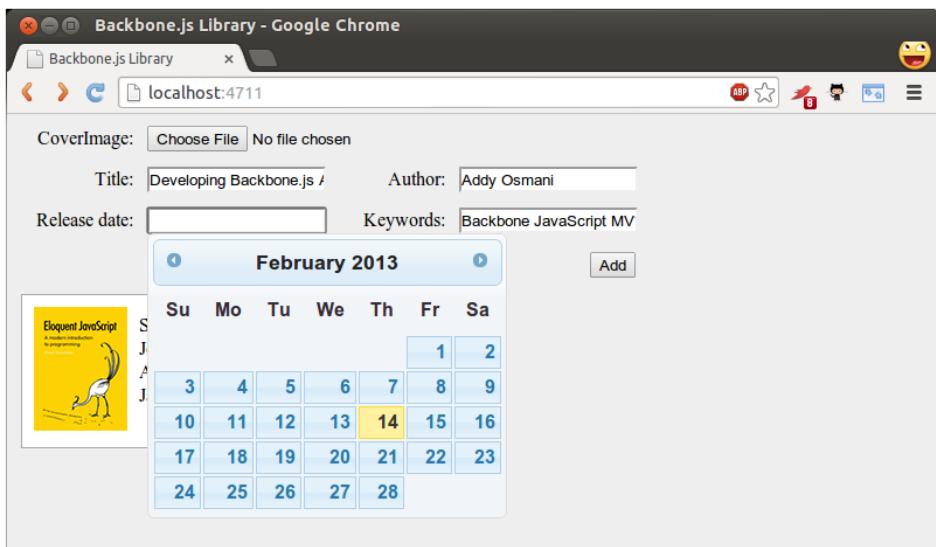


Рис. 5.12. Выбор даты из поля releaseDate в нашем приложении

Наконец, нам необходимо, чтобы данные, введенные в форму, правильно преобразовывались в формат нашего хранилища. Измените функцию addBook представления LibraryView следующим образом:

```
addBook: function( e ) {
    e.preventDefault();
    var formData = {};
    $('#addBook div').children('input').each( function( i, el ) {
        if( $( el ).val() != '' )
        {
            if( el.id === 'keywords' ) {
                formData[ el.id ] = [];
                _each( $( el ).val().split( ' ' ), function( keyword ) {
                    formData[ el.id ].push({ 'keyword': keyword });
                });
            }
        }
    });
}
```

```

        });
    } else if( el.id === 'releaseDate' ) {
        formData[ el.id ] = $( '#releaseDate' ).datepicker(
            'getDate' ).getTime();
    } else {
        formData[ el.id ] = $( el ).val();
    }
}
// очистка значения поля ввода
$( el ).val('');
});
this.collection.create( formData );
},

```

Мы добавили две проверки полей ввода формы. Сначала проверяем, является ли текущий элемент полем для ввода ключевых слов: в этом случае мы делим строку на части по пробелам и создаем массив объектов ключевых слов.

Затем мы проверяем, является ли текущий элемент полем ввода `releaseDate`, и в этом случае вызываем функцию `datePicker("getDate")`, которая возвращает объект Date. Затем применяем к этому объекту функцию `getTime`, чтобы получить время в миллисекундах.

Теперь у вас должна появиться возможность добавлять новые книги, указывая и дату издания, и ключевые слова (рис. 5.13)!

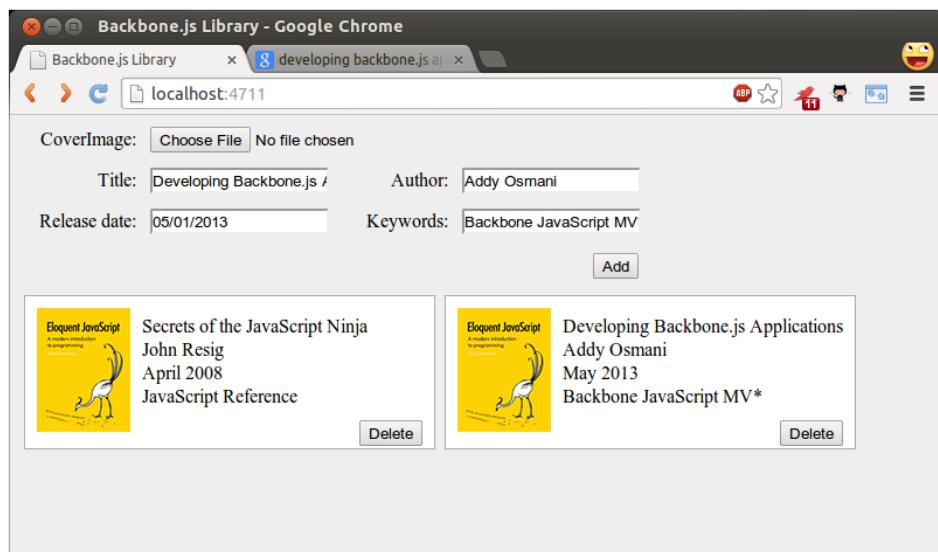


Рис. 5.13. Добавление записей о новых книгах, отображающих одновременно дату издания и ключевые слова

Выводы

В этой главе мы добавили в наше приложение возможности сохранения данных, связав его с сервером при помощи REST API. Мы также изучили некоторые проблемы, которые могут возникать при сериализации и десериализации данных, и познакомились с решением этих проблем. Мы рассмотрели плагины `dateFormat` и `datepicker` библиотеки jQuery и научились выполнять ряд нетривиальных действий с помощью шаблонов Underscore. Приведенный код доступен на моей странице в GitHub.

6

Расширения Backbone

Backbone — гибкая, простая и мощная библиотека. Тем не менее может оказаться, что приложение, которое вы разрабатываете, требует больших возможностей, чем Backbone предлагает в готовом виде. Поскольку Backbone задумана как минималистичная библиотека, существуют задачи, которые она не способна решить напрямую.

Для примера рассмотрим представления, в которых метод `render` по умолчанию не делает ничего и его вызов не приводит ни к каким результатам. Тем не менее в большинстве приложений этот метод генерирует HTML-код, которым управляет представление. Модели и коллекции не обладают встроенными возможностями обработки вложенных иерархий, и если вам нужен такой функционал, то придется написать его самостоятельно или воспользоваться плагином.

Для подобных случаев существуют многочисленные Backbone-плагины, представляющие развитые решения крупномасштабным приложениям на основе Backbone. Вики-страница библиотеки Backbone содержит неплохой список доступных плагинов и фреймворков для нее. Возможности этих плагинов достаточны для разработки приложений почти любого размера.

В этом разделе книги мы займемся изучением двух популярных Backbone-плагинов: *MarionetteJS* и *Thorax*.

MarionetteJS (Backbone.Marionette)

Авторы: Дерик Бейли (Derick Bailey) и Эдди Османни (Addy Osmani)

Как мы уже видели, библиотека Backbone предоставляет замечательный набор компонентов для разработки JavaScript-приложений. Она обеспечивает ключевые конструкции для создания небольших и средних приложений, обработки DOM-событий библиотеки jQuery и разработки одностраничных приложений, удовлетворяющих потребности как мобильных устройств, так и крупных

организаций. Тем не менее библиотека Backbone не является полнофункциональным фреймворком. Она лишь содержит набор компонентов и оставляет на усмотрение разработчика существенную часть вопросов, касающихся архитектуры и масштабирования приложений, в том числе управления памятью, представлениями и пр.

Библиотека MarionetteJS, также известная как Backbone.Marionette, предоставляет широкие возможности для создания сложных приложений, которые отсутствуют в Backbone. MarionetteJS – это составная прикладная библиотека, предназначенная для упрощения разработки крупномасштабных приложений. MarionetteJS предоставляет набор общих паттернов проектирования и реализации, которые использовались Дериком Бейли и другими авторами для создания Backbone-приложений.

Библиотека Marionette имеет следующие основные преимущества:

- позволяет масштабировать приложения с помощью модульной архитектуры на основе событий;
- имеет рациональные настройки по умолчанию, например использование Underscore-шаблонов для отображения представлений;
- может быть с легкостью модифицирована под специфические требования вашего приложения;
- сокращает объем стандартного кода при работе с представлениями благодаря специальным типам представлений;
- имеет модульную архитектуру, состоящую из приложения и подключаемых к нему модулей;
- позволяет формировать графику приложения в процессе его работы с помощью регионов и макетов;
- поддерживает вложенные представления и макеты внутри графических регионов;
- предусматривает управление встроенной памятью и уничтожение «зомбиированных» представлений, регионов и макетов;
- обеспечивает встроенную очистку событий с помощью класса *EventBinder*;
- объединяет событийную архитектуру с агрегатором событий;
- предоставляет гибкую, адаптируемую архитектуру, благодаря которой разработчик может выбирать необходимые компоненты.

Библиотека Marionette похожа на Backbone тем, что обеспечивает набор компонентов, которые могут использоваться как независимо друг от друга, так и совместно, что дает разработчикам существенные преимущества. Marionette

делает «шаг вперед» по сравнению со структурными компонентами библиотеки Backbone и обеспечивает прикладной уровень, состоящий из более чем дюжины компонентов и блоков.

Компоненты Marionette значительно отличаются друг от друга по предоставляемым функциям, однако вместе они образуют комплексную прикладную систему, способную не только сократить размер вспомогательного кода, но и придать приложению требуемую структуру.

Главные плюсы Marionette:

- различные специализированные типы представлений, которые устраниют необходимость использования вспомогательного кода при типичных отображениях классов Backbone.Model и Backbone.Collection;
- объект Application и архитектура Module, позволяющие разделять приложения на подприложения;
- дополнительные возможности и файлы;
- интеграция паттерна «команда», агрегатора событий и механизма запросов/ответов;
- а также многие другие типы объектов, которые можно расширять неограниченным числом способов для создания архитектуры под специфические требования вашего приложения.

Несмотря на то что библиотека Marionette включает большое количество компонентов, вы не обязаны использовать весь набор только потому, что вам нужна его часть. Как и в Backbone, вы сами выбираете, когда и какие возможности задействовать. Это позволяет очень легко работать с другими Backbone-фреймворками и плагинами, а это, в свою очередь, значит, что вам не потребуется отказываться от всех своих прежних наработок для того, чтобы начать пользоваться Marionette.

Вспомогательный код отображения

Давайте разберемся с кодом, необходимым для отображения представления с помощью библиотеки Backbone и Underscore-шаблона. Нам нужен шаблон, помещаемый непосредственно в DOM, и сценарий JavaScript, который определяет представление, использующее шаблон, и заполняет его данными модели.

```
<script type="text/html" id="my-view-template">
  <div class="row">
    <label>First Name:</label>
    <span><%= firstName %></span>
```

продолжение ↗

```

</div>
<div class="row">
  <label>Last Name:</label>
  <span><%= lastName %></span>
</div>
<div class="row">
  <label>Email:</label>
  <span><%= email %></span>
</div>
</script>
var MyView = Backbone.View.extend({
  template: $('#my-view-template').html(),
  render: function(){
    // compile the Underscore.js template
    var compiledTemplate = _.template(this.template);
    // render the template with the model data
    var data = this.model.toJSON();
    var html = compiledTemplate(data);
    // populate the view with the rendered html
    this.$el.html(html);
  }
});

```

Теперь создадим экземпляр представления и передадим в него нашу модель. После этого элемент `el` представления добавим в DOM, чтобы отобразить представление.

```

var Derick = new Person({
  firstName: 'Derick',
  lastName: 'Bailey',
  email: 'derickbailey@example.com'
});
var myView = new MyView({
  model: Derick
})
myView.render();
$('#content').html(myView.el)

```

Это стандартная подготовка к определению, формированию, отображению и выводу представления в библиотеке Backbone. Такой код мы также называем *болванкой* — он содержит одну и ту же функциональность и повторяется в каждом проекте и приложении. Этот код быстро становится рутинным и однообразным.

Воспользуйтесь классом `ItemView` библиотеки Marionette — он позволяет легко сократить «болванку» определения представления.

Сокращение вспомогательного кода с помощью класса Marionette.ItemView

В библиотеке Marionette все типы представлений, кроме Marionette.View, содержат встроенный метод render, в котором находится основная логика отображения. Мы можем воспользоваться этим и изменить экземпляр класса MyView так, чтобы он был наследником не класса Backbone.View, а класса другого представления. Вместо того чтобы создавать для представления собственный метод render, поручим отображение представления библиотеке Marionette. Для этого воспользуемся тем же Underscore-шаблоном и механизмом отображения, но его реализация останется «за кулисами». Так мы сможем сократить количество кода, который потребуется написать для данного представления.

```
var MyView = Marionette.ItemView.extend({  
    template: '#my-view-template'  
});
```

Это все, что нужно для получения представления, которое ведет себя точно так же, как в предыдущем примере. Просто замените Backbone.View.extend на Marionette.ItemView.extend, а затем избавьтесь от метода render. Вы по-прежнему можете создать экземпляр представления с помощью модели, вызвать его метод render и отобразить представление в DOM так, как мы делали ранее. Тем не менее определение представления сократилось до одной строки конфигурации шаблона.

Управление памятью

Библиотека Marionette не только позволяет сократить код определения представления, но и обеспечивает во всех типах своих представлений эффективные возможности управления памятью, упрощающие уничтожение экземпляра представления и его обработчиков событий.

Рассмотрим следующую реализацию представления:

```
var ZombieView = Backbone.View.extend({  
    template: '#my-view-template',  
    initialize: function(){  
        // привязка изменения модели к повторному отображению этого представления  
        this.model.on('change', this.render, this);  
    },  
    render: function(){  
        // Это оповещение будет сигнализировать о проблеме  
        alert('We`re rendering the view');  
    }  
});
```

Если мы создадим два экземпляра этого представления с одним и тем же именем переменной, а затем изменим значение в модели, то сколько раз мы увидим окно оповещения?

```
var Person = Backbone.Model.extend({
  defaults: {
    "firstName": "Jeremy",
    "lastName": "Ashkenas",
    "email": "jeremy@example.com"
  }
});
var Derick = new Person({
  firstName: 'Derick',
  lastName: 'Bailey',
  email: 'derick@example.com'
});
// создание первого экземпляра представления
var zombieView = new ZombieView({
  model: Derick
});
// создание второго экземпляра представления,
// использование переменной с тем же именем для его хранения
zombieView = new ZombieView({
  model: Derick
});
Derick.set('email', 'derickbailey@example.com');
```

Поскольку мы используем переменную `zombieView` для обоих экземпляров представления, первый экземпляр окажется вне области видимости сразу после создания второго экземпляра. Это даст возможность «сборщику мусора» JavaScript уничтожить первый экземпляр, в результате чего представление утратит активность и перестанет реагировать на события изменения модели.

Тем не менее если мы запустим этот код, то увидим окно оповещения дважды!

Эта проблема вызвана привязкой события модели в методе `initialize` представления. Каждый раз, когда мы передаем `this.render` в качестве функции обратного вызова методу `on` модели, сама модель получает прямую ссылку на экземпляр представления. Наличие этой ссылки в модели приводит к тому, что при присваивании нового экземпляра представления переменной `zombieView` первое представление остается в области видимости.

Поскольку второе представление появляется в области видимости в тот момент, когда первое представление продолжает находиться в ней, оба представления реагируют на изменение данных модели.

Решить эту проблему просто: вызовите функцию `stopListening`, когда представление уже закончило работу и готово к закрытию. Для этого добавьте в представление метод `close`.

```

var ZombieView = Backbone.View.extend({
  template: '#my-view-template',
  initialize: function(){
    // привязка изменения модели к повторному отображению этого представления
    this.listenTo(this.model, 'change', this.render);
  },
  close: function(){
    // отключение событий, прослушиваемых этим представлением
    this.stopListening();
  },
  render: function(){
    // Это оповещение будет сигнализировать о проблеме
    alert('We`re rendering the view');
  }
});

```

Затем вызовите метод `close` первого экземпляра, когда он перестанет быть нужным, и «в живых» останется лишь одно представление. Более подробную информацию о функциях `listenTo` и `stopListening` можно получить в главе 3 и статье Дерика «Managing Events As Relationships, Not Just References» («Управление событиями как отношениями, а не только как ссылками»).

```

var Jeremy = new Person({
  firstName: 'Jeremy',
  lastName: 'Ashkenas',
  email: 'jeremy@example.com'
});
// создание первого экземпляра представления
var zombieView = new ZombieView({
  model: Person
})
zombieView.close(); // двойной выстрел в зомби
// создание второго экземпляра представления,
// использование переменной с тем же именем для его хранения
zombieView = new ZombieView({
  model: Person
})
Person.set('email', 'jeremyashkenas@example.com');

```

Теперь при запуске этого кода мы видим только одно окно оповещения.

Вместо того чтобы удалять обработчики событий вручную, мы можем поручить эту задачу библиотеке Marionette.

```

var ZombieView = Marionette.ItemView.extend({
  template: '#my-view-template',
  initialize: function(){
    // привязка изменения модели к повторному отображению этого представления
    this.listenTo(this.model, 'change', this.render);
  },

```

продолжение ↗

```
render: function(){
    // Это оповещение будет сигнализировать о проблеме
    alert('We`re rendering the view');
}
});
```

Обратите внимание на то, что в этом случае мы используем метод с именем `listenTo`. Этот метод принадлежит классу `Backbone.Events` и доступен во всех объектах, которые примешивают к себе этот класс, в том числе в большинстве объектов Marionette. Прототип метода `listenTo` схож с прототипом метода `on`, за исключением того, что первым его параметром является объект, генерирующий событие.

Представления Marionette также поддерживают метод `close`, в котором автоматически удаляются привязки событий, созданные методом `listenTo`. Это означает, что больше не нужно явно определять метод `close`, и при использовании метода `listenTo` мы можем быть уверены, что события будут удалены, а представления не превратятся в «зомби».

Но как нам обеспечить автоматический вызов метода `close` представления в реальном приложении? Когда и где он должен вызываться? Воспользуйтесь объектом `Marionette.Region`, который управляет жизненным циклом отдельного представления.

Управление регионами

После того как представление создано, его необходимо поместить в DOM, чтобы оно стало видимым. Как правило, это делается с помощью селектора jQuery и вызова метода `html()` результирующего объекта:

```
var Joe = new Person({
  firstName: 'Joe',
  lastName: 'Bob',
  email: 'joebob@example.com'
});
var myView = new MyView({
  model: Joe
})
myView.render();
// отобразить представление в DOM
$('#content').html(myView.el)
```

Этот код тоже является «болванкой». Не следует вручную вызывать метод `render` и выбирать DOM-элементы для отображения представления. Этот код также не закрывает существующие экземпляры представления, которые могут

быть подключены к заполняемому DOM-элементу. Кроме того, мы знаем и об опасности «зомбированных» представлений.

Для решения этих проблем библиотека Marionette предлагает объект `Region`, управляющий жизненным циклом отдельных представлений в определенном DOM-элементе.

```
// создание экземпляра региона и указание DOM-элемента,
// которым необходимо управлять
var myRegion = new Marionette.Region({
  el: '#content'
});
// отображение представления в регионе
var view1 = new MyView({ /* ... */ });
myRegion.show(view1);
// где-нибудь в другой части кода
// отобразите другое представление
var view2 = new MyView({ /* ... */ });
myRegion.show(view2);
```

Здесь требуется несколько пояснений. Во-первых, мы указываем региону DOM-элемент, которым он будет управлять, задавая значение `el` в экземпляре региона. Во-вторых, мы больше не вызываем метод `render` наших представлений. И наконец, мы не вызываем и метод `close` представления, — он и вызывается без нашего участия.

Когда мы используем регион для управления жизненным циклом наших представлений и отображаем их в DOM, регион сам выполняет эти задачи. Когда мы передаем экземпляр представления методу `show` региона, регион вызывает метод `render` представления за нас. Затем он берет результирующий элемент `el` представления и заполняет DOM-элемент.

При следующем вызове метода `show` регион «вспоминает», что в текущий момент он отображает представление. Регион вызывает метод `close` этого представления, удаляет его из DOM и затем запускает код, который отображает и выводит на экран новое представление, которое было передано ему.

Поскольку регион работает за нас с методом `close` и мы используем метод привязки событий `listenTo` в экземпляре нашего представления, нам больше не нужно беспокоиться о «зомбированных» представлениях в нашем приложении.

Отметим, что регионы работают не только с Marionette-представлениями. Любой корректный экземпляр класса `Backbone.View` может управляться классом `Marionette.Region`. Если у вашего представления есть метод `close`, то он будет вызываться при закрытии представления. Если этот метод отсутствует, то вместо него будет вызван встроенный метод `remove` класса `Backbone.View`.

Приложение для управления задачами на основе Marionette

Теперь, когда мы знакомы с общими концепциями библиотеки Marionette, давайте реорганизуем приложение для управления задачами, которое мы разработали в первом упражнении. Полный код этого приложения можно найти в ветке TodoMVC, принадлежащей Дерику.

Окончательный результат нашей разработки будет визуально и функционально повторять исходное приложение, как показано на рис. 6.1.

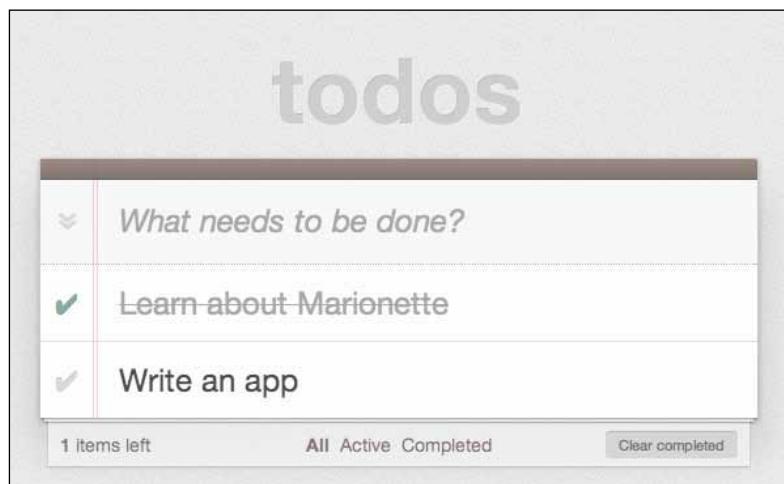


Рис. 6.1. Приложение для управления задачами на основе Marionette, которое мы разрабатаем

Первое, что мы сделаем, — определим объект нашего базового приложения TodoMVC. Этот объект будет содержать в себе код инициализации приложения и определять для него регионы макета по умолчанию.

TodoMVC.js

```
var TodoMVC = new Marionette.Application();
TodoMVC.addRegions({
    header : '#header',
    main : '#main',
    footer : '#footer'
});
TodoMVC.on('initialize:after', function(){
    Backbone.history.start();
});
```

Регионы используются для управления содержимым, которое отображается внутри определенных элементов, а метод `addRegions` объекта `TodoMVC` представляет собой всего лишь «быструю ссылку» для создания объектов регионов. Мы создаем `jQuery`-селектор для каждого управляемого региона (например, `#header`, `#main` и `#footer`), а затем даем региону команды отображать различные Backbone-представления внутри себя.

После завершения инициализации объекта приложения вызываем метод `Backbone.history.start()` для маршрутизации начального URL.

Затем определяем *макеты*. Макет — это особый тип представления, который прямо расширяет класс `Marionette.ItemView`. Это означает, что он предназначен для отображения одного шаблона и может как иметь, так и не иметь модели (или элемент `item`), связанной с этим шаблоном.

Одно из главных различий между макетом и представлением `ItemView` заключается в том, что макет содержит в себе регионы. При определении макета указывается шаблон и регионы, которые он включает в себя. После отображения макета отобразим внутри него другие представления с помощью регионов, которые мы определили.

В модуле макетов нашего приложения мы определяем следующие макеты:

- **Header** — здесь мы можем создавать новые задачи.
- **Footer** — здесь мы подсчитываем, сколько задач завершено и сколько находится в процессе выполнения.

В этих компонентах мы размещаем часть логики представлений, которая раньше содержалась в объектах `AppView` и `TodoView`.

Обратите внимание, что модули библиотеки `Marionette` (такие, как приведены ниже) образуют простую систему, обеспечивающую закрытость и инкапсуляцию `Marionette`-приложений. Разумеется, использовать эти модули не обязательно, и в разделе «`Marionette` и гибкость» мы приведем ссылки на альтернативные реализации, в которых используется библиотека `RequireJS` и интерфейс AMD (asynchronous module definition, определение асинхронных модулей).

TodoMVC.Layout.js

```
TodoMVC.module('Layout', function(Layout, App, Backbone, Marionette, $, _){  
    // Представление заголовка макета  
    // -----  
    Layout.Header = Marionette.ItemView.extend({  
        template : '#template-header',  
        // Привязки пользовательского интерфейса создают кэшированные  
        // атрибуты, которые указывают на объекты, выбранные jQuery  
        ui : {  
            input : '#new-todo'  
        },  
    });  
});
```

продолжение ↗

```

events : {
    'keypress #new-todo': 'onInputKeypress'
},
onInputKeypress : function(evt) {
    var ENTER_KEY = 13;
    var todoText = this.ui.input.val().trim();
    if ( evt.which === ENTER_KEY && todoText ) {
        this.collection.create({
            title : todoText
        });
        this.ui.input.val('');
    }
}
});
// Представление нижнего колонтитула макета
// -----
Layout.Footer = Marionette.Layout.extend({
    template : '#template-footer',
    // Привязки пользовательского интерфейса создают кэшированные
    // атрибуты, которые указывают на объекты, выбранные jQuery
    ui : {
        count : '#todo-count strong',
        filters : '#filters a'
    },
    events : {
        'click #clear-completed' : 'onClearClick'
    },
    initialize : function() {
        this.listenTo(App.vent, 'todoList:filter', this.updateFilterSelection);
        this.listenTo(this.collection, 'all', this.updateCount);
    },
    onRender : function() {
        this.updateCount();
    },
    updateCount : function() {
        var count = this.collection.getActive().length;
        this.ui.count.html(count);
        if (count === 0) {
            this.$el.parent().hide();
        } else {
            this.$el.parent().show();
        }
    },
    updateFilterSelection : function(filter) {
        this.ui.filters
            .removeClass('selected')
            .filter('[href="#' + filter + '"]')
            .addClass('selected');
    },
    onClearClick : function() {
        var completed = this.collection.getCompleted();
        completed.forEach(function destroy(todo) {
            todo.destroy();
        });
    }
});
});
});
```

Далее мы реализуем в приложении такие функции, как маршрутизация и управление макетами, которые можно как отображать на странице, так и скрывать.

Вспомните, как библиотека Backbone маршрутизирует методы генерации событий: ниже приведен код маршрутизатора из исходного рабочего пространства нашего первого упражнения:

```
var Workspace = Backbone.Router.extend({
  routes: {
    '*filter': 'setFilter'
  },
  setFilter: function( param ) {
    // задание текущего фильтра
    if (param){ param = param.trim() }
    app.TodoFilter = param || "";
    // генерация события filter коллекции, вызывающего
    // скрытие/отображение задач
    app.Todos.trigger('filter');
  }
});
```

Для упрощения маршрутизации библиотека Marionette использует объект AppRouter. Он сокращает «болванку» для обработки событий маршрутизации и позволяет конфигурировать маршрутизаторы так, чтобы они напрямую вызывали методы объекта. Мы конфигурируем маршрутизатор AppRouter с помощью элемента appRoutes, который заменяет маршрут '*filter': 'setFilter', определенный в исходном маршрутизаторе, и вызывает метод нашего контроллера.

Контроллер списка задач, который также находится в следующем блоке кода, реализует часть оставшейся логики визуализации, изначально входившей в состав представлений AppView и TodoView, с помощью очень удобных для восприятия макетов.

TodoMVC.TodoList.js

```
TodoMVC.module('TodoList', function(TodoList, App, Backbone, Marionette, $, _){
  // Маршрутизатор списка задач
  // -----
  //
  // обрабатывает маршруты для отображения активных и завершенных задач
  TodoList.Router = Marionette.AppRouter.extend({
    appRoutes : {
      '*filter': 'filterItems'
    }
  });
  // Контроллер списка задач ("посредник")
  // -----
  //
  // управляет операциями и логикой на уровне приложения,
```

продолжение ↗

```
// находящегося над реализацией представлений и моделей
TodoList.Controller = function(){
    this.todoList = new App.Todos.TodoList();
};

_.extend(TodoList.Controller.prototype, {
    // запуск приложения: отображение соответствующих представлений
    // и считывание списка задач, если они есть
    start: function(){
        this.showHeader(this.todoList);
        this.showFooter(this.todoList);
        this.showTodoList(this.todoList);
        this.todoList.fetch();
    },
    showHeader: function(todoList){
        var header = new App.Layout.Header({
            collection: todoList
        });
        App.header.show(header);
    },
    showFooter: function(todoList){
        var footer = new App.Layout.Footer({
            collection: todoList
        });
        App.footer.show(footer);
    },
    showTodoList: function(todoList){
        App.main.show(new TodoList.Views.ListView({
            collection : todoList
        }));
    },
    // установка фильтра, отображающего только завершенные
    // задачи либо все задачи
    filterItems: function(filter){
        App.vent.trigger('todoList:filter', filter.trim() || '');
    }
});
// Инициализатор списка задач
// -----
// Подготовка списка задач путем инициализации "посредника"
// при запуске приложения, считывание и отображение
// всех существующих задач.
TodoList.addInitializer(function(){
    var controller = new TodoList.Controller();
    new TodoList.Router({
        controller: controller
    });
    controller.start();
});
});
```

Контроллеры

Обратите внимание, что в этом примере контроллеры не вносят существенного вклада в общий функционал приложения. С точки зрения Marionette маршрутизаторы используются в конце работы над приложением. Мы часто видели, что разработчики неправильно используют систему маршрутизации Backbone, полностью возлагая на нее управление операциями и логикой приложения.

Это неизбежно приводит к тому, что весь код приложения, а именно создание представлений, загрузка моделей, координация между различными частями приложения и т. д., оказывается связанным с методами маршрутизатора. Разработчики вроде Дерика считают, что такой подход нарушает принципы персональной ответственности и разделения задач.

Поддержка маршрутизаторов и журнала в Backbone связана с управлением кнопками браузера для перелистывания страниц вперед и назад. Подход библиотеки Marionette состоит в том, что такая поддержка должна ограничиваться именно этим, а код, который исполняется при навигации, должен находиться в другом месте. Это позволяет использовать приложение как с маршрутизатором, так и без него. Мы можем вызвать метод отображения контроллера щелчком по кнопке из обработчика событий приложения или из маршрутизатора: наше приложение окажется в одном и том же состоянии независимо от способа, которым мы вызвали этот метод.

Дерик подробно высказал свои размышления по данной теме в статьях, которые опубликованы в его блоге Lostechies:

- The Responsibilities Of The Various Pieces Of Backbone.js («Функции различных компонентов библиотеки Backbone.js»);
- Reducing Backbone Routers To Nothing More Than Configuration («Как сократить функционал маршрутизаторов Backbone, оставив только конфигурирование»);
- 3 Stages Of A Backbone Application’s Startup («Три стадии запуска Backbone-приложения»).

CompositeView

Наше следующее действие — определить в приложении представления для отдельных задач и их списков. Для этого воспользуемся классом **CompositeView**, который визуализирует составную или иерархическую структуру листьев (или узлов) и ветвей.

Данные представления можно считать иерархией родительско-дочерних моделей, которая по умолчанию рекурсивна. Тот же тип `CompositeView` будет использоваться для отображения каждого элемента в коллекции, которая обрабатывается составным представлением. Для нерекурсивных иерархий мы можем переопределить представление элемента, определив атрибут `itemView`.

Для элемента нашего списка задач определяем этот атрибут как `itemView`; представление списка задач — это объект `CompositeView`, в котором мы переопределяем параметр `itemView` и указываем ему использовать представление элемента списка задач для каждого элемента коллекции.

TodoMVC.TodoList.Views.js

```
TodoMVC.module('TodoList.Views', function
  (Views, App, Backbone, Marionette, $, _){
  // Представление элемента списка задач
  // -----
  //
  // Отображение отдельной задачи и реагирование на ее изменения,
  // в том числе установку флага завершения.
  Views.ItemView = Marionette.ItemView.extend({
    tagName : 'li',
    template : '#template-todoItemView',
    ui : {
      edit : '.edit'
    },
    events : {
      'click .destroy' : 'destroy',
      'dblclick label' : 'onEditClick',
      'keypress .edit' : 'onEditKeypress',
      'click .toggle' : 'toggle'
    },
    initialize : function() {
      this.listenTo(this.model, 'change', this.render);
    },
    onRender : function() {
      this.$el.removeClass('active completed');
      if (this.model.get('completed')) this.$el.addClass('completed');
      else this.$el.addClass('active');
    },
    destroy : function() {
      this.model.destroy();
    },
    toggle : function() {
      this.model.toggle().save();
    },
    onEditClick : function() {
      this.$el.addClass('editing');
      this.ui.edit.focus();
    },
  });
});
```

```

onEditKeypress : function(evt) {
    var ENTER_KEY = 13;
    var todoText = this.ui.edit.val().trim();
    if ( evt.which === ENTER_KEY && todoText ) {
        this.model.set('title', todoText).save();
        this.$el.removeClass('editing');
    }
}
});

// Представление элемента списка
// -----
//
// Управляет отображением списка элементов, в том числе
// фильтрацией завершенных и незавершенных задач.
Views.ListView = Marionette.CompositeView.extend({
template : '#template-todoListCompositeView',
itemView : Views.ItemView,
itemViewContainer : '#todo-list',
ui : {
    toggle : '#toggle-all'
},
events : {
    'click #toggle-all' : 'onToggleAllClick'
},
initialize : function() {
    this.listenTo(this.collection, 'all', this.update);
},
onRender : function() {
    this.update();
},
update : function() {
    function reduceCompleted(left, right)
    { return left && right.get('completed'); }
    var allCompleted = this.collection.reduce(reduceCompleted,true);
    this.ui.toggle.prop('checked', allCompleted);
    if (this.collection.length === 0) {
        this.$el.parent().hide();
    } else {
        this.$el.parent().show();
    }
},
onToggleAllClick : function(evt) {
    var isChecked = evt.currentTarget.checked;
    this.collection.each(function(todo){
        todo.save({'completed': isChecked});
    });
}
});
// Обработчики событий приложения
// -----
//
// Обработчик фильтрации списка элементов, скрывающий и отображающий
// элементы с помощью различных CSS-классов.
App.vent.on('todoList:filter',function(filter) {

```

продолжение ↗

```

        filter = filter || 'all';
        $('#todoapp').attr('class', 'filter-' + filter);
    });
});

```

В конце последнего блока кода вы обнаружите обработчик события, использующего элемент `vent`. Это агрегатор событий, который позволяет реагировать на срабатывания функции `filterItem` контроллера списка задач.

Наконец, мы определяем модель и коллекцию, соответствующие нашим задачам. Семантически они мало отличаются от своих исходных версий из первого упражнения; изменения, которые внесены в них, сделаны для того, чтобы они лучше соответствовали стилю кодирования Дерика.

Todos.js

```

TodoMVC.module('Todos', function(Todos, App, Backbone, Marionette, $, _){
    // Модель задачи
    // -----
    Todos.Todo = Backbone.Model.extend({
        localStorage: new Backbone.LocalStorage('todos-backbone'),
        defaults: {
            title: '',
            completed: false,
            created: 0
        },
        initialize: function() {
            if (this.isNew()) this.set('created', Date.now());
        },
        toggle: function() {
            return this.set('completed', !this.isCompleted());
        },
        isCompleted: function() {
            return this.get('completed');
        }
    });
    // Коллекция задач
    // -----
    Todos.TodoList = Backbone.Collection.extend({
        model: Todos.Todo,
        localStorage: new Backbone.LocalStorage('todos-backbone'),
        getCompleted: function() {
            return this.filter(this._isCompleted);
        },
        getActive: function() {
            return this.reject(this._isCompleted);
        },
        comparator: function( todo ) {
            return todo.get('created');
        },
        _isCompleted: function(todo){

```

```
        return todo.isCompleted();
    }
});  
});
```

Запустим приложение в индексном файле, вызвав функцию `start` главного объекта приложения. Инициализация выглядит следующим образом:

```
$(function(){
    // запуск приложения TodoMVC (определенного в файле js/TodoMVC.js)
    TodoMVC.start();
});
```

Вот и всё!

Удобнее ли поддерживать приложения на Marionette?

Дерик считает, что удобство поддержки приложения в значительной степени обеспечивается его модульностью и разделением ответственности (принципом личной ответственности и распределения задач), которое достигается применением паттернов, предотвращающих смешение функций. Тем не менее бывает сложно структурировать приложение в виде модулей в соответствии с канонами выделения, абстрагирования и дробления концепции на простейшие элементы.

Принцип личной ответственности говорит об обратном: мы должны понять контекст, в котором происходят изменения. Какие части *этой* системы всегда меняются совместно? Какие части меняются независимо друг от друга? Не зная этого, мы не поймем, какие фрагменты приложения необходимо разбить на отдельные компоненты и модули, а какие поместить в один модуль или объект.

Дерик формирует модульную структуру приложения, разделяя концепции на каждом уровне. Модуль верхнего уровня решает самые глобальные задачи и объединяет ответственности. Каждая зона ответственности делится на выражительный набор API, которые реализуются модулями более низкого уровня (этот подход известен как принцип инверсии зависимостей). Эти модули координируются посредником, в роли которого обычно выступает контроллер модуля.

Способ организации файлов также напрямую влияет на удобство поддержки приложения. Дерик написал статьи о том, как важно поддерживать осмысленную структуру папок в приложении, с которыми я рекомендую ознакомиться:

- JavaScript File & Folder Structures: Just Pick One («Структуры файлов и папок в JavaScript: сделайте свой выбор»);

- How to organize and structure the files and folders of HiloJS («Как организовать и структурировать файлы и папки в HiloJS»).

Marionette и гибкость

Marionette, как и Backbone, является гибким фреймворком. Он обеспечивает широкий спектр инструментов для создания и организации архитектуры приложения, основанного на Backbone, и не требует использования всех своих компонентов только потому, что вам необходим один из них. Вы легко убедитесь в гибкости и адаптивности библиотеки Marionette, если изучите и сравните три версии приложения TodoMVC, реализованные с ее помощью.

Простая версия, автор — Джеррод Оверсон (Jarrod Overson)

Эта версия TodoMVC демонстрирует непосредственное использование различных типов представлений библиотеки Marionette, объекта приложения и агрегатора событий. Создаваемые объекты очень просты и напрямую добавляются в глобальное пространство имен. Это прекрасный пример расширения существующего кода без переписывания приложения с нуля на Marionette.

RequireJS-версия, автор тот же

Использование Marionette совместно с библиотекой RequireJS упрощает создание приложений с модульной структурой, что крайне важно для масштабирования JavaScript-приложений. Библиотека RequireJS предоставляет набор очень полезных инструментов, делающих Marionette еще более гибкой.

Версия с Marionette-модулями, автор — Дерик Бейли

Библиотека RequireJS — не единственный способ создания приложений с модульной архитектурой. Те, кто хочет разрабатывать приложения с использованием модулей и пространств имен, могут воспользоваться библиотекой Marionette, которая содержит встроенную структуру для работы с ними. Данное приложение представляет собой «простую» версию, переписанную с использованием пространств имен и контроллера приложения (объекта посредника/операции), объединяющего отдельные части приложения в целое.

Безусловно, библиотека Marionette «рекомендует», как структурировать Backbone-приложение. Сочетание модулей, представлений различных типов, агрегатора событий, объектов приложений и других компонентов позволяет создать мощную и гибкую архитектуру на основе этих «рекомендаций».

Тем не менее библиотека Marionette не является жестким и бескомпромиссным фреймворком. Она предоставляет большое количество базовых элементов для создания приложений, которые можно использовать совместно друг с другом и применять в приложениях с альтернативными подходами к архитектуре (например, с AMD и пространствами имен). Этими элементами можно пользоваться и в существующих проектах для сокращения подготовительного кода при отображении представлений.

Такая гибкость делает Marionette ценным инструментом для проектов, поскольку позволяет задействовать именно те ее возможности, которые требуются вашему приложению.

Дополнительная информация

Рассмотренные нами возможности библиотеки Marionette, даже объекты `ItemView` и `Region`, являются лишь «верхушкой айсберга». Эти объекты содержат в себе гораздо больше функций, возможностей и гибких вариантов настройки. Библиотека Marionette содержит еще около дюжины компонентов, каждый из которых имеет свой набор встроенных возможностей, настроек, точек расширения и других составляющих.

Сайт <http://marionettejs.com> содержит дополнительные сведения о компонентах и возможностях библиотеки Marionette, примеры ее использования, документацию, ссылки на вики-статьи, исходные коды, материалы основных авторов проекта и другую информацию.

Thorax

Авторы — Райан Истридж (*Ryan Eastridge*) и Эдди Османи (*Addy Osmani*)

Сильная сторона библиотеки Backbone в том, что она придает приложению структуру, однако сама библиотека довольно абстрактна, в том числе и в подходе к представлениям. Библиотека Thorax предлагает использовать библиотеку Handlebars как решение для работы с шаблонами. В Thorax также можно найти некоторые паттерны библиотеки Marionette. В Marionette большинство этих паттернов доступно через JavaScript API, а в Thorax они в основном представлены в виде помощников шаблонов. Для изучения этого раздела необходимо, чтобы читатель был знаком с библиотекой Handlebars.

Райан Истридж и Кевин Деккер (*Kevin Decker*) разработали библиотеку Thorax для мобильного веб-приложения компании Walmart. В этой главе речь идет лишь о возможностях Thorax, касающихся работы с шаблонами, и ее паттернами, которые можно использовать в приложениях независимо от того, используется ли в них сама библиотека. Узнать о других возможностях Thorax и загрузить типовые проекты на ее основе можно на веб-сайте, посвященном Thorax.

Простейшее приложение

В библиотеке Backbone при создании нового представления переданные ему параметры объединяются с существующими параметрами по умолчанию и доступны для дальнейшего использования в виде элемента `this.options`.

Класс Thorax.View отличается от Backbone.View тем, что в нем нет объекта `options`. Все аргументы, передаваемые конструктору, становятся свойствами представления, доступными шаблону:

```
var view = new Thorax.View({
  greeting: 'Hello',
  template: Handlebars.compile('{{greeting}} World!')
});
view.appendTo('body');
```

В большинстве примеров этой главы будет использоваться свойство `template`. В крупных проектах, в том числе в примерах, опубликованных на веб-сайте Thorax, вместо этого свойства используется свойство `name`, а шаблон, находящийся в файле с таким же именем в вашем проекте, будет автоматически привязан к представлению.

Если в представлении задан элемент `model`, то его атрибуты также становятся доступными шаблону:

```
var view = new Thorax.View({
  model: new Thorax.Model({key: 'value'}),
  template: Handlebars.compile('{{key}}')
});
```

Встраивание дочерних представлений

Помощник представления позволяет встраивать одни представления в другие. Дочерние представления могут быть указаны как свойства родительского представления:

```
var parent = new Thorax.View({
  child: new Thorax.View(...),
  template: Handlebars.compile('{{view child}}')
});
```

Можно также передать имя дочернего представления, которое нужно инициализировать, и любые его дополнительные свойства. В этом случае дочернее представление должно быть заранее создано с использованием функции `extend` и свойства `name`:

```
var ChildView = Thorax.View.extend({
  name: 'child',
  template: ...
});
var parent = new Thorax.View({
  template: Handlebars.compile('{{view "child" key="value"}}')
});
```

Помощник представления может также использоваться как помощник блока, в этом случае блок будет присвоен свойству `template` дочернего представления:

```
{{#view child}}
Этот блок будет задан в качестве
свойства template дочернего представления
{{/view}}
```

Библиотека Handlebars работает со строками, в то время как экземпляры класса `Backbone.View` содержат DOM-элемент `el`. Поскольку мы «смешиваем образы», встраивание представлений действует через механизм заполнения, в котором помощник `view` добавляет переданное ему представление в элемент `children`, а затем вставляет HTML-заполнитель в шаблон. Пример:

```
<div data-view-placeholder-cid="view2"></div>
```

После отображения родительского представления мы ищем в DOM все созданные заполнители, заменяя их элементами `el` дочерних представлений:

```
this.$el.find('[data-view-placeholder-cid]').forEach(function(el) {
  var cid = el.getAttribute('data-view-placeholder-cid'),
      view = this.children[cid];
  view.render();
  $(el).replaceWith(view.el);
}, this);
```

Помощники представлений

Один из наиболее полезных компонентов Thorax — `Handlebars.registerViewHelper` (не путайте с `Handlebars.registerHelper`). Этот метод регистрирует новый блочный помощник, который создает и встраивает экземпляр представления `HelperView`, записав занимаемый им блок в его свойство `template`. Экземпляр `HelperView` отличается от обычного дочернего представления тем, что его контекст совпадает с контекстом родительского представления в шаблоне. Как и у других дочерних представлений, его свойство `parent` будет совпадать с одноименным свойством представления, в котором оно объявлено. Подобным образом в Thorax создаются многие встроенные помощники, в том числе помощник коллекций.

В качестве простого примера приведем помощник `on`, он заново отображает сгенерированный экземпляр представления `HelperView` при генерации события в объявившем или породившем его представлении:

```
Handlebars.registerViewHelper('on', function(eventName, helperView) {
    helperView.parent.on(eventName, function() {
        helperView.render();
    });
});
```

Такой помощник можно использовать, например, в ситуации, где требуется увеличивать на единицу счетчик при каждом нажатии кнопки. В данном примере используется Thorax-помощник `button`, который просто создает кнопку, вызывающую метод при щелчке по ней:

```
{{#on "incremented"}}{{i}}{{/on}}
{{#button trigger="incremented"}}Add{{/button}}
```

И соответствующий класс представления:

```
new Thorax.View({
    events: {
        incremented: function() {
            ++this.i;
        }
    },
    initialize: function() {
        this.i = 0;
    },
    template: ...
});
```

Помощник `collection`

Помощник `collection` создает и встраивает экземпляр класса `CollectionView`, формируя представление для каждого элемента коллекции и обновляя эти представления при добавлении, удалении или изменении элементов коллекции. Простейший пример использования этого помощника выглядит так:

```
{{#collection kittens}}
<li>{{name}}</li>
{{/collection}}
```

и соответствующее представление:

```
new Thorax.View({
    kittens: new Thorax.Collection(...),
    template: ...
});
```

В этой ситуации блок будет присвоен элементу `template` каждого создаваемого представления элемента, а контекстом будет элемент `attributes` соответствующей модели. Этот помощник принимает параметры, в роли которых могут выступать произвольные HTML-атрибуты, параметр `tag`, определяющий тип тега, содержащего коллекцию, или любое из следующих значений:

○ `item-template`

Шаблон, который будет отображен для каждой модели. Если указан блок, то он присваивается `item-template`.

○ `item-view`

Класс представления, который будет использоваться при создании представления каждого элемента.

○ `empty-template`

Шаблон, который будет отображен, когда коллекция пуста. Если указан блок `inverse/else`, то он будет присвоен `empty-template`.

○ `empty-view`

Представление, которое будет отображено, когда коллекция пуста.

Параметры и блоки могут использоваться в сочетании друг с другом. В нашем примере создается класс `KittenView` с элементом `template`, в который записывается занятый блок для каждого котенка в коллекции:

```
{{#collection kittens item-view="KittenView" tag="ul"}}
<li>{{name}}</li>
{{else}}
<li>No kittens!</li>
{{/collection}}
```

Обратите внимание на то, что одно представление можно использовать с несколькими коллекциями, а коллекции могут быть вложенными. Это полезно при наличии моделей, содержащих коллекции, которые содержат модели и т. д.:

```
{{#collection kittens}}
<h2>{{name}}</h2>
<p>Kills:</p>
{{#collection miceKilled tag="ul"}}
    <li>{{name}}</li>
{{/collection}}
{{/collection}}
```

Настраиваемые атрибуты HTML-данных

В библиотеке Thorax очень активно используются настраиваемые атрибуты HTML-данных. Одни атрибуты имеют смысл только в контексте Thorax, однако

другие будут весьма полезны в любом Backbone-проекте для написания связанных с ними дополнительных функций, а также для общей отладки. Чтобы добавить такие атрибуты в представления в проектах, не использующих Thorax, переопределите метод `setElement` в базовом классе представления:

```
MyApplication.View = Backbone.View.extend({
    setElement: function() {
        var response = Backbone.View.prototype.setElement.apply(this, arguments);
        this.name && this.$el.attr('data-view-name', this.name);
        this.cid && this.$el.attr('data-view-cid', this.cid);
        this.collection && this.$el.attr('data-collection-cid',
            this.collection.cid);
        this.model && this.$el.attr('data-model-cid', this.model.cid);
        return response;
    }
});
```

Ваше приложение сразу становится понятнее в инспекторе, и, кроме того, теперь можно расширить jQuery/Zepto функциями поиска представления, модели или коллекции, наиболее близкой к заданному элементу. Чтобы этот механизм заработал, вам необходимо сохранить ссылки на каждое представление, созданное в базовом классе представления, путем переопределения метода `_configure`:

```
MyApplication.View = Backbone.View.extend({
    _configure: function() {
        Backbone.View.prototype._configure.apply(this, arguments);
        Thorax._viewsIndexedByCid[this.cid] = this;
    },
    dispose: function() {
        Backbone.View.prototype.dispose.apply(this, arguments);
        delete Thorax._viewsIndexedByCid[this.cid];
    }
});
```

Затем мы можем расширить jQuery/Zepto:

```
$.fn.view = function() {
    var el = $(this).closest('[data-view-cid]');
    return el && Thorax._viewsIndexedByCid[el.attr('data-view-cid')];
};

$.fn.model = function(view) {
    var $this = $(this),
        modelElement = $this.closest('[data-model-cid]'),
        modelCid = modelElement && modelElement.attr('[data-model-cid]');
    if (modelCid) {
        var view = $this.view();
        return view && view.model;
    }
    return false;
};
```

Теперь вы можете использовать метод `$(element).model()` вместо того, чтобы хаотически сохранять ссылки на модели внутри вашего приложения и затем искать их при наступлении определенного DOM-события. В библиотеке Thorax эта возможность особенно полезна в сочетании с помощником `collection`, который генерирует класс представления (со свойством `model`) для каждой модели коллекции. Вот пример шаблона:

```
{{{#collection kittens tag="ul"}}
<li>{{name}}</li>
{{/collection}}
```

И соответствующий класс представления:

```
Thorax.View.extend({
  events: {
    'click li': function(event) {
      var kitten = $(event.target).model();
      console.log('Clicked on ' + kitten.get('name'));
    }
  },
  kittens: new Thorax.Collection(...),
  template: ...
});
```

Распространенный антипаттерн в Backbone-приложениях — задавать атрибут `className` в единственном классе представления. Вместо этого можно воспользоваться атрибутом `data-view-name` в качестве CSS-селектора, сохраняя классы CSS для многократного использования:

```
[data-view-name="child"] {
```

Ресурсы, посвященные Thorax

Любое руководство по Backbone было бы неполным без приложения, управляющего задачами. Существует реализация приложения TodoMVC на основе Thorax, а также еще один, гораздо более простой пример, состоящий из единственного Handlebars-шаблона, который приведен ниже:

```
{{{#collection todos tag="ul"}}
<li{{#if done}} class="done"{{/if}}>
  <input type="checkbox" name="done"{{#if done}} checked="checked"{{/if}}>
  <span>{{item}}</span>
</li>
{{/collection}}
<form>
```

продолжение ↗

```
<input type="text">
<input type="submit" value="Add">
</form>
```

Вот соответствующий JavaScript-код:

```
var todosView = Thorax.View({
  todos: new Thorax.Collection(),
  events: {
    'change input[type="checkbox"]': function(event) {
      var target = $(event.target);
      target.model().set({done: !!target.attr('checked')});
    },
    'submit form': function(event) {
      event.preventDefault();
      var input = this.$('input[type="text"]');
      this.todos.add({item: input.val()});
      input.val('');
    }
  },
  template: '...'
});
todosView.appendTo('body');
```

Чтобы увидеть, как библиотека Thorax работает на крупномасштабном веб-сайте, зайдите на walmart.com с любого Android- или iOS-устройства. Полный список ресурсов о Thorax вы найдете на веб-сайте библиотеки Thorax.

Выводы

Несмотря на то что библиотека Backbone часто используется для создания современных клиентских приложений, в некоторых проектах требуется больше готовых возможностей. Библиотека Thorax обеспечивает работу с Backbone в стиле Rails, предоставляя нужные возможности в готовом виде. Thorax дает ответы на вопросы о том, как должен выглядеть Backbone-проект, какую структуру каталогов следует использовать и как разделить клиентское приложение на готовые модули для всех его целевых платформ. Несмотря на то что библиотека Thorax не является универсальным инструментом, она может оказаться полезной для тех, кто занимается разработкой относительно сложных приложений.

7

Типичные проблемы и пути их решения

В этой главе мы рассмотрим ряд распространенных проблем, с которыми разработчики сталкиваются при использовании Backbone.js в относительно нетривиальных проектах, и приведем возможные решения этих проблем.

Пожалуй, чаще всего у разработчиков возникает вопрос, как максимально эффективно использовать представления. В этой главе вы узнаете о работе с вложенными представлениями, а также об освобождении и наследовании представлений.

Работа с вложенными представлениями

Проблема

Каков оптимальный подход к отображению и добавлению вложенных представлений (или подпредставлений) в библиотеке Backbone.js?

Решение 1

Поскольку страница состоит из вложенных элементов, а представления Backbone соответствуют элементам внутри нее, естественным подходом к управлению иерархией элементов является механизм вложенных представлений.

Наилучший способ объединения представлений опирается на библиотеку jQuery и выглядит так:

```
this.$('.someContainer').append(innerView.el);
```

В реальном примере мы могли бы использовать его следующим образом:

```
...
initialize : function () {
    //...
},
render : function () {
    this.$el.empty();
    this.innerView1 = new Subview({options});
    this.innerView2 = new Subview({options});
    this$('.inner-view-container')
        .append(this.innerView1.el)
        .append(this.innerView2.el);
}
```

Решение 2

Иногда для решения этой проблемы новички пытаются использовать метод `setElement`, однако следует иметь в виду, что этим методом можно легко «выстрелить себе в ногу». Избегайте такого подхода, если можете воспользоваться первым решением:

```
// дочернее подпредставление в существующем
// представлении можно создать так:
...
initialize : function () {
    this.innerView1 = new Subview({options});
    this.innerView2 = new Subview({options});
},
render : function () {
    this.$el.html(this.template());
    this.innerView1.setElement('.some-element1').render();
    this.innerView2.setElement('.some-element2').render();
}
```

Здесь мы создаем подпредставления в методе `initialize()` родительского представления и отображаем их в его методе `render()`. Элементы, которыми управляют подпредставления, находятся в шаблоне родительского представления, а метод `View.setElement()` используется для повторного связывания элемента с каждым подпредставлением.

Метод `setElement()` изменяет элемент представления: в частности, он удаляет обработчиков событий из старого элемента и связывает их с новым элементом. Обратите внимание, что метод `setElement()` возвращает представление, это дает возможность объединить его в «цепочку» с вызовом метода `render()`.

Этот подход успешно работает и имеет несколько достоинств: во-первых, вам не нужно заботиться о порядке DOM-элементов при добавлении подпредставлений, во-вторых, представления инициализируются быстро, и, в-третьих, методу `render()` не требуется решать слишком много задач.

К сожалению, у данного подхода имеются и недостатки, которых не было у первого решения: вы не можете задать свойство `tagName` для подпредставления, а события необходимо повторно делегировать.

Решение 3

Можно решить эту задачу еще одним способом:

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.inner = new InnerView();
    },
    render: function() {
        this.$el.html(template); // или this.$el.empty(), если у вас нет шаблона
        this.$el.append(this.inner.$el);
        this.inner.render();
    }
});
var InnerView = Backbone.View.extend({
    render: function() {
        this.$el.html(template);
        this.delegateEvents();
    }
});
```

У этого решения есть несколько особенностей:

- порядок добавления субэлементов имеет значение;
- представление `OuterView` не содержит HTML-элементов, задаваемых в представлении (или представлениях) `InnerView`, что позволяет указывать в `InnerView` свойство `tagName`;
- метод `render()` вызывается после того, как элемент `InnerView` помещен в DOM. Это полезно, если метод `render()` представления `InnerView` задает свой размер на странице в зависимости от размеров другого элемента. Такая ситуация встречается часто.

Поскольку представление `InnerView` заменяет содержимое своего элемента, оно должно вызывать метод `View.delegateEvents()`, чтобы связать своих обработчиков событий с новым DOM-элементом.

Решение 4

Более удачное и понятное решение (которое, тем не менее, может негативно повлиять на производительность приложения) выглядит следующим образом:

```

var OuterView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },
    render: function() {
        this.$el.html(template);
        // или this.$el.empty(), если у вас нет шаблона
        this.inner = new InnerView();
        this.$el.append(this.inner.$el);
    }
});
var InnerView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },
    render: function() {
        this.$el.html(template);
    }
});

```

Если требуется вложить несколько представлений в определенное место шаблона, воспользуйтесь набором дочерних представлений, индексированным по идентификаторам клиента (`cid`). В шаблоне создайте элементы-заполнители для каждого встраиваемого представления с помощью настраиваемого HTML-атрибута с именем `data-view-cid`. После отображения шаблона и добавления его содержимого в элемент `$el` родительского представления у каждого заполнителя запросите элемент `el` дочернего представления, а также замените заполнитель с помощью этого элемента.

Пример реализации с единственным дочерним представлением:

```

var OuterView = Backbone.View.extend({
    initialize: function() {
        this.children = {};
        this.child = new Backbone.View();
        this.children[this.child.cid] = this.child;
    },
    render: function() {
        this.$el.html('<div data-view-cid="' + this.child.cid + '"></div>');
        _.each(this.children, function(view, cid) {
            this.$('[' + 'data-view-cid=' + cid + ']').replaceWith(view.el);
        }, this);
    }
});

```

Здесь использование элементов `cid` хорошо иллюстрирует разделение между моделью и ее представлениями, поскольку обращения к представлениям осуществляются по их экземплярам, а не по атрибутам. Запрос всех представлений с определенным атрибутом в соответствующих моделях — обычное действие, но

его невозможно выполнить при использовании рекурсивных подпредставлений или повторяющихся представлений (которые широко распространены), если не указать дополнительные атрибуты, позволяющие отличить дубликаты друг от друга. Использование идентификаторов `cid` решает эту проблему, поскольку дает возможность напрямую обращаться к представлениям.

Большинство разработчиков выбирают решение 1 или 4 по следующим причинам:

- большая часть представлений уже может использоваться в методе `render()` благодаря своему присутствию в DOM;
- при повторном отображении представления `OuterView` не обязательно заново инициализировать те представления, в которых повторная инициализация может привести к утечкам памяти и проблемам с существующими привязками.

Расширения библиотеки Backbone с названиями Marionette и Thorax поддерживают функции создания вложенных представлений и отображения коллекций, с каждым элементом которых связано представление. Библиотека Marionette предоставляет API на языке JavaScript, а API Thorax основано на помощниках шаблонов библиотеки Handlebars.

Спасибо Лукасу Тейлору (Lukas Taylor) и Иану Тейлору (Ian Taylor) за приведенные выше рекомендации.

Управление моделями во вложенных представлениях

Проблема

Как лучше всего управлять моделями во вложенных представлениях?

Решение

Для обращения моделей к атрибутам связанных с ними моделей при использовании вложенных представлений необходимо, чтобы модели заранее «знали» друг о друге; библиотека Backbone не обеспечивает такую возможность своими штатными средствами.

Одно из решений — использовать в каждой дочерней модели атрибут «родитель». Это позволяет перейти по иерархии сначала вверх к родителю, а затем вниз ко всем известным «дочерям». Пусть у нас есть модели `modelA`, `modelB` и `modelC`:

```
// При инициализации модели modelA можно создать ссылку
// на родительскую модель следующим образом:
ModelA = Backbone.Model.extend({
    initialize: function(){
        this.modelB = new modelB();
        this.modelB.parent = this;
        this.modelC = new modelC();
        this.modelC.parent = this;
    }
})
```

Это дает возможность получить доступ к родительской модели из любой функции дочерней модели через элемент `this.parent`.

Мы уже обсудили несколько способов создания вложенных представлений с использованием библиотеки Backbone. Для простоты давайте представим, что мы создаем новое дочернее представление `ViewB` в методе `initialize()` представления `ViewA` (см. ниже). Представление `ViewB` может «выйти за пределы» модели представления `ViewA` и прослушивать изменения любых ее вложенных моделей.

Комментарии в приведенном ниже коде поясняют все выполняемые действия:

```
// определение представления ViewA
ViewA = Backbone.View.extend({
    initialize: function(){
        // Создание экземпляра представления ViewB
        this.viewB = new ViewB();
        // Создание ссылки на дочернее представление
        this.viewB.parentView = this;
        // Добавление представления ViewB в представление ViewA
        $(this.el).append(this.viewB.el);
    }
});
// Определение представления ViewB
ViewB = Backbone.View.extend({
    //...
    initialize: function(){
        // Прослушивание изменений во вложенных моделях
        // родительским представлением ViewA
        this.listenTo(this.model.parent.modelB, "change", this.render);
        this.listenTo(this.model.parent.modelC, "change", this.render);
        // Мы также можем вызвать любой метод родительского представления,
        // если он определен
        // $(this.parentView.el).shake();
    }
});
// Создание экземпляра представления ViewA с моделью ModelA
// viewA создает свой собственный экземпляр представления ViewB
// внутри метода initialize()
var viewA = new ViewA({ model: ModelA });
```

Отображение родительского представления из дочернего представления

Проблема

Как одному из дочерних представлений выполнить отображение родительского представления?

Решение

При использовании вложенных представлений (например, в фотогалерее с функцией просмотра увеличенных фотографий) вы можете столкнуться с ситуацией, когда дочернему представлению требуется отобразить (впервые или повторно) родительское представление. К счастью, решить эту проблему совсем несложно.

Простейшее решение — использовать метод `this.parentView.render();`.

Если желательно осуществить передачу управления, то аналогичное решение можно создать с использованием механизма событий.

Допустим, нам требуется начать отображение при наступлении определенного события — для примера назовем его `somethingHappened`. Родительское представление может сначала связать себя с событием, происходящим в дочернем представлении, а затем отобразить себя при наступлении этого события.

Код родительского представления:

```
// Инициализация родителя
this.listenTo(this.childView, 'somethingHappened', this.render);
// Удаление родителя
this.stopListening(this.childView, 'somethingHappened');
```

Код дочернего представления:

```
// после генерации события
this.trigger('somethingHappened');
```

Дочернее представление сгенерирует событие `somethingHappened`, которое приведет к вызову функции `render` родительского представления.

Спасибо Тэлу Березницкею (Tal Bereznitskey) за приведенную рекомендацию.

Удаление иерархий представлений

Проблема

Если ваше приложение включает множество родительских и дочерних представлений, то, скорее всего, вы захотите удалить все DOM-элементы, связанные с ними, а также отключить всех обработчиков событий от дочерних элементов, когда они перестанут быть нужными.

Решение

В этой ситуации должно подойти предыдущее решение, однако если вам нужен более конкретный пример работы с дочерними представлениями, то он приведен ниже:

```
Backbone.View.prototype.close = function() {
    if (this.onClose) {
        this.onClose();
    }
    this.remove();
};

NewView = Backbone.View.extend({
    initialize: function() {
        this.childViews = [];
    },
    renderChildren: function(item) {
        var itemView = new NewChildView({ model: item });
        $(this.el).prepend(itemView.render());
        this.childViews.push(itemView);
    },
    onClose: function() {
        _(this.childViews).each(function(view) {
            view.close();
        });
    }
});
NewChildView = Backbone.View.extend({
    tagName: 'li',
    render: function() {
    }
});
```

Здесь мы реализуем метод `close()`, который удаляет представление, когда оно перестает быть нужным или должно быть очищено.

В большинстве случаев удаление представления не должно затрагивать связанные с ним модели. Например, возможна ситуация, когда в приложении,

управляющем блогами, вы удаляете представление с комментариями, а в это время другое представление отображает подмножество этих комментариев; очистка коллекции повлияет на второе представление.

Спасибо Дире (Dira) за этот совет.



Составные представления библиотеки Marionette описывались в главе 6.

Отображение иерархий представлений

Проблема

Предположим, что у вас есть коллекция, каждый элемент которой сам является коллекцией. Вы можете отображать отдельные элементы коллекции, даже если они тоже являются коллекциями. Проблема в этой ситуации может заключаться в отображении HTML-кода, отражающего иерархическую структуру данных.

Решение

Самое простое решение – воспользоваться фреймворком наподобие Backbone. Marionette, написанным Дериком Бейли (Derick Bailey). Этот фреймворк включает представления типа `CompositeView` (составные представления).

Основная идея составного представления заключается в том, что оно отображает модель и коллекцию внутри одного и того же представления. Составное представление может отображать единственную модель с помощью шаблона. Оно также берет коллекцию из модели и отображает представление для каждой модели, входящей в состав коллекции. По умолчанию составное представление использует тот тип представления, который вы определили для отображения каждой модели коллекции. Указав экземпляру представления, где находится коллекция, с помощью метода `initialize` вы сможете отобразить рекурсивную иерархию представлений.

В Интернете имеется демонстрация этого решения.

Вы также можете скачать исходные коды и документацию библиотеки Marionette.

Работа с вложенными моделями и коллекциями

Проблема

Библиотека Backbone не обеспечивает встроенную поддержку вложенных моделей и коллекций в расчете на то, что для создания моделей из структурированных данных на клиентской стороне будут использоваться подходящие паттерны.

Как обойти это ограничение?

Решение

Как мы видели, в Backbone принято создавать коллекции, представляющие группы моделей. Нередко в разрабатываемых приложениях требуется встраивать коллекции в модели.

Например, модель «здание» может содержать в себе множество моделей «комната», которые находятся в коллекции «комнаты».

Создайте для каждого здания коллекцию `this.rooms`, которая позволит вам загружать комнаты по необходимости после открытия здания.

```
var Building = Backbone.Model.extend({
  initialize: function(){
    this.rooms = new Rooms;
    this.rooms.url = '/building/' + this.id + '/rooms';
    this.rooms.on("reset", this.updateCounts);
  },
  // ...
});
// Создание новой модели здания
var townHall = new Building;
// загрузка комнат по необходимости после открытия здания
townHall.rooms.fetch({reset: true});
```

Для упрощения работы с вложенными структурами данных также существует несколько плагинов для библиотеки Backbone, например Backbone Relational. Этот плагин поддерживает различные отношения между моделями — «одна к одной», «одна ко многим», «многие к одной», и сопровождается очень хорошей документацией.

Улучшенная валидация свойств моделей

Проблема

Как мы уже знаем из этой книги, метод `validate` модели вызывается методами `set` (если задан параметр `validate`) и `save`. Метод `validate` получает атрибуты модели с обновленными значениями, которые переданы в методы `set` и `save`.

Если мы определяем свой собственный метод `validate`, то библиотека Backbone по умолчанию передает ему все атрибуты модели независимо от того, какие из них были изменены.

По этой причине может оказаться сложно определить, какие именно поля обновляются и нуждаются в проверке, не обрабатывая остальные поля, которые в это время остаются неизменными.

Решение

Чтобы лучше проиллюстрировать эту проблему, рассмотрим типичную регистрационную форму, которая:

- проверяет поля формы, используя событие расфокусировки;
- проверяет каждое поле независимо от корректности других атрибутов модели (то есть от других данных формы).

Вот пример корректного поведения формы. Допустим, что пользователь сначала наводит на нее фокус, а затем снимает его, оставив пустыми HTML-поля для ввода имени, фамилии и адреса электронной почты. В этом случае рядом с каждым полем должно отобразиться сообщение «*This field is required*» («Это поле является обязательным»).

HTML-код:

```
<!doctype html>
<html>
<head>
  <meta charset=utf-8>
  <title>Form Validation - Model#validate</title>
  <script src='http://code.jquery.com/jquery.js'></script>
  <script src='http://underscorejs.org/underscore.js'></script>
  <script src='http://backbonejs.org/backbone.js'></script>
</head>
<body>
  <form>
    <label>First Name</label>
    <input name='firstname'>
    <span data-msg='firstname'></span>
```

продолжение ↗

```

<br>
<label>Last Name</label>
<input name='lastname'>
<span data-msg='lastname'></span>
<br>
<label>Email</label>
<input name='email'>
<span data-msg='email'></span>
</form>
</body>
</html>

```

Можно создать простой механизм валидации этой формы с помощью обычного Backbone-метода `validate`, реализовав его следующим образом:

```

validate: function(attrs) {
    if(!attrs.firstname) return 'first name is empty';
    if(!attrs.lastname) return 'last name is empty';
    if(!attrs.email) return 'email is empty';
}

```

К сожалению, такой метод будет генерировать ошибку `firstname` каждый раз при снятии фокуса с любого из полей, а сообщение об ошибке будет появляться только рядом с полем для ввода имени.

Одно из возможных решений этой проблемы — проверять все поля и возвращать все ошибки:

```

validate: function(attrs) {
    var errors = {};
    if (!attrs.firstname) errors.firstname = 'first name is empty';
    if (!attrs.lastname) errors.lastname = 'last name is empty';
    if (!attrs.email) errors.email = 'email is empty';
    if (!_.isEmpty(errors)) return errors;
}

```

На основе этого кода можно создать решение, в котором для каждого поля ввода нашей формы определяется модель `Field`, работающая с параметрами формы следующим образом:

```

$(function($) {
    `var User = Backbone.Model.extend({
        ``validate: function(attrs) {
            var errors = this.errors = {};
            if (!attrs.firstname) errors.firstname = 'firstname is required';
            if (!attrs.lastname) errors.lastname = 'lastname is required';
            if (!attrs.email) errors.email = 'email is required';
            if (!_.isEmpty(errors)) return errors;
        }
    });
});

```

```

var Field = Backbone.View.extend({
  events: {blur: 'validate'},
  initialize: function() {
    this.name = this.$el.attr('name');
    this.$msg = $('[data-msg=' + this.name + ']');
  },
  validate: function() {
    this.model.set(this.name, this.$el.val(), {validate:true});
    this.$msg.text(this.model.errors[this.name] || '');
  }
});
var user = new User;
$('input').each(function() {
  new Field({el: this, model: user});
});
});

```

Это решение отлично работает, поскольку проверяет корректность каждого атрибута по отдельности и задает сообщение для соответствующего поля при снятии с него фокуса. Демоверсия данного примера, написанная *@braddunbar*, доступна по адресу <http://jsbin.com/afetez/2/edit>.

К сожалению, в этом решении каждая проверка затрагивает все поля, хотя мы отображаем ошибку только для того поля, значение которого изменилось. Если на клиентской стороне нашего приложения есть несколько методов валидации, то нам вряд ли нужно вызывать все эти методы при каждом изменении каждого атрибута, поэтому представленное решение не всегда оптимально.

Backbone.validateAll

Возможно, более удачной альтернативой предыдущему решению является использование плагина `Backbone.validateAll`, написанного *@gfranko* и выполняющего валидацию только определенных (а не всех) свойств моделей (или полей форм).

С помощью этого плагина создадим для нашего примера упрощенную модель пользователя и метод `validate` следующим образом:

```

// Создание новой модели пользователя
var User = Backbone.Model.extend({
  // Шаблоны регулярных выражений
  patterns: {
    specialCharacters: '[^a-zA-Z 0-9]+',
    digits: '[0-9]',
    email: '^@[a-zA-Z0-9._-]+@[a-zA-Z0-9][a-zA-Z0-9.-]*[.]{1}[a-zA-Z]{2,6}$'
  },
  // Валидаторы
  validators: {
    minLength: function(value, minLength) {
      продолжение ↗
    }
  }
});

```

```

        return value.length >= minLength;
    },
    maxLength: function(value, maxLength) {
        return value.length <= maxLength;
    },
    isEmail: function(value) {
        return User.prototype.validators.pattern(value,
            User.prototype.patterns.email);
    },
    hasSpecialCharacter: function(value) {
        return User.prototype.validators.pattern(value,
            User.prototype.patterns.specialCharacters);
    },
    ...
    // Мы можем определить, какие свойства проверяются
    // на равенство null
    validate: function(attrs) {
        var errors = this.errors = {};
        if(attrs.firstname != null) {
            if (!attrs.firstname) {
                errors.firstname = 'firstname is required';
                console.log('first name isEmpty validation called');
            }
            else if(!this.validators.minLength(attrs.firstname, 2))
                errors.firstname = 'firstname is too short';
            else if(!this.validators.maxLength(attrs.firstname, 15))
                errors.firstname = 'firstname is too large';
            else if(this.validators.hasSpecialCharacter(attrs.firstname))
                errors.firstname = 'firstname cannot contain special characters';
        }
        if(attrs.lastname != null) {
            if (!attrs.lastname) {
                errors.lastname = 'lastname is required';
                console.log('last name isEmpty validation called');
            }
            else if(!this.validators.minLength(attrs.lastname, 2))
                errors.lastname = 'lastname is too short';
            else if(!this.validators.maxLength(attrs.lastname, 15))
                errors.lastname = 'lastname is too large';
            else if(this.validators.hasSpecialCharacter(attrs.lastname))
                errors.lastname = 'lastname cannot contain special characters';
        }
    }
}

```

Это позволяет определять в наших методах `validate`, какие поля формы в текущий момент времени изменяются или проверяются, и игнорировать свойства модели, которые остаются неизменными.

Пользоваться такой моделью несложно. Можно определить новый экземпляр модели и затем задать ее данные с помощью параметра `validateAll` для активизации действий, определенных плагином:

```
var user = new User();
user.set({ 'firstname': 'Greg' }, {validate: true, validateAll: false});
```

Вот и все! Логика плагина `Backbone.validateAll` по умолчанию не переопределяет обычную логику библиотеки Backbone, поэтому этот плагин подходит для ситуаций, в которых производительность валидации полей очень важна, и для ситуаций, где она не критична. В любом случае оба решения, приведенные в этом разделе, приводят к нужным результатам.

Backbone.Validation

Как мы видели, метод `validate` библиотеки Backbone по умолчанию имеет значение `undefined`. Вам нужно указать собственный метод валидации моделей. Разработчики часто сталкиваются с проблемами, когда реализуют валидацию в виде вложенных конструкций `if` и `else`, которые могут выйти из-под контроля при усложнении приложения.

Существует еще один полезный плагин для Backbone под названием `Backbone.Validation`, в котором данная задача решается с помощью механизма расширяемого определения правил валидации модели и «закулисного» переопределения метода `validate`.

Этот плагин предоставляет удобную возможность (псевдо)динамической валидации модели с помощью метода `preValidate`. Данный метод позволяет при каждом нажатии клавиши проверять корректность вводимых данных модели, не изменяя при этом саму модель. Вы можете применять к модели любые валидаторы, вызывая метод `preValidate` и передавая ему имя атрибута и значение, которое хотите проверить.

```
var errorMsg = user.preValidate('firstname', 'Greg');
```

Классы для валидации отдельных форм

С учетом сказанного выше оптимальное решение может заключаться не в валидации атрибутов модели. Вместо нее мы создадим функцию, которая проверяет определенную форму; для валидации форм существует много хороших JavaScript-библиотек, способных помочь в решении этой задачи.

Чтобы связать функцию валидации с моделью, включите ее в соответствующий класс:

```
User.validate = function(formElement) {  
    //...  
};
```

Более подробную информацию о плагинах валидации для библиотеки Backbone вы можете получить на ее вики-странице.

Предотвращение конфликтов при использовании нескольких версий Backbone

Проблема

По независящим от вас обстоятельствам вы вынуждены работать с несколькими версиями библиотеки Backbone на одной странице. Как решить эту проблему без конфликтов?

Решение

Как и в большинстве клиентских проектов, код библиотеки Backbone заключен в описании непосредственно вызываемой функции:

```
(function(){
    // Backbone.js
}).call(this);
```

На этом этапе конфигурирования происходит несколько вещей: создается пространство имен Backbone и обеспечивается поддержка нескольких версий Backbone с помощью режима noConflict:

```
var root = this;
var previousBackbone = root.Backbone;
Backbone.noConflict = function() {
    root.Backbone = previousBackbone;
    return this;
};
```

Используйте несколько версий Backbone на одной странице, вызывая noConflict следующим образом:

```
var Backbone19 = Backbone.noConflict();
// Backbone19 ссылается на версию, загруженную последней,
// а `window.Backbone` будет ссылаться на версию,
// загруженную раньше
```

Создание иерархий моделей и представлений

Проблема

Как работает механизм наследования в библиотеке Backbone? Как похожие модели и представления могут совместно использовать код? Как вызывать методы, которые были переопределены?

Решение

Механизм наследования Backbone использует функцию `inherits`, аналогичную функции `goog.inherits`, которая реализована компанией Google в библиотеке Closure. По существу, функция `inherits` создает надлежащую цепочку прототипов.

```
var inherits = function(parent, protoProps, staticProps) {  
  ...
```

Единственное существенное отличие между этими функциями в том, что API библиотеки Backbone принимает два объекта, содержащих методы `instance` и `static`.

Для реализации механизма наследования все объекты Backbone включают в себя метод `extend`, как показано ниже:

```
Model.extend = Collection.extend = Router.extend = View.extend = extend;
```

Разработка приложений с помощью библиотеки Backbone в значительной степени основана на наследовании от этих объектов, воспроизводящих классическую объектно-ориентированную архитектуру.

Механизм наследования Backbone отличается от метода `Object.create` языка ECMAScript 5, поскольку действительно копирует свойства (методы и значения) из одного объекта в другой. Так как этого недостаточно для поддержки модели наследования и классов библиотеки Backbone, выполняются следующие шаги:

1. Проверяется наличие конструктора среди методов экземпляра. Если конструктор существует, то он вызывается; в противном случае вызывается конструктор родительского класса (например, `Backbone.Model`).
2. Вызывается метод `extend` библиотеки Underscore для добавления методов родительского класса в новый дочерний класс.
3. Прототип родительского класса присваивается свойству `prototype` пустой функции-конструктора, а значение свойства `prototype` дочернего класса — новому экземпляру указателя `this`.
4. Метод `extend` библиотеки Underscore дважды вызывается для добавления в дочерний класс статических методов и методов экземпляра.
5. Задается конструктор прототипа дочернего класса и свойство `__super__`.
6. Этот паттерн также используется в классах языка CoffeeScript, поэтому классы библиотеки Backbone совместимы с классами CoffeeScript.

Метод `extend` можно использовать гораздо шире, в частности для примешивания классов. Вы можете определять функционал в одном объекте, а затем почти в буквальном смысле «копировать и вставлять» все его методы и атрибуты в Backbone-объект. Например:

```
var MyMixin = {
  foo: 'bar',
  sayFoo: function(){alert(this.foo);}
};
var MyView = Backbone.View.extend({
  // ...
});
_.extend(MyView.prototype, MyMixin);
var myView = new MyView();
myView.sayFoo(); //=> 'bar'
```

Сделаем шаг вперед и применим этот код к наследованию представлений. Ниже приведен пример расширения одного представления с помощью другого:

```
var Panel = Backbone.View.extend({});
var PanelAdvanced = Panel.extend({});
```

Вызов переопределенных методов

Если ваш класс `Panel` содержит метод `initialize()`, то он не будет вызван, если одноименный метод есть в классе `PanelAdvanced`. По этой причине вам придется явно вызвать метод `initialize()` класса `Panel`:

```
var Panel = Backbone.View.extend({
  initialize: function(options){
    console.log('Panel initialized');
    this.foo = 'bar';
  }
});
var PanelAdvanced = Panel.extend({
  initialize: function(options){
    Panel.prototype.initialize.call(this, [options]);
    console.log('PanelAdvanced initialized');
    console.log(this.foo); // Log: bar
  }
});
// если необходимо, то мы можем также унаследовать класс PanelAdvanced
var PanelAdvancedExtra = PanelAdvanced.extend({
  initialize: function(options){
```

```
    PanelAdvanced.prototype.initialize.call(this, [options]);
    console.log('PanelAdvancedExtra initialized');
}
});

new Panel();
new PanelAdvanced();
new PanelAdvancedExtra();
```

Это не самое элегантное решение, поскольку при большом количестве представлений, наследующих класс `Panel`, вам придется позаботиться о вызове метода `initialize` класса `Panel` в каждом из этих представлений.

Следует заметить, что если класс `Panel` не имеет метода `initialize` в настоящий момент, но вы планируете добавить его в будущем, то вам придется переработать все классы-наследники так, чтобы они не вызывали метод `initialize` класса `Panel`.

Ниже приведен альтернативный способ определения класса `Panel`, благодаря которому представлениям-наследникам не требуется вызывать его метод `initialize`:

```
var Panel = function (options) {
    // поместите сюда весь код инициализации класса Panel
    console.log('Panel initialized');
    this.foo = 'bar';
    Backbone.View.apply(this, [options]);
};

_.extend(Panel.prototype, Backbone.View.prototype, {
    // поместите сюда все методы класса Panel. Пример:
    sayHi: function () {
        console.log('hello from Panel');
    }
});
Panel.extend = Backbone.View.extend;
// классы-наследники Panel, например:
var PanelAdvanced = Panel.extend({
    initialize: function (options) {
        console.log('PanelAdvanced initialized');
        console.log(this.foo);
    }
});
var panelAdvanced = new PanelAdvanced();
// В журнал: Panel initialized, PanelAdvanced initialized, bar
panelAdvanced.sayHi(); // В журнал: hello from Panel
```

С помощью метода `extend` библиотеки Underscore вы можете сэкономить много времени и усилий, избавив себя от написания излишнего кода.

Спасибо Алексу Янгу (Alex Young), Дерику Бейли (Derick Bailey) и ДжонниО (JohnnyO) за эти рекомендации.

Backbone-Super

Библиотека Backbone-Super, написанная Лукасом Ольсоном (Lucas Olson), добавляет метод `_super` в класс `Backbone.Model` с помощью сценария Inheritance Джона Ресайга (John Resig). Вместо того чтобы использовать вызов `Backbone.Model.prototype.set.call` в соответствии с документацией библиотеки `Backbone.js`, можно обратиться к методу `_super`:

```
// наши обычные действия
var OldFashionedNote = Backbone.Model.extend({
    set: function(attributes, options) {
        // вызов метода родителя
        Backbone.Model.prototype.set.call(this, attributes, options);
        // ваш код
        // ...
    }
});
```

Включив в приложение плагин `Backbone-super`, вы сможете выполнить эти же действия следующим образом:

```
// эти же действия при использовании плагина Backbone-super
var Note = Backbone.Model.extend({
    set: function(attributes, options) {
        // вызов метода родителя
        this._super(attributes, options);
        // ваш код
        // ...
    }
});
```

Агрегаторы событий и посредники

Проблема

Как направить события от нескольких источников в один объект?

Решение

С помощью агрегатора событий. Когда разработчики сталкиваются с этой задачей, им обычно приходит в голову идея использовать посреднический механизм, поэтому давайте познакомимся с понятиями *агрегатор событий* и *посредник*, а также с различиями между ними.

Различия между паттернами проектирования часто ограничиваются только их семантикой и предназначением; другими словами, специфика конкретного паттерна заключается в большей степени в языке, на котором он написан, нежели

в его реализации. Паттерны можно сравнить с квадратами, прямоугольниками и многоугольниками: вы можете сложить нужный вам результат из фигур любого из трех типов (если ограничения квадрата позволяют это сделать), однако многоугольники дают возможность образовывать гораздо более сложные и разнообразные формы, чем прямоугольники и квадраты.

Паттерны посредника и агрегатора событий реализованы очень похоже, поэтому иногда может казаться, что они взаимозаменяемы. Тем не менее их семантика и предназначение существенно отличаются друг от друга. Даже если эти паттерны построены на одинаковых ключевых элементах, то между ними существуют четкие различия, из-за которых я считаю их взаимную замену нежелательной.

Агрегатор событий

Согласно Мартину Фаулеру (Martin Fowler), основная идея агрегатора событий — направлять события от нескольких источников через один объект, чтобы другие объекты, которым необходимо подписаться на эти события, не были связаны с их источниками.

Агрегатор событий в Backbone

Простейший агрегатор событий, который можно привести в качестве примера, — это агрегатор событий библиотеки Backbone, встраиваемый в объект:

```
var View1 = Backbone.View.extend({
  // ...
  events: {
    "click .foo": "doIt"
  },
  doIt: function(){
    // генерация события через агрегатор событий
    Backbone.trigger("some:event");
  }
});
var View2 = Backbone.View.extend({
  // ...
  initialize: function(){
    // подписка на событие агрегатора событий
    Backbone.on("some:event", this.doStuff, this);
  },
  doStuff: function(){
    // ...
  }
})
```

В этом примере первое представление генерирует событие при щелчке по DOM-элементу. Генерация этого события происходит через встроенный агрегатор

событий — объект Backbone. Разумеется, Backbone позволяет с легкостью создать собственный агрегатор событий; при работе с агрегаторами нужно иметь в виду лишь несколько ключевых аспектов, чтобы не усложнять код приложения.

Агрегатор событий jQuery

Библиотека jQuery имеет встроенный агрегатор событий, который, хотя и называется иначе, но находится в области видимости DOM-событий и выглядит схоже с агрегатором событий Backbone:

```
$("#mainArticle").on("click", function(e){
    // обработка события щелчка по любому элементу,
    // расположенному под элементом #mainArticle
});
```

Приведенный выше код создает функцию-обработчик, которая ждет событий от неопределенного числа источников и позволяет произвольному количеству слушателей подписаться на эти события. Библиотека jQuery лишь делает агрегатор событий видимым для DOM.

Посредник

Посредник — это объект, который координирует взаимодействие (логику и поведение) нескольких объектов. Он принимает решения о том, когда и какие объекты вызывать, в зависимости от действий (или их отсутствия) других объектов и источников данных.

Посредник в Backbone

Библиотека Backbone, в отличие от многих других MV*-фреймворков, не имеет встроенного посредника. Тем не менее это не означает, что вы не можете создать его одной строкой кода:

```
var mediator = {};
```

Конечно, это всего лишь объектная константа JavaScript. Напомню, что сейчас мы говорим только о семантике. Посредник предназначен для управления взаимодействием между объектами, а для этого нам не нужно ничего, кроме объектной константы.

```
var orgChart = {
    addNewEmployee: function(){
        // метод getEmployeeDetail создает представление,
```

```
// с которым взаимодействуют пользователи
var employeeDetail = this.getEmployeeDetail();
// когда все данные о сотруднике готовы, посредник (объект 'orgchart')
// решает, что произойдет дальше
employeeDetail.on("complete", function(employee){
    // создание объектов с дополнительными событиями, которые
    // используются посредником для выполнения дополнительных действий
    var managerSelector = this.selectManager(employee);
    managerSelector.on("save", function(employee){
        employee.save();
    });
});
},
// ...
})
```

Этот пример демонстрирует простейшую реализацию объекта-посредника и Backbone-объектов, которые могут генерировать события и подписываться на них. Раньше я называл посредника «процессным объектом», однако его смысл заключается именно в координации. Посредник управляет согласованной работой множества других объектов, группируя принципы их взаимодействия в едином объекте. В результате совместная работа компонентов приложения становится понятнее и проще в поддержке.

Сходства и различия

Безусловно, представленные здесь примеры агрегатора событий и посредника похожи в двух ключевых аспектах: событиях и сторонних объектах. Тем не менее эти сходства, в лучшем случае, являются внешними. Если мы вдумаемся в смысл примера и поймем, что представленные реализации могут быть совершенно иными, то суть этих паттернов становится очевиднее.

События

В предыдущих примерах и агрегатор событий, и посредник использовали события. То, что агрегатор событий работает с событиями, очевидно из его названия. Посредник же использует события только для упрощения работы приложения с библиотекой Backbone. Нет никаких правил, обязывающих посредника работать с событиями. Можно создать посредника с помощью методов обратного вызова, передать ссылку на посредника дочернему объекту или воспользоваться любым другим способом.

Тем не менее события используются обоими паттернами. Агрегатор событий придуман для работы с событиями, а посредник использует их лишь потому, что это удобно.

Сторонние объекты

И агрегатор событий, и посредник специально используют сторонний объект для упрощения схемы. Агрегатор событий сам является сторонним объектом как для источника, так и для подписчика событий. Он действует как центральный узел, через который проходят события. Посредник тоже является сторонним по отношению к другим объектам. Так в чем же разница? Почему мы не называем агрегатора событий посредником? Ответ в значительной степени объясняется тем, где реализована логика и процессы приложения.

В случае агрегатора событий сторонний объект лишь упрощает передачу событий от неизвестного количества источников неизвестному количеству обработчиков. Все процессы и бизнес-логика, которую необходимо активизировать, помещаются непосредственно в объект, генерирующий события и объекты, которые обрабатывают эти события.

В случае посредника бизнес-логика и процессы помещаются в самого посредника. Он принимает решения о том, когда вызывать методы объекта и обновлять его атрибуты, основываясь на известной ему информации. Посредник содержит процессы и обеспечивает требуемое поведение системы, координируя множество объектов. Отдельные объекты, участвующие в процессе, знают о том, как выполнять свои задачи, а посредник принимает решения на более высоком уровне и активизирует объекты в нужные моменты времени.

Агрегатор событий реализует модель взаимодействия «запустил и забыл». Генерирующий событие объект не интересует наличие подписчиков на это событие: он просто генерирует его и продолжает работать. Посредник тоже может использовать события для принятия решений, но не по принципу «запустил и забыл»: он анализирует известный набор источников данных и действий для того, чтобы активизировать и координировать дополнительные действия определенного множества участников (объектов).

Когда и какие отношения использовать

Понимание сходств и различий между агрегатором событий и посредником важно с точки зрения семантики. Столь же важно понимать, в каких ситуациях следует использовать каждый из паттернов. Основной смысл и предназначение этих паттернов отвечают на второй вопрос, однако опыт их использования поможет понять тонкости и нюансы, влияющие на принятие необходимых решений.

Использование агрегатора событий

Агрегатор событий обычно используется в ситуациях, когда приложение содержит слишком много объектов, чтобы прослушивать их непосредственно, либо объекты никак не связаны между собой.

Если между двумя объектами (например, родительским и дочерним представлением) существует прямая связь, то агрегатор событий не принесет ощутимой пользы. Родительское представление способно обрабатывать события, которые генерирует дочернее представление. Это можно чаще всего видеть на примере коллекции и модели Backbone, где все события моделей «проходят» через родительскую коллекцию. Коллекция часто использует события моделей для того, чтобы изменять собственное состояние или состояние других моделей. Хорошим примером этого механизма является выборочная обработка событий коллекции.

Удачным примером агрегатора событий служит метод `on` библиотеки jQuery в ситуации, когда требуется прослушивать большое количество объектов. Если у вас 10, 20, 100 или 200 DOM-элементов, способных генерировать событие щелчка, то создавать слушателя, индивидуально работающего с этими элементами, — плохая идея. Такой подход способен ухудшить производительность приложения и качество работы его пользовательского интерфейса. В то же время использование jQuery-метода `on` позволяет агрегировать все эти события и использовать один обработчик вместо 10, 20, 100 или 200.

Уместно пользоваться агрегаторами событий и при косвенных отношениях. В Backbone-приложениях часто существует несколько представлений, не связанных между собой напрямую, но которым нужно взаимодействовать друг с другом. К примеру, меню может включать в себя представление, обрабатывающее щелчки по его элементам, но мы не хотели бы, чтобы меню было непосредственно связано с представлениями содержимого, которые отображают подробную информацию при щелчке по элементу меню.

Объединение меню с содержимым значительно усложнило бы поддержку кода в долгосрочной перспективе. Вместо этого воспользуемся агрегатором событий для генерации событий `menu:click:foo` и объектом `foo`, обрабатывающим событие щелчка и отображающим свое содержимое на экране.

Использование посредника

Посредника лучше всего использовать в ситуации, когда два или более объектов имеют косвенные «рабочие» взаимосвязи, и бизнес-логика или процессы приложения должны координировать взаимодействия между этими объектами.

Хороший пример такой ситуации приведен в интерфейсе мастера приложения orgChart в разделе «Посредник» этой главы. Мастер использует несколько представлений. Вместо того чтобы связывать представления с помощью прямых ссылок друг на друга, мы разделим их и более явно опишем взаимодействие между ними с помощью посредника.

Посредник отделяет описание взаимодействия от реализации и создает более естественную и наглядную абстракцию на более высоком уровне. Нам больше

не нужно разбираться в деталях каждого представления, чтобы понять, как представления взаимодействуют друг с другом.

Совместное использование агрегатора событий и посредника

Тонкость различий между агрегатором событий и посредником, а также причину, по которой их не следует взаимозаменять, лучше всего иллюстрирует их совместное использование. Пример меню с участием агрегатора событий также удачно демонстрирует посредника.

Щелчок по элементу меню может вызывать цепочку изменений в приложении. Некоторые изменения будут независимыми от остальных, поэтому использование агрегатора в такой ситуации имеет смысл. Тем не менее другие изменения могут быть внутренне связаны между собой, и для их обработки можно воспользоваться посредником. Затем настроим посредника на прослушивание агрегатора событий. Посредник может выполнять свою собственную логику и координировать множество объектов, которые связаны между собой, но не связаны с источником событий.

```
var MenuItem = Backbone.View.extend({
  events: {
    "click .thatThing": "clickedIt"
  },
  clickedIt: function(e){
    e.preventDefault();
    // предполагаем, что этот код генерирует событие "menu:click:foo"
    Backbone.trigger("menu:click:" + this.model.get("name"));
  }
});
// ... где-то в другой части приложения
var MyWorkflow = function(){
  Backbone.on("menu:click:foo", this.doStuff, this);
};
MyWorkflow.prototype.doStuff = function(){
  // создание нескольких объектов
  // задание обработчиков событий для этих объектов
  // координация взаимодействия всех объектов,
  // обеспечивающая осмысленный процесс
};
```

При щелчке по пункту меню `MenuItem` с соответствующей моделью генерируется событие `menu:click:foo`. Если экземпляр объекта `MyWorkflow` уже создан, то он обрабатывает это событие и обеспечивает выполнение требуемых процессов и взаимодействие с пользователем, координируя все известные ему объекты.

Сочетание агрегатора событий и посредника сделало код и само приложение гораздо более осмысленным. Теперь благодаря агрегатору событий у нас есть четкое разделение между меню и логикой приложения, а сама логика понятна и удобна в поддержке за счет использования посредника.

Язык паттернов: семантика

Ко всем нашим рассуждениям есть одно существенное дополнение: семантика. Цель и термины коммуникации с использованием упомянутых паттернов имеют смысл только тогда, когда все взаимодействующие стороны одинаково понимают языки.

Если я говорю «лук», то что я имею в виду — зеленый лук, репчатый лук или лук для стрельбы? Как говорит Шарон Чичелли (Sharon Cichelli), «смысл терминов будет сохранять свою важность до тех пор, пока мы не научимся общаться не только с помощью языка».

8

Модульная разработка

Модульным мы называем приложение, которое состоит из множества отдельных, слабо связанных между собой наборов функций, хранящихся в виде модулей. Как вы, вероятно, знаете, слабая связь между модулями приложения делает его поддержку более удобной за счет устранения зависимостей там, где это возможно. Грамотно реализованная модульная структура системы позволяет легко понять, как изменения в одной ее части влияют на другую.

Существующий стандарт языка JavaScript (ЕСМА-262), в отличие от некоторых более традиционных языков программирования, не предоставляет разработчикам понятных и хорошо организованных методов импортирования таких модулей кода.

Вместо этого разработчики вынуждены пользоваться различными вариантами модульных или объектных констант в сочетании с тегами `<script>` или загрузчиком сценариев. Сценарии модулей связываются воедино в DOM-модели, при этом пространства имен описываются единственным глобальным объектом, что может приводить к конфликтам имен. Кроме того, не существует элегантного способа управления зависимостями, который бы не требовал выполнения некоторых действий вручную или использования сторонних инструментов.

Возможно, «штатные» решения этих проблем появятся в модулях, которые предполагается ввести в стандарте ES6 (новой официальной спецификации языка JavaScript), однако уже сегодня вы можете приступить к созданию модульных приложений на JavaScript, и сделать это довольно легко.

В этой главе мы научимся четко разделять код приложения на управляемые модули с помощью технологии AMD (Asynchronous Module Definition, асинхронное определение модулей) и библиотеки *RequireJS*. Мы также рассмотрим альтернативное решение этой задачи с помощью инструмента *Lumbar*, который загружает модули с помощью маршрутов.

Организация модулей с помощью RequireJS и AMD

RequireJS — популярный загрузчик сценариев, написанный разработчиком Джеймсом Берком (James Burke), который принимал активное участие в создании формата AMD-модуля (его мы рассмотрим в ближайшее время). Библиотека RequireJS предоставляет различные возможности, в том числе позволяет загружать несколько сценариев, определять модули с зависимостями и без них, а также загружать зависимости, не являющиеся сценариями (например, текстовые файлы).

Проблемы поддержки множества файлов сценариев

Возможно, использование библиотеки RequireJS покажется вам малоэффективным. В конце концов, можно просто загрузить JavaScript-файлы с помощью нескольких тегов `<script>`. Но не стоит забывать, что у такого подхода много недостатков, в том числе избыточные затраты, связанные с использованием протокола HTTP.

При загрузке файла, на который ссылается тег `<script>`, браузер посыпает HTTP-запрос на скачивание содержимого этого файла. Каждому файлу требуется новый запрос, что порождает несколько проблем:

- Число запросов, одновременно посыпаемое браузером, ограничено, поэтому загрузка множества файлов происходит медленно. Количество одновременно загружаемых файлов зависит от браузера и его пользовательских настроек, обычно от 4 до 8. При разработке Backbone-приложения рекомендуется разбивать его на множество JS-файлов, что может легко привести к превышению данного ограничения. Вы, конечно, можете поместить код в один файл при загрузке, однако это не решает другой проблемы.
- Сценарии загружаются *синхронно*. Другими словами, браузер не может продолжать отображение страницы в процессе загрузки сценария.

Инструменты вроде RequireJS выполняют асинхронную загрузку сценариев. Для этого нам потребуется немного исправить код приложения (нельзя просто заменить элементы `<script>` на фрагмент RequireJS-кода), однако результат оправдывает себя:

- процесс асинхронной загрузки сценариев ничего не блокирует. Браузер может продолжать отображение оставшейся части страницы во время скачивания сценариев, что сокращает длительность начальной загрузки страницы;
- мы можем гибко определять, когда модули должны загружаться, а также обеспечивать правильный порядок загрузки модулей, имеющих зависимости.

Для чего требуется улучшать управление зависимостями

Управление зависимостями — сложная задача, особенно если вы создаете JavaScript-код в браузере. Чтобы сделать нечто похожее на управление зависимостями, можно просто упорядочить теги `<script>` так, чтобы код, который зависит от другого файла, загружался позже этого файла. Как было сказано выше, такой подход имеет несколько недостатков: загрузка нескольких файлов данным способом отрицательно влияет на производительность и создает риск в ситуации, когда файлы требуется загрузить в определенном порядке.

Библиотека RequireJS — эффективный инструмент, позволяющий загружать код по необходимости. Вместо того чтобы скачивать весь JavaScript-код при начальной загрузке страницы, динамически загружайте отдельные модули, когда их код потребуется приложению. Это позволяет не загружать весь код при запуске приложения и, как следствие, ускоряет его начальную загрузку.

Задумайтесь о том, как работает веб-клиент Gmail. Когда пользователь впервые открывает страницу Gmail, Google скрывает ее виджеты (например, чат) до тех пор, пока пользователь не проявит желание пользоваться ими, щелкнув по значку «раскрыть». Динамическая загрузка зависимостей позволяет Google загрузить модуль чата именно в этот момент, а не заставлять всех пользователей загружать модуль чата при инициализации страницы. Это повышает производительность, сокращает длительность загрузки и создает удобство при разработке больших приложений. С ростом кодовой базы приложения механизм динамической загрузки становится еще важнее.

Особо отметим, что нет ничего плохого в разработке приложений без загрузчики сценариев; тем не менее использование инструментов вроде RequireJS дает приложению значительные преимущества.

Асинхронное определение модулей (AMD)

Загрузчик RequireJS реализует спецификацию AMD, которая определяет метод написания модульного кода и управления зависимостями. На веб-сайте RequireJS существует раздел, в котором объясняются причины создания AMD.



Формат AMD обеспечивает востребованную модульную структуру, которая эффективнее существующей практики написания набора тегов `<script>` с неявными зависимостями, требующими управления вручную, и удобна для непосредственного использования в браузере. Эта структура обеспечивает эффективную отладку приложения и не требует его специальной доработки для запуска на сервере.

Создание AMD-модулей с помощью RequireJS

Как было сказано, главная цель формата AMD — предоставить разработчикам решение для модульной разработки JavaScript-приложений. Для использования AMD совместно с загрузчиком сценариев необходимо знать два ключевых аспекта — метод `define()` для определения модулей и метод `require()` для загрузки зависимостей. Метод `define()` определяет именованные и неименованные модули с помощью следующей сигнатуры:

```
define(  
    module_id /*необязательно*/,  
    [dependencies] /*необязательно*/,  
    definition function /*функция создания экземпляра модуля или объекта */  
)
```

Как видно из комментариев в этом коде, аргумент `module_id` необязателен и, как правило, нужен лишь при использовании инструментов конкатенации, сторонних по отношению к AMD (а также в ряде других нетипичных ситуаций). Модуль, в котором этот аргумент не используется, называется *анонимным*. Библиотека RequireJS использует в качестве идентификатора анонимного модуля путь к его файлу, поэтому при вызове метода `define()` не следует указывать идентификатор модуля во избежание дублирования.

Аргумент `dependencies` представляет собой массив модулей, от которых зависит данный модуль; третий аргумент — это фабрика, которая может быть функцией, создающей экземпляр модуля, или объектом.

Мы можем определить простой модуль, совместимый с RequireJS, с помощью метода `define()` следующим образом:

```
// Здесь идентификатор модуля не используется, чтобы модуль был анонимным  
define(['foo', 'bar'],  
// функция определения модуля  
// зависимости (foo и bar) связаны с ее параметрами  
function ( foo, bar ) {  
    // возвращает значение, определяющее экспорт модуля  
    // (то есть функционал, который мы хотим открыть для доступа)  
    // создайте здесь свой собственный модуль  
    var myModule = {  
        doStuff:function(){  
            console.log('Yay! Stuff');  
        }  
    }  
    return myModule;  
});
```

Загрузчик RequireJS автоматически добавляет расширение `.js` к именам сценариев, поэтому его не нужно указывать при определении зависимостей.

Альтернативный синтаксис

Существует другая, «подслащенная» версия метода `define()`, позволяющая определять зависимости как локальные переменные с помощью метода `require()`. Эта версия покажется знакомой тем, кто пользовался библиотекой Node; с ее помощью проще добавлять и удалять зависимости. Ниже приведен вариант предыдущего примера с использованием альтернативного синтаксиса:

```
// Здесь идентификатор модуля не используется, чтобы модуль был анонимным
define(function(require){
    // функция определения модуля
    // зависимости (foo и bar) определены как локальные переменные
    var foo = require('foo'),
        bar = require('bar');
    // возврат значения, которое определяет экспорты модуля
    // (то есть функционал, который мы хотим открыть для доступа)
    // создайте здесь свой собственный модуль
    var myModule = {
        doStuff:function(){
            console.log('Yay! Stuff');
        }
    }
    return myModule;
});
```

Метод `require()` обычно используется для загрузки кода в JavaScript-файл верхнего уровня или в модуль при динамической доставке зависимостей. Вот пример его использования:

```
// Пусть 'foo' и 'bar' – два внешних модуля
// В этом примере "экспорты" обоих загруженных модулей передаются как
// аргументы функции обратного вызова (foo и bar),
// чтобы к ним можно было получить доступ аналогичным образом
require( ['foo', 'bar'], function ( foo, bar ) {
    // остальная часть вашего кода
    foo.doSomething();
});
```

Спецификация AMD подробнее рассмотрена в моей публикации «Writing Modular JS» («Написание модульных приложений на JS»). Мы кратко ознакомимся с определением и использованием модулей в этой книге при изучении структурированных примеров, использующих RequireJS.

Начало работы с RequireJS

Перед тем как приступить к работе с RequireJS и Backbone, создадим простейший RequireJS-проект и продемонстрируем, как он работает. Первое, что нужно

сделать, — это скачать библиотеку RequireJS. При загрузке сценария RequireJS в HTML-файл укажите, где находится главный сценарий JavaScript (обычно он имеет имя вроде `app.js` и является главной точкой входа в приложение). Для этого добавьте атрибут `data-main` в тег `<script>`:

```
<script data-main="app.js" src="lib/require.js"></script>
```

Теперь RequireJS автоматически загрузит файл `app.js`.

Конфигурирование RequireJS

В главном JavaScript-файле, который загружается с помощью атрибута `data-main`, сконфигурируйте загрузку остальной части вашего приложения библиотекой RequireJS. Для этого вызовите метод `require.config` и передайте ему объект:

```
require.config({
    // пары "ключ-значение" вашего приложения
    baseUrl: "app",
    // Обычно тот же каталог, в котором находится сценарий верхнего уровня,
    // указанный в атрибуте data-main
    paths: {},
    // set up custom paths to libraries, or paths to RequireJS plug-ins
    shim: {}, // используется для настройки адаптеров (подробнее см. ниже)
});
```

Главная причина, по которой требуется конфигурировать RequireJS, — добавление адаптеров (мы рассмотрим их ниже). О других конфигурационных параметрах библиотеки RequireJS вы узнаете, познакомившись с документацией.

Адаптеры для RequireJS. В идеале каждая библиотека, совместно используемая с RequireJS, должна поддерживать AMD, то есть определять библиотеку как модуль с помощью метода `define`. Тем не менее некоторые библиотеки, в том числе Backbone и одна из ее зависимостей *Underscore*, не предоставляют такую поддержку. К счастью, RequireJS позволяет решить эту проблему.

Чтобы продемонстрировать это решение, создадим адаптер для библиотеки *Underscore*, а затем — для Backbone. Реализовать адаптеры очень легко:

```
require.config({
    shim: {
        'lib/underscore': {
            exports: '_'
        }
    }
});
```

Обратите внимание, что при определении путей для RequireJS следует опускать окончание .js в названиях сценариев.

Здесь важна строка exports: '_', указывающая RequireJS, что сценарий в файле lib/underscore.js создает глобальную переменную с именем _ вместо определения модуля. Теперь, когда мы указываем библиотеку Underscore в качестве зависимости, RequireJS передаст нам глобальную переменную так, как будто это модуль, определенный сценарием. Мы также можем создать адаптер и для Backbone:

```
require.config({
    shim: {
        'lib/underscore': {
            exports: '_'
        },
        'lib/backbone': {
            deps: ['lib/underscore', 'jquery'],
            exports: 'Backbone'
        }
    }
});
```

Аналогично, такая конфигурация указывает библиотеке RequireJS вернуть глобальную переменную Backbone, экспортируемую библиотекой Backbone, однако на этот раз определяются зависимости библиотеки Backbone.

Это означает, что при каждом запуске следующего кода он сначала проверит наличие зависимостей, а затем передаст глобальный объект Backbone в функцию обратного вызова:

```
require( 'lib/backbone', function( Backbone ) {...} );
```

Эти действия нужно выполнять не над всеми библиотеками, а лишь над теми, которые не поддерживают AMD. Например, библиотека jQuery версии 1.7 включает в себя поддержку AMD.

Вы можете найти дополнительную информацию об использовании библиотеки RequireJS в ее документации.

Настраиваемые пути

Иногда вводить длинные пути к файлам вроде lib/backbone неудобно. Библиотека RequireJS позволяет нам задавать собственные пути в конфигурационном объекте. В следующем примере при использовании ссылки underscore RequireJS будет искать файл lib/underscore.js:

```
require.config({
  paths: {
    'underscore': 'lib/underscore'
  }
});
```

Разумеется, эту конструкцию можно использовать вместе с адаптером:

```
require.config({
  paths: {
    'underscore': 'lib/underscore'
  },
  shim: {
    'underscore': {
      exports: '_'
    }
  }
});
```

Проследите за тем, чтобы в настройках адаптера тоже использовался заданный вами путь. Теперь создадим адаптер для Underscore с применением этого пути:

```
require( ['underscore'], function(_) {
  // ваш код
});
```

Пример совместного использования Require.js и Backbone

Теперь, когда мы знаем, как определять AMD-модули, давайте рассмотрим «обертывание» компонентов вроде представлений и коллекций, позволяющее легко загружать их в виде зависимостей любой части приложения, использующей их. В простейшем случае модель Backbone требует лишь двух библиотек: Backbone и Underscore.js. Поскольку они являются зависимостями, мы можем задать их при определении новых модулей. Будьте внимательны: следующие примеры рассчитаны на то, что вы уже создали адаптеры для связи RequireJS с Backbone и Underscore, как было показано выше.

«Обертывание» моделей, представлений и других компонентов с помощью AMD

Пример определения модели может выглядеть следующим образом:

```
define(['underscore', 'backbone'], function(_, Backbone) {
  var myModel = Backbone.Model.extend({
    // Атрибуты по умолчанию
    defaults: {
```

продолжение ⇨

```
        content: 'hello world',
    },
    // пустой метод инициализации
initialize: function() {
},
clear: function() {
    this.destroy();
    this.view.remove();
}
});
return myModel;
});
```

Обратите внимание, что мы создаем псевдоним `_` для экземпляра `Underscore.js`, а к `Backbone` обращаемся по ее полному имени `Backbone`, что позволяет нам легко преобразовать код, не соответствующий формату AMD, так, чтобы он использовал этот формат. Представление, которому требуются другие зависимости (например, `jQuery`), можно определить так:

```
define([
    'jquery',
    'underscore',
    'backbone',
    'collections/mycollection',
    'views/myview'
], function($, _, Backbone, myCollection, myView){
var AppView = Backbone.View.extend({
    ...
});
```

Использование знака доллара (`$`) в качестве псевдонима еще раз упрощает инкапсулирование любой части приложения с помощью AMD.

Подобным образом вы можете легко структурировать `Backbone` по своему желанию. Рекомендуется хранить модули в отдельных папках: например, создайте отдельные папки для моделей, коллекций, представлений и т. д. Библиотека RequireJS не накладывает требований на структуру папок; для получения доступа к файлу достаточно правильно указывать путь при использовании конструкции `require`.

В этой главе я создал с использованием RequireJS очень простое `Backbone`-приложение, которое вы найдете на GitHub. Это приложение управляет складом и предназначено для менеджера магазина. Оно не слишком функционально: менеджер может только добавлять новые товары и сортировать товары по цене.

Простота этого приложения позволяет сконцентрироваться на реализации возможностей RequireJS вместо того, чтобы работать со сложной JavaScript-и Backbone-логикой.

Основа приложения — модель `Item`, описывающая единицу товара на складе. Ее реализация выглядит очень просто:

```
define( ["lib/backbone"], function ( Backbone ) {
    var Item = Backbone.Model.extend({
        defaults: {
            price: 35,
            photo: "http://www.placedog.com/100/100"
        }
    });
    return Item;
});
```

Преобразование отдельной модели, коллекции, представления или другого компонента в AMD-формат, совместимый с RequireJS, выполняется очень просто. Как правило, все, что нужно для этого, — первая строка, вызов `define` и возврат объекта, который был определен (в нашем случае `Item`).

Теперь создадим представление для единицы товара:

```
define( ["lib/backbone"], function ( Backbone ) {
    var ItemView = Backbone.View.extend({
        tagName: "div",
        className: "item-wrap",
        template: _.template($("#itemTemplate").html()),
        render: function() {
            this.$el.html(this.template(this.model.toJSON()));
            return this;
        }
    });
    return ItemView;
});
```

Это представление не зависит от модели, с которой будет использоваться, поэтому его единственной зависимостью является `Backbone`. В остальном же это обычное `Backbone`-представление. В нем нет ничего особенного, кроме возврата объекта и использования `define`, чтобы библиотека RequireJS могла работать с ним. Теперь создадим коллекцию для просмотра списка товаров. Для этого сошлемся на модель `Item`, которую мы добавим в качестве зависимости:

```
define(["lib/backbone", "models/item"], function(Backbone, Item) {
    var Cart = Backbone.Collection.extend({
        model: Item,
        initialize: function() {
            this.on("add", this.updateSet, this);
        },
        updateSet: function() {
            items = this.models;
        }
    });
    return Cart;
});
```

Я дал этой коллекции название `Cart`¹, поскольку она содержит в себе набор товаров. Так как модель `Item` является второй зависимостью, я могу связать с ней переменную `Item`, передав ее вторым аргументом в функцию обратного вызова, а затем использовать ее внутри коллекции.

Теперь рассмотрим представление для этой коллекции (в приложении этот файл существенно больше, здесь приведены его фрагменты для простоты изучения):

```
define(["lib/backbone", "models/item", "views/itemview"],
function(Backbone, Item, itemView) {
    var ItemCollectionView = Backbone.View.extend({
        el: '#yourcart',
        initialize: function(collection) {
            this.collection = collection;
            this.render();
            this.collection.on("reset", this.render, this);
        },
        render: function() {
            this.$el.html("");
            this.collection.each(function(item) {
                this.renderItem(item);
            }, this);
        },
        renderItem: function(item) {
            var itemView = new itemView({model: item});
            this.$el.append(itemView.render().el);
        },
        // more methods here removed
    });
    return ItemCollectionView;
});
```

Если вы поняли общий принцип, то не увидите здесь ничего нового. Определите каждый объект (модель, представление, коллекцию, маршрутизатор и т. д.) с помощью RequireJS, а затем укажите эти объекты в качестве зависимостей для тех объектов, которым они необходимы. Напомню, что полную версию этого приложения вы можете найти на GitHub.

Если вы хотите познакомиться с другими приложениями, то хороший пример использования библиотеки RequireJS для структурирования Backbone-приложений — репозиторий Backbone Stack Петера Хокинса (Pete Hawkins). Грег Франко (Greg Franko) написал обзор о том, как он использует Backbone совместно с Require, а публикация Джереми Кана (Jeremy Kahn) обстоятельно описывает его собственный подход. Первое полнофункциональное приложение, с которым имеет смысл ознакомиться в качестве примера, — это версия TodoMVC, использующая Backbone и Require.

¹ Англ. «корзина». — Примеч. перев.

Внешнее хранение шаблонов с помощью библиотеки RequireJS и плагина Text

Помещать шаблоны во внешние файлы довольно просто, если они находятся в формате Underscore, Mustache, Handlebars или любом другом текстовом формате. Рассмотрим, как сделать это с помощью библиотеки RequireJS.

У RequireJS есть специальный плагин `text.js`, который используется для загрузки текстовых файлов зависимостей. Чтобы воспользоваться текстовым плагином, выполните следующие шаги:

1. Скачайте плагин и поместите его в директорию, где находится главный JS-файл приложения, или в подходящую поддиректорию.
2. Затем включите плагин `text.js` в параметры начальной конфигурации RequireJS. Следующий фрагмент кода рассчитан на то, что на момент его запуска библиотека RequireJS уже включена в страницу.

```
require.config( {
    paths: {
        'text': 'libs/require/text',
    },
    baseUrl: 'app'
} );
```

3. Если в качестве зависимости указан префикс `text!`, то библиотека RequireJS автоматически загрузит текстовый плагин и будет обращаться с зависимостью как с текстовым ресурсом. Типичный пример применения этого механизма на практике выглядит так:

```
require(['js/app', 'text!templates/mainView.html'],
    function( app, mainView ) {
        // содержимое файла mainView будет
        // загружено в mainView для доступа к нему.
    }
);
```

4. Наконец, мы можем использовать загруженный текстовый ресурс в качестве шаблона. Возможно, вы привыкли хранить HTML-шаблоны во встроенном виде с помощью сценария с определенным идентификатором.

При использовании микрошаблонов библиотеки Underscore.js (и jQuery) это будет выглядеть так:

○ HTML

```
<script type="text/template" id="mainViewTemplate">
  <% _.each( person, function( person_item ){ %>
    <li><%= person_item.get('name') %></li>
  <% }); %>
</script>
```

○ JS

```
var compiled_template = _.template( $('#mainViewTemplate').html() );
```

Тем не менее с помощью RequireJS и текстового плагина достаточно сохранить тот же шаблон во внешнем текстовом файле (например, *mainView.html*) и сделать следующее:

```
require(['js/app', 'text!templates/mainView.html'],
  function(app, mainView){
    var compiled_template = _.template( mainView );
  }
);
```

Вот и всё! Примените свой шаблон к представлению в Backbone следующим образом:

```
collection.someview.$el.html( compiled_template
  ( { results: collection.models } ) );
```

У всех шаблонизаторов есть свои методы компиляции шаблонов, однако если вы поняли приведенные выше примеры, то заменить Underscore на любой другой шаблонизатор совсем несложно.

Оптимизация Backbone-приложений для рабочей среды с помощью оптимизатора RequireJS

Следующий важный шаг после написания приложения — его подготовка к развертыванию в рабочей среде. Большинство сложных приложений состоит из нескольких сценариев, поэтому если вы оптимизируете, минимизируете и объедините их до того, как опубликуете приложение, вы сократите число сценариев, которое потребуется скачать вашим пользователям.

В этом вам поможет командно-строковый инструмент *r.js*, который оптимизирует проекты RequireJS. Вот некоторые из его возможностей:

- объединение определенных сценариев и уменьшение их размера с помощью внешних инструментов, таких как *UglifyJS* (он используется по умолчанию) или *Google Closure Compiler*, для оптимальной загрузки в браузер, с сохранением возможности динамической загрузки модулей;
- оптимизация CSS и таблиц стилей путем встраивания CSS-файлов, импортированных с помощью правила *@import*, вырезка комментариев и пр.;
- возможность запускать AMD-проекты в Node и Rhino (подробнее об этом чуть позже).

Если вы захотите использовать единственный файл, в который включены все зависимости приложения, то инструмент *r.js* поможет вам и в этом. Хотя библиотека RequireJS поддерживает отложенную загрузку, ваше приложение может оказаться достаточно компактным, чтобы для его загрузки потребовался лишь HTTP-запрос к единственному файлу сценария.

Обратите внимание на словосочетание *определенных сценариев* в первом пункте списка. Оптимизатор библиотеки RequireJS объединяет только модульные сценарии, которые были указаны как строковые литералы в вызовах *require* и *define* (ими вы, вероятно, воспользовались). Согласно документации оптимизатора это означает, что модули, определенные так, как показано ниже, будут успешно объединены:

```
define(['jquery', 'backbone', 'underscore', 'collections/sample', 'views/test'],
  function($, Backbone, _, Sample, Test){
    //...
});
```

Тем не менее следующие динамические зависимости будут проигнорированы:

```
var models = someCondition ? ['models/ab', 'models/ac'] :
['models/ba', 'models/bc'];
define(['jquery', 'backbone', 'underscore'].concat(models),
  function($, Backbone, _, firstModel, secondModel){
    //...
});
```

Это сделано специально, поскольку позволяет динамически загружать зависимости/модули даже после оптимизации.

Хотя оптимизатор RequireJS успешно работает как в среде Node, так и в среде Java, я рекомендую запускать его в Node, поскольку в ней он работает значительно быстрее.

Сначала скачайте *r.js* со страницы загрузки RequireJS или воспользуйтесь NPM. Чтобы собрать проект с использованием *r.js*, нам потребуется создать новый профиль сборки.

Если код нашего приложения и внешние зависимости находятся в каталоге `app/libs`, то профиль сборки `build.js` будет выглядеть так:

```
({
  baseUrl: 'app',
  out: 'dist/main.js',
```

Эти пути являются относительными к параметру `baseUrl` нашего проекта, и в нашем случае есть смысл указать в нем папку `app`. Параметр `out` указывает оптимизатору `r.js`, что мы хотим объединить всё в единственный файл с именем `main.js`, расположенный в директории `dist/`. Обратите внимание, что здесь необходимо добавить расширение `.js` к имени файла. В более ранних примерах у нас не было необходимости использовать это расширение при обращении к модулям по файловым именам, однако в данном случае это требуется.

В качестве альтернативы мы можем воспользоваться параметром `dir`, который позволяет указать, в какую директорию будет скопировано содержимое директории `app`. Пример:

```
({
  baseUrl: 'app',
  dir: 'release',
  out: 'dist/main.js'
```

Можно указать ряд дополнительных параметров (например, `modules` и `appDir`), которые несовместимы с параметром `out`, мы кратко опишем их на случай, если они понадобятся вам.

Параметр `modules` — это массив, где можно явно указать имена оптимизируемых модулей.

```
modules: [
  {
    name: 'app',
    exclude: [
      // если вы не хотите включать определенные
      // библиотеки, исключите их здесь
    ]
}
```

Если указывается параметр `appDir`, то параметр `baseUrl` становится относительным к нему. Если `appDir` не указан, то `baseUrl` относителен к файлу `build.js`.

```
appDir: './',
```

В нашем профиле сборки параметр `main` используется для определения главного модуля приложения; здесь мы применяем параметр `include` для того, чтобы

воспользоваться урезанным загрузчиком RequireJS-модулей Almond, который полезен при отсутствии необходимости загружать модули динамически.

```
include: ['libs/almond', 'main'],
wrap: true,
```

Параметр `include` — еще один массив, в котором указываются модули, участвующие в сборке. Когда мы указываем параметр `main`, оптимизатор `r.js` перебирает все модули, от которых зависит `main`, и включает их в сборку. Параметр `wrap` помещает модули, необходимые библиотеке RequireJS, в замыкание, обеспечивая включение в глобальное окружение только экспортируемых модулей.

```
paths: { backbone: 'libs/backbone', underscore: 'libs/underscore',
jquery: 'libs/jquery', text: 'libs/text' } })
```

В оставшейся части файла `build.js` будет находиться обычный объект, конфигурирующий пути. Мы можем скомпилировать наш проект в целевой файл командой:

```
node r.js -o build.js
```

Эта команда поместит собранный проект в файл `dist/main.js`.

Профиль сборки обычно помещают в директорию `scripts` или `js` внутри проекта. Согласно документации, этот файл может находиться в любом месте, однако вам необходимо соответствующим образом отредактировать профиль сборки.

Вот и всё. Как только вы правильно настроите инструменты UglifyJS/Closure, то сможете оптимизировать весь Backbone-проект с помощью утилиты `r.js`, которая потребует лишь нескольких нажатий клавиш.

На примере Джеймса Берка можно подробно изучить профили сборки, он содержит подробные комментарии и демонстрирует все имеющиеся параметры.

Заключение

Управление зависимостями в JavaScript может оказаться непростой задачей. Если вы решите создать приложение с модульной структурой и разобьете его на несколько файлов, то вам придется определить зависимости для каждого файла и позаботиться о том, чтобы они были загружены в правильном порядке. Без строгих правил именования легко создать хаос в глобальном пространстве имен, заполнив его своими объектами.

Технология AMD и библиотека RequireJS упрощают процесс разработки модульного приложения, предоставляя удобный синтаксис, позволяющий определять модули и их зависимости, не засоряя глобальное пространство имен. Хотя технология AMD не универсальна, она помогает структурировать код путем четкого разделения групп моделей, представлений и коллекций, формирующих область веб-страницы, на модули. Оцените, хорошо ли AMD-модули подходят для вашего приложения; если ответ положительный, то они принесут вам существенную пользу.

9

Упражнение 3: ваше первое модульное приложение на Backbone и RequireJS

В этой главе мы изучим первый реальный проект с использованием библиотек Backbone и RequireJS – разработку модульного приложения для управления задачами. Как и упражнение 1 главы 4, это приложение позволяет добавлять новые и редактировать существующие задачи, а также удалять задачи, помеченные как завершенные. Более сложный пример вы найдете в главе 12.

Полный код данного приложения находится в папке <https://github.com/bogavante/addyosmani-backbone-fundamentals/tree/master/practicals/modular-todo-app> (спасибо Томасу Дэвису (Thomas Davis) и Джерому Грэвл-Никвету (Gravel-Niquet)). Вы также можете скачать копию моего проекта TodoMVC (<https://github.com/tastejs/todomvc>), который содержит версии этого приложения как с использованием технологии AMD, так и без нее.

Введение

Написать модульное Backbone-приложение, как правило, несложно. Тем не менее, если вы собираетесь делать это с использованием AMD-модулей, следует иметь в виду несколько аспектов:

- Поскольку AMD не является встроенным форматом ни для JavaScript, ни для браузера, то для поддержки определений компонентов и модулей в формате AMD вам придется пользоваться загрузчиком сценариев (например, *RequireJS* или *curl.js*). Мы уже говорили о том, что формат AMD обладает рядом преимуществ, а библиотека *RequireJS* помогает при разработке приложений с его использованием.
- Необходимо инкапсулировать модели, представления, контроллеры и маршрутизаторы с помощью формата AMD. Это дает возможность каждому

компоненту Backbone-приложения четко управлять зависимостями (например, коллекциями, которые нужны представлению) так же, как это делают AMD-модули, разработанные без участия Backbone.

- Компоненты/модули, не входящие в состав Backbone (например, утилиты или помощники приложений), также можно инкапсулировать с помощью AMD. Я рекомендую разрабатывать эти модули так, чтобы их было можно использовать и тестировать независимо от Backbone-кода, поскольку это расширит возможность их повторного применения в других приложениях.

Итак, начнем знакомиться с разработкой приложения. Наше приложение будет иметь следующую структуру:

```
index.html
...js/
    main.js
    .../models
        todo.js
    .../views
        app.js
        todos.js
    .../collections
        todos.js
    .../templates
        stats.html
        todos.html
    .../libs
        .../backbone
        .../jquery
        .../underscore
        .../require
            require.js
            text.js
...css/
```

Разметка

Разметка приложения относительно проста и состоит из трех основных частей: секции ввода для добавления задач (`create-todo`), секции списка для отображения существующих задач, которые можно редактировать на месте (`todo-list`), и секции, в которой отображается количество незавершенных задач (`todo-stats`).

```
<div id="todoapp">
    <div class="content">
        <div id="create-todo">
            <input id="new-todo" placeholder="What needs to be done?" type="text" />
```

```
<span class="ui-tooltip-top">Press Enter to save this task</span>
</div>
<div id="todos">
  <ul id="todo-list"></ul>
</div>
<div id="todo-stats"></div>
</div>
</div>
```

Остальная часть этого руководства посвящена JavaScript-коду приложения.

Конфигурационные параметры

Если вы прочитали предыдущую главу, посвященную AMD, то, возможно, заметили, что явно задавать каждую зависимость, требуемую Backbone-модулю (представлению, коллекции и др.), — весьма трудоемкое занятие. Попробуем упростить задачу.

Для этого воспользуемся конфигурационным объектом RequireJS, который обычно определяется как файл сценария верхнего уровня.

У конфигурационных объектов есть ряд полезных возможностей, из которых наиболее ценной является *карта имен*. Карты имен, по сути, представляют собой пары «ключ-значение», где ключ задает псевдоним пути, а значение — сам путь.

В следующем примере, `main.js`, показаны типичные карты имен: `backbone`, `underscore`, `jquery` и, в зависимости от вашего выбора, RequireJS-плагин `text`, который участвует в загрузке текстовых ресурсов как шаблонов.

```
require.config({
  baseUrl: '../',
  paths: {
    jquery: 'libs/jquery/jquery-min',
    underscore: 'libs/underscore/underscore-min',
    backbone: 'libs/backbone/backbone-optamd3-min',
    text: 'libs/require/text'
  }
});
require(['views/app'], function(AppView){
  var app_view = new AppView;
});
```

Метод `require()` в конце файла `main.js` загружает начальное представление нашего приложения (`views/app.js`) и создает его экземпляр. Обычно этот метод и конфигурационный объект располагаются в файлах сценариев, расположенных на наиболее высоких уровнях проекта.

Конфигурационный объект не только позволяет задавать карты имен, но и предоставляет дополнительные свойства, такие как `waitSeconds` (тайм-аут загрузки сценария в секундах) и `locale` (на случай, если вы захотите загрузить пакеты поддержки нестандартных языков для интернационализации). Свойство `baseUrl` представляет собой путь для поиска модулей.

Более подробную информацию о конфигурационных объектах вы найдете в руководствах, имеющихся в документации библиотеки RequireJS.

Создание модулей из моделей, представлений и коллекций

Перед тем как мы углубимся в изучение Backbone-компонентов, упакованных в AMD-модули, давайте рассмотрим пример представления в формате, отличном от AMD.

Следующее представление прослушивает изменения в своей модели (задаче) и заново отображает ее, если пользователь редактирует ее содержимое:

```
var TodoView = Backbone.View.extend({
    //... тег списка.
    tagName: 'li',
    // Кэширование функции шаблона для отдельной задачи.
    template: _.template($('#item-template').html()),
    // DOM-события, специфичные для задачи
    events: {
        'click .check' : 'toggleDone',
        'dblclick div.todo-content' : 'edit',
        'click span.todo-destroy' : 'clear',
        'keypress .todo-input' : 'updateOnEnter'
    },
    // Представление TodoView прослушивает изменения своей модели
    // и повторно отображает себя. Поскольку
    // **Todo** и **TodoView** соотносятся как 1:1, для удобства
    // мы создаем прямую ссылку на модель.
    initialize: function() {
        this.model.on('change', this.render, this);
        this.model.view = this;
    },
    ...
});
```

Обратите внимание: при работе с шаблоном мы типичным образом обращаемся к сценарию по `id` (или другому селектору) и считываем его значение. Разумеется, для этого необходимо, чтобы шаблон, к которому осуществляется доступ, был неявно определен в разметке. Ниже приведена встраиваемая версия шаблона, к которому мы только что обратились:

```
<script type="text/template" id="item-template">
  <div class="todo <%= done ? 'done' : '' %>">
    <div class="display">
      <input class="check" type="checkbox" <%= done ?
        'checked="checked"' : '' %> />
      <div class="todo-content"></div>
      <span class="todo-destroy"></span>
    </div>
    <div class="edit">
      <input class="todo-input" type="text" value="" />
    </div>
  </div>
</script>
```

В этом шаблоне нет ничего неправильного, однако если мы начнем разрабатывать более крупные приложения, которым требуется множество шаблонов, то их включение в разметку при загрузке страницы быстро станет неконтролируемым и отрицательно повлияет на производительность приложения. Скоро мы узнаем, как решить эту проблему.

Теперь давайте обратимся к AMD-версии представления `views/todo.js`. Как мы отмечали ранее, этот модуль «обернут» AMD-методом `define()`, позволяющим указывать зависимости представления. Использование карты имен упрощает обращение к общим зависимостям, а экземпляры зависимостей связаны с доступными нам локальными переменными (например, к `jquery` можно обратиться через `$`).

```
define([
  'jquery',
  'underscore',
  'backbone',
  'text!templates/todos.html'
], function($, _, Backbone, todosTemplate){
  var TodoView = Backbone.View.extend({
    //... тег списка.
    tagName: 'li',
    // Кэширование функции шаблона для отдельной задачи.
    template: _.template(todosTemplate),
    // DOM-события, специфичные для задачи.
    events: {
      'click .check' : 'toggleDone',
      'dblclick div.todo-content' : 'edit',
      'click span.todo-de stroy' : 'clear',
      'keypress .todo-input' : 'updateOnEnter'
    },
    // Представление TodoView прослушивает изменения своей модели
    // и повторно отображает себя. Поскольку
    // **Todo** и **TodoView** соотносятся как 1:1, для удобства
    // мы создаем прямую ссылку на модель.
    initialize: function() {
```

продолжение ↗

```

        this.model.on('change', this.render, this);
        this.model.view = this;
    },
    // повторное отображение содержания задачи.
    render: function() {
        this.$el.html(this.template(this.model.toJSON()));
        this.setContent();
        return this;
    },
    // Воспользуйтесь `jQuery.text`, чтобы задать содержание задачи.
    setContent: function() {
        var content = this.model.get('content');
        this$('.todo-content').text(content);
        this.input = this$('.todo-input');
        this.input.on('blur', this.close);
        this.input.val(content);
    },
    ...

```

С точки зрения поддержки кода эта версия представления не имеет логических отличий за исключением работы с шаблонами.

Используя текстовый плагин библиотеки RequireJS (зависимость, помеченная как `text`), мы сохраним все содержимое шаблона, рассмотренное ранее, во внешнем файле (`templates/todos.html`).

```

<div class="todo <%= done ? 'done' : '' %>">
    <div class="display">
        <input class="check" type="checkbox" <%= done ?
            'checked="checked"' : '' %> />
        <div class="todo-content"></div>
        <span class="todo-destroy"></span>
    </div>
    <div class="edit">
        <input class="todo-input" type="text" value="" />
    </div>
</div>

```

Нет необходимости в использовании `id` шаблона, поскольку мы свяжем его содержимое с локальной переменной (в данном случае `todosTemplate`). Затем просто передадим эту переменную функции `_template()` шаблонизатора `Underscore.js` так же, как обычное значение сценария шаблона.

Теперь рассмотрим определение моделей как зависимостей, которые можно поместить в коллекции. Следующий файл `models/todo.js` представляет собой AMD-совместимый модуль с двумя значениями по умолчанию: атрибутом `content`, в котором находится содержание задачи, и логической переменной состояния `done`, позволяющей помечать задачу как завершенную или незавершенную.

```

define(['underscore', 'backbone'], function(_, Backbone) {
  var TodoModel = Backbone.Model.extend({
    // Атрибуты задачи по умолчанию.
    defaults: {
      // У каждой задачи должно быть содержание
      content: 'empty todo...',
      done: false
    },
    initialize: function() {
    },
    // Изменение состояния завершения этой задачи.
    toggle: function() {
      this.save({done: !this.get('done')});
    },
    // Удаление этой задачи из локального хранилища
    // и уничтожение ее представления
    clear: function() {
      this.destroy();
      this.view.remove();
    }
  });
  return TodoModel;
});

```

Что касается других типов зависимостей, то мы можем связать модуль нашей модели с локальной переменной (в данном случае Todo), чтобы с ее помощью включать модель в коллекцию TodosCollection. Эта коллекция, collections/todos.js, также поддерживает простой фильтр done() для отображения завершенных задач и фильтр remaining() для отображения задач, находящихся в процессе выполнения.

```

define([
  'underscore',
  'backbone',
  'libs/backbone/localstorage',
  'models/todo'
], function(_, Backbone, Store, Todo){
  var TodosCollection = Backbone.Collection.extend({
    // ссылка на модель этой коллекции.
    model: Todo,
    // сохранение всех задач в пространстве имен `todos`.
    localStorage: new Store('todos'),
    // фильтрация завершенных задач.
    done: function() {
      return this.filter(function(todo){ return todo.get('done'); });
    },
    // фильтрация незавершенных задач.
    remaining: function() {
      return this.without.apply(this, this.done());
    },
    ...
  });
}

```

Нам необходимо не только дать пользователям возможность добавлять новые задачи в представлениях (которые мы затем вставляем в коллекцию в виде моделей), но также отображать число завершенных и незавершенных задач. В предыдущей коллекции мы уже определили фильтры, которые могут представлять такую информацию, и теперь воспользуемся ими в главном представлении нашего приложения, `views/app.js`.

```
define([
  'jquery',
  'underscore',
  'backbone',
  'collections/todos',
  'views/todo',
  'text!templates/stats.html'
], function($, _, Backbone, Todos, TodoView, statsTemplate){
  var AppView = Backbone.View.extend({
    // вместо генерации нового элемента мы привязываемся к существующему
    // "скелету" приложения в формате HTML.
    el: $('#todoapp'),
    // Шаблон строки статистики в нижней части приложения.
    statsTemplate: _.template(statsTemplate),
    // ...события, initialize() и др. есть в полном файле.
    // Повторное отображение приложения означает обновление статистики;
    // остальная часть приложения остается неизменной.
    render: function() {
      var done = Todos.done().length;
      this.$('#todo-stats').html(this.statsTemplate({
        total: Todos.length,
        done: Todos.done().length,
        remaining: Todos.remaining().length
      }));
    },
    ...
  });
});
```

Здесь мы связываем второй шаблон этого проекта, `templates/stats.html`, с переменной `statsTemplate`, которая используется для отображения завершенных и незавершенных задач. Мы передаем шаблону размер коллекции задач (`Todos.length`), а также количество завершенных задач (`Todos.done().length`) и незавершенных задач (`Todos.remaining().length`).

Ниже приведено содержимое шаблона `statsTemplate`. Он не содержит ничего сложного, однако в нем используются тернарные операции, чтобы определить, один или два элемента находятся в определенном состоянии.

```
<% if (total) { %>
  <span class="todo-count">
    <span class="number"><%= remaining %></span>
    <span class="word"><%= remaining == 1 ? 'item' : 'items' %>
  </span> left.
```

```
</span>
<% } %>
<% if (done) { %>
  <span class="todo-clear">
    <a href="#">
      Clear <span class="number-done"><%= done %></span>
      completed <span class="word-done"><%= done == 1 ?
        'item' : 'items' %></span>
    </a>
  </span>
<% } %>
```

Остальная часть приложения состоит из кода, обрабатывающего события, которые генерируются пользователями и самим приложением, однако этот код обеспечивает полноту ключевых концепций данного примера.

Чтобы ознакомиться с полной версией приложения, клонируйте этот репозиторий или просмотрите его содержимое в онлайн-режиме. Надеюсь, что это будет полезно для вас.

Загрузка модулей с использованием маршрутов

В этом разделе описывается маршрутный подход к загрузке модулей, реализованный в инструменте *Lumbar*, написанном Кевином Деккером (Kevin Decker). Lumbar, как и RequireJS, является средством сборки модульных приложений, однако реализованный в нем механизм загрузки маршрутов можно использовать с любой системой сборки.

Lumbar не рассматривается подробно в данной книге. Полноценный проект на основе Lumbar, включающий загрузчика и систему сборки, можно найти в библиотеке Thorax, в которой имеются примеры проектов для различных сред, в том числе для Lumbar.

Конфигурация модуля в формате JSON

В библиотеке RequireJS зависимости определяются для каждого файла; в Lumbar для каждого модуля определяется список файлов в центральном конфигурационном файле формата JSON, и в результате сборки для каждого модуля создается единственный JavaScript-файл. Lumbar требует, чтобы в каждом модуле (за исключением базового) был определен единственный маршрутизатор и список маршрутов. Например:

```
{
  "modules": {
    "base": {
      "scripts": [
        "js/lib/underscore.js",
        "js/lib/backbone.js",
        "etc"
      ]
    },
    "pages": {
      "scripts": [
        "js/routers/pages.js",
        "js/views/pages/index.js",
        "etc"
      ],
      "routes": {
        """: "index",
        "contact": "contact"
      }
    }
  }
}
```

Каждому JavaScript-файлу, определенному в модуле, соответствует модульный объект в области видимости, содержащей имя и маршруты модуля. В файле `js/routers/pages.js` мы можем определить Backbone-маршрутизатор для нашего страничного модуля следующим образом:

```
new (Backbone.Router.extend({
  routes: module.routes,
  index: function() {},
  contact: function() {}
}));
```

Маршрутизатор загрузчика модулей

Одна из редко используемых возможностей `Backbone.Router` — создание множества маршрутизаторов, которые прослушивают один и тот же набор маршрутов. С ее помощью Lumbar создает маршрутизатор, прослушивающий все маршруты приложения. Когда маршруты совпадают, этот главный маршрутизатор проверяет, загружен ли требуемый модуль. Если модуль уже загружен, то главный маршрутизатор не выполняет никаких действий, и обработка маршрута осуществляется маршрутизатором, определенным в модуле. Если требуемый модуль не загружен, то выполняется его загрузка, а затем вызывается метод `Backbone.history.loadUrl`. Это приводит к перезагрузке маршрута, при которой главный маршрутизатор ничего не делает, а маршрутизатор, который определен в новом загруженном модуле, реагирует на нее надлежащим образом.

Пример реализации описанного механизма приведен ниже. Конфигурационный объект должен содержать данные из упомянутого выше примера конфигурационного JSON-файла, а в объекте загрузчика требуется реализовать методы `isLoaded` и `loadModule`. Обратите внимание, что инструмент Lumbar предоставляет все эти реализации; они помогут вам при создании собственного приложения.

```
// создание объекта, который будет использоваться
// как прототип для главного маршрутизатора
var handlers = {
    routes: {}
};

_.each(config.modules, function(module, moduleName) {
    if (module.routes) {
        // генерация обратного вызова, загружающего модуль
        var callbackName = "loader_" + moduleName;
        handlers[callbackName] = function() {
            if (loader.isLoaded(moduleName)) {
                // ничего не делать, если модуль загружен
                return;
            } else {
                // необходимо загрузить модуль
                loader.loadModule(moduleName, function() {
                    // модуль загружен, перезагрузка маршрута
                    // приведет к обратному вызову
                    // в маршрутизаторе модуля
                    Backbone.history.loadUrl();
                });
            }
        };
        // каждый маршрут модуля должен сгенерировать
        // обратный вызов, выполняющий загрузку
        _.each(module.routes, function(methodName, route) {
            handlers.routes[route] = callbackName;
        });
    }
});
// создание главного маршрутизатора
new (Backbone.Router.extend(handlers));
```

Обработка pushState с использованием NodeJS

Для поддержки метода `window.history.pushState` (обслуживающего маршруты Backbone без знака #) требуется, чтобы сервер обладал информацией о том, какие URL будет обрабатывать Backbone-приложение, поскольку пользователь должен иметь возможность войти в приложение с помощью любого из этих маршрутов (или нажать кнопку перезагрузки после перехода на pushState URL).

Определение всех маршрутов в одном месте имеет еще одно преимущество: представленный выше конфигурационный JSON-файл может быть загружен сервером, прослушивающим каждый маршрут. Пример реализации в *Node.js* и *Express*:

```
var fs = require('fs'),
    _ = require('underscore'),
    express = require('express'),
    server = express(),
    config = JSON.parse(fs.readFileSync('path/to/config.json'));
_.each(config.modules, function(module, moduleName) {
  if (module.routes) {
    _.each(module.routes, function(methodName, route) {
      server.get(route, function(req, res) {
        res.sendFile('public/index.html');
      });
    });
  }
});
```

Здесь предполагается, что файл `index.html` запускает ваше Backbone-приложение. Объект `Backbone.History` может заниматься обработкой остальной логики маршрутизации, если указан параметр `root`.

Вот пример конфигурации для простого приложения, находящегося в корневом каталоге:

```
Backbone.history || (Backbone.history = new Backbone.History());
Backbone.history.start({
  pushState: true,
  root: '/'
});
```

Пакеты активов — альтернатива управлению зависимостями

Для приложений, состоящих не только из простейших представлений, платформа *DocumentCloud* предлагает самодельный инструмент для пакетирования активов под названием *Jammit*, он легко интегрируется с библиотекой `Underscore.js` и может использоваться для управления зависимостями.

Jammit рассчитан на то, что ваши JavaScript-шаблоны (JST) находятся в одном месте с другими ERB-шаблонами, которые вы используете в виде `jst`-файлов. *Jammit* упаковывает эти шаблоны в глобальный JST-объект, который используется для преобразования шаблонов в строки. Обеспечить доступ *Jammit*

к вашим шаблонам просто: добавьте запись вроде `views/**/*.jst` в пакет вашего приложения в файле `assets.yml`.

Чтобы указать зависимости Jammit, выпишите содержимое файла `assets.yml`, в котором эти зависимости перечислены в надлежащем порядке либо используется комбинация факультативных каталогов (пример: `//.js`, `templates/.js` и конкретные файлы).

Шаблон, использующий инструмент Jammit, может извлекать свои данные из переданного ему объекта коллекции:

```
this.$el.html(JST.myTemplate({ collection: this.collection }));
```

10

Пагинация запросов и коллекций Backbone.js

Пагинация — это распространенная задача, которую часто приходится решать при разработке веб-приложений; более того, столкновение с ней почти неизбежно, если вы имеете дело с API служб и толстыми клиентами JavaScript, использующими эти API. Большинство разработчиков относятся к проблеме пагинации недостаточно серьезно, поскольку считают, что разобраться с ней относительно легко. Это не всегда так, и пагинация зачастую оказывается более хитроумной, чем представляется на первый взгляд.

Перед тем как приступить к изучению решений для пагинации данных в Backbone-приложениях, давайте четко определим, что же такое пагинация.

Пагинация — это система управления, которая позволяет пользователям просматривать страницы с результатами поиска или любым другим содержимым, не умещающимся на одной странице. Вывод результатов поиска — классический пример пагинации, однако на сегодняшний день она также используется на новостных сайтах, в блогах и дискуссионных клубах и часто имеет вид ссылок **Назад** (Previous) и **Вперед** (Next). Более сложные системы пагинации предоставляют пользователю дополнительные возможности управления переходами между страницами, упрощая поиск нужной информации.

Пагинация — это не только создание страниц с визуальными элементами для перехода на другие страницы; сайты вроде Facebook, Pinterest и Twitter наглядно демонстрируют полезность бесконечных страниц. Разумеется, речь идет о механизме, который выполняет (или пытается выполнить) предвыборку содержимого следующей страницы и добавляет его непосредственно на текущую страницу пользователя, создавая ощущение, что эта страница имеет бесконечную длину.

Пагинация очень сильно зависит от контекста и отображаемого содержимого. В поисковике Google пагинация важна потому, что Google стремится разместить наиболее подходящие результаты на одной-двух первых страницах. Изучив их

содержимое, вы, скорее всего, будете переходить на другие страницы выборочно или случайно. Возможны и иные ситуации, в которых необходимо просмотреть набор последовательно расположенных страниц (например, новостных статей или публикаций в блоге).

Пагинация почти целиком зависит от особенностей содержимого и контекста, однако, как ранее отметил Фарук Атес (Faruk Ates), принципы эффективной пагинации действуют независимо от содержимого и контекста. С помощью Backbone вы можете написать собственную реализацию пагинации (как и при использовании других расширяемых возможностей Backbone) и решить многие задачи, специфичные для отображаемого содержимого. Конечно, вам придется потратить на это время, но в некоторых ситуациях вы сможете воспользоваться проверенными готовыми решениями.

Мы рассмотрим набор компонентов пагинации, написанных для библиотеки *Backbone.js* мной совместно с группой программистов. Я надеюсь, что эти компоненты помогут при разработке приложений, в которых требуется реализовать постраничное отображение коллекций Backbone. Данные компоненты входят в состав расширения под названием **Backbone.Paginator**.

Backbone.Paginator

При работе с данными на клиентской стороне обычно используются три типа пагинации:

○ Отправка запросов уровню служб (API).

Например, это может быть запрос результатов, содержащих слово «*Пол*»; при наличии 5000 совпадений их требуется отобразить по 20 результатов на странице (соответственно, результаты содержатся на 250 страницах, между которыми пользователь может осуществлять переходы).

Эта задача весьма обширна и включает, например, сохранение других параметров URL (таких, как «*сортировка*», «*запрос*» и «*порядок*»), которые могут изменяться в зависимости от настроек поиска, задаваемых в пользовательском интерфейсе. Также следует продумать понятный способ привязки представлений к этой пагинации, чтобы можно было легко перемещаться между страницами (например, В начало (*First*), В конец (*Last*), Следующая (*Next*), Предыдущая (*Previous*), 1, 2, 3 и т. п.), управления количеством результатов на странице и др.

○ Дополнительная пагинация возвращенных данных на клиентской стороне.

Допустим, мы получили ответ в формате JSON, который содержит 100 результатов поиска. Вместо того чтобы отображать все 100 результатов пользователю, выведем только 20 из них и создадим в браузере пользовательский интерфейс, позволяющий перелистывать результаты.

При пагинации на клиентской стороне, как и при использовании запроса, возникает ряд задач: навигация (Следующая, Предыдущая, 1, 2, 3), сортировка, порядок, задание количества результатов на странице и т. п.

○ Бесконечные результаты.

В службах вроде Facebook вместо численной пагинации используется кнопка Загрузить еще (Load More) или Просмотреть еще (View More). При ее нажатии обычно загружается следующая страница с N результатами, которые добавляются к полученным ранее результатам, а не полностью заменяют их.

Объединение в представлении новых и старых результатов фактически формирует бесконечную «книгу».

Рассмотрим готовые решения, которые можно использовать для пагинации.

Набор *Backbone.Paginator* (рис. 10.1) содержит компоненты, осуществляющие пагинацию коллекций данных с помощью библиотеки Backbone.js. Paginator предоставляет решения для пагинации как запросов к серверу (например, API), так и отдельно загружаемых порций данных, где коллекцию из N результатов внутри нашего представления может оказаться необходимо разбить на M страниц.



Рис. 10.1. Комплект Backbone.Paginator оформляет компоненты проекта в нужном стиле

Набор Backbone.Paginator содержит два ключевых компонента пагинации:

○ Backbone.Paginator.requestPager

Предназначен для пагинации запросов между клиентом и серверным API.

○ Backbone.Paginator.clientPager

Предназначен для данных, возвращенных сервером, над которыми необходимо выполнить дополнительную пагинацию в пользовательском интерфейсе приложения (например, вы получили 60 результатов и их необходимо отобразить на трех страницах по 20 штук на странице).

Реальные примеры

Для ознакомления с примерами, в которых используются упомянутые компоненты, изучите официальные демонстрации, ссылки на которые приведены

ниже (чтобы эти примеры работали с настоящими источниками данных, воспользуйтесь Netflix API):

- Backbone.Paginator.requestPager();
- Backbone.Paginator.clientPager();
- Бесконечная пагинация (Backbone.Paginator.requestPager());
- Плагин для работы с диакритическими знаками (Diacritic).

Paginator.requestPager

В этом разделе мы познакомимся с применением компонента `requestPager` (рис. 10.2). Он используется для работы со служебным API, поддерживающим пагинацию, и позволяет пользователям управлять настройками пагинации, участвующими в запросах к этому API (например, переход на следующую страницу, предыдущую страницу или страницу с номером N) через клиентскую сторону.

The screenshot shows a Backbone.js application interface. At the top, there is a header with the title "Paginator.requestPager()". Below the header, the content area displays a list of movies. Each movie entry includes the title, a small thumbnail image, and a brief description of its runtime, release date, and rating. For example, "BMX Bandits" is described as having a runtime of 90 mins, released in 1983, with a PG rating. The descriptions mention Nicole Kidman's debut and her helping two pals become bank robbers. Another entry, "Nightmaster", has a runtime of 91 mins, released in 1986, with an R rating. The descriptions mention Robbie and Amy taking part in a simulated war game that turns them into killing machines. A third entry, "Dead Calm", has a runtime of 96 mins, released in 1989, with an R rating. The descriptions mention the Ingmans setting off on a sailing trip after their son dies tragically in a car crash, and then finding a man feebly paddling away from a sinking schooner. They bring him aboard and realize he's a murderous sociopath. At the bottom of the page, there is a navigation bar with page numbers from 1 to 18, a "Previous" and "Next" link, a "Last" link, a "Show 3 | 9 | 12 per page" dropdown, a "Page: 1 of 18 shown" indicator, and a "Sort by:" dropdown menu. There are also search and filter input fields.

Рис. 10.2. Запрос пагинированных результатов от Netflix API с помощью компонента `requestPager`

Идея заключается в том, что пагинация, поиск и фильтрация данных могут быть выполнены в Backbone-приложении без необходимости перезагружать страницу.

1. Создайте новую коллекцию для пагинации.

Мы определяем новую коллекцию для пагинации с помощью `Backbone.Paginator.requestPager()` следующим образом:

```
var PaginatedCollection = Backbone.Paginator.requestPager.extend({
```

2. Задайте модель для коллекции.

Как обычно, внутри нашей коллекции мы указываем модель, которая будет использоваться вместе с ней, а затем URL (или базовый URL) службы, предоставляющей данные (например, Netflix API).

```
model: model,
```

3. Задайте базовый URL и тип запроса.

Нам необходимо задать базовый URL. Типом запроса по умолчанию является GET, а параметр dataType установлен в значение jsonp, чтобы разрешить междоменные запросы.

```
paginator_core: {
    // тип запроса (по умолчанию GET)
    type: 'GET',
    // тип ответа (по умолчанию jsonp)
    dataType: 'jsonp',
    // URL (или базовый URL) службы
    // если вы хотите иметь более динамичный URL, то можете сделать этот элемент
    // функцией, возвращающей строку
    url: 'http://odata.netflix.com/Catalog/People(49446)/TitlesActedIn?'
},
```



Если вы задаете параметру dataType значение, отличное от jsonp, удалите параметр обратного вызова в конфигурации server_api.

4. Настройте отображение результатов библиотекой.

Необходимо указать библиотеке число элементов, отображаемых на странице, текущую страницу, диапазон номеров страниц и другие параметры.

```
paginator_ui: {
    // наименьший индекс страницы, к которому разрешает доступ ваш API
    firstPage: 0,
    // начальная страница для пагинатора
    // (а также его текущая страница)
    currentPage: 0,
    // число элементов на странице
    perPage: 3,
    // общее число запрашиваемых страниц по умолчанию
    // для случаев, когда API или служба не
    // предоставляют его нам.
    // В нашем случае по умолчанию запрашивается 10 страниц.
    totalPages: 10
},
```

5. Настройте параметры, которые будут отправлены серверу.

В большинстве случаев базового URL недостаточно, поэтому передайте серверу дополнительные параметры. Обратите внимание, что вместо констант можно использовать функции, а также ссылаться на значения, указанные в `paginator_ui`.

```
server_api: {  
    // поле запроса  
    '$filter': '',  
    // число возвращаемых элементов на один запрос/страницу  
    '$top': function() { return this.perPage },  
    // число результатов, которые запрос должен пропустить.  
    // Для Netflix API было необходимо пропустить число  
    // результатов, равное произведению номера текущей страницы  
    // и числа результатов на одной странице  
    '$skip': function() { return this.currentPage * this.perPage },  
    // поле для сортировки  
    '$orderby': 'ReleaseYear',  
    // формат, в котором будут возвращены результаты  
    '$format': 'json',  
    // нестандартные параметры  
    '$inlinecount': 'allpages',  
    '$callback': 'callback'  
},
```



Если вы используете `$callback`, удостоверьтесь, что параметр `dataType` в конфигурации `paginator_core` имеет значение `jsonp`.

6. Наконец, настройте метод `Collection.parse()` — и готово!

Последнее, что нам необходимо сделать, — это настроить метод `parse()` нашей коллекции. Мы должны вернуть фрагмент JSON-ответа, содержащий данные, которые будут включены в коллекцию, — в данном случае `response.d.results` (для Netflix API).

```
parse: function (response) {  
    // измените этот код в соответствии со структурой  
    // ваших результатов (например, d.results специфично для Netflix)  
    var tags = response.d.results;  
    // обычно this.totalPages равен response.d.__count  
    // но, поскольку этот NetFlix-запрос возвращает  
    // только общее число элементов поиска, мы выполняем деление  
    this.totalPages = Math.ceil(response.d.__count / this.perPage);  
    return tags;  
}  
});  
});
```

Обратите внимание, что в параметре `this.totalPages` задается суммарное количество страниц, возвращаемое API. С его помощью мы определяем максимальное число (результатирующих) страниц, которое можно использовать при запросе текущей/последней страницы, чтобы отобразить его в пользовательском интерфейсе. Кроме того, этот параметр позволяет решить, будет ли выполняться дальнейшая обработка запроса после нажатия, скажем, кнопки `Next` (Далее).

Вспомогательные методы

Для удобства используйте в представлениях следующие методы взаимодействия с `requestPager`:

○ `Collection.goTo(n, options)`

Переход на заданную страницу.

○ `Collection.nextPage(options)`

Переход на следующую страницу.

○ `Collection.prevPage(options)`

Переход на предыдущую страницу.

○ `Collection.howManyPer(n)`

Задание числа элементов, отображаемых на странице.

Методы `.goTo()`, `.nextPage()` и `.prevPage()` коллекции `requestPager` являются расширениями исходных методов `Backbone Collection.fetch()`. По этой причине все они могут принимать объект параметров в качестве аргумента. Этот объект `option` использует параметры `success` и `error` для передачи функции, которая должна быть выполнена после получения ответа от сервера.

```
Collection.goTo(n, {
  success: function( collection, response ) {
    // вызывается, если запрос к серверу успешен
  },
  error: function( collection, response ) {
    // вызывается, если запрос к серверу неуспешен
  }
});
```

Чтобы организовать обратный вызов, воспользуйтесь объектом `jqXHR`, возвращаемым этими методами:

```
Collection
  .requestNextPage()
  .done(function( data, textStatus, jqXHR ) {
```

```

        // вызывается, если запрос к серверу успешен
    })
.fail(function( data, textStatus, jqXHR ) {
    // вызывается, если запрос к серверу неуспешен
})
.always(function( data, textStatus, jqXHR ) {
    // действия после завершения запроса к серверу
});
});

```

Если вы хотите добавить принятые модели в текущую коллекцию, а не заменить ее содержимое, передайте в эти методы аргумент {update: true, remove: false}.

```
Collection.prevPage({ update: true, remove: false });
```

Paginator.clientPager

Объект clientPager (рис. 10.3) используется для дополнительной пагинации данных, которые уже возвращены служебным API. Допустим, вы запросили у службы 100 результатов и хотите разбить их на клиентской стороне на пять страниц, каждая из которых содержит 20 результатов; объект clientPager позволяет сделать это очень легко.

The screenshot shows a user interface for paginating movie data from the Netflix API. At the top, there's a header: "Paginator.clientPager() Paginating data at the UI layer". Below it, the title "NetFlix movies" is displayed. The first movie listed is "Natural Wonders of America", with its thumbnail, meta-data (Runtime: 210 mins, Released: 1998, Rating: NR), and a short description: "A look at the natural wonders of America". The second movie listed is "An American Affair", with its thumbnail, meta-data (Runtime: 92 mins, Released: 1997, Rating: NR), and a description: "A lust for political power -- and best friends Barbara and Genevieve -- leads overly ambitious District Attorney Sam Brady down a perilous path in this taut psychological thriller helmed by Sebastian Shaw.". The third movie listed is "American Pie", with its thumbnail, meta-data (Runtime: 96 mins, Released: 1999, Rating: UR), and a description: "This smash-hit comedy follows four high school seniors as they strive for the most eagerly anticipated rite of adulthood: losing one's virginity." At the bottom of the page, there's a navigation bar with links for "1 2 3 4 5 6 7 Next Last" and a dropdown menu for "Select a field to sort on" with options "Sort (Asc)" and "Sort (Desc)". There's also a "Show 3 | 9 | 12" button and a "Select a field to filter on" input field with a "Filter" button.

Рис. 10.3. Дополнительная пагинация результатов, возвращенных Netflix API, с помощью объекта clientPager

Используйте объект clientPager, если вы предпочитаете загружать результаты единовременно и тем самым исключать дополнительные сетевые запросы каждый раз, когда пользователи хотят вывести следующую страницу с элементами. Так как вы уже получили все результаты, дальнейшие действия сводятся лишь к переключению между диапазонами данных, отображаемых пользователю.

1. Создайте новую коллекцию для пагинации с моделью и URL.

Как и в случае с requestPager, сначала мы создадим новую коллекцию Backbone.Paginator.clientPager для пагинации с использованием модели:

```
var PaginatedCollection = Backbone.Paginator.clientPager.extend({  
  model: model,  
});
```

2. Задайте базовый URL и тип запроса.

Нам необходимо задать базовый URL. Типом запроса по умолчанию является GET, а параметр dataType установлен в значение jsonp, чтобы разрешить междоменные запросы.

```
paginator_core: {  
  // тип запроса (по умолчанию GET)  
  type: 'GET',  
  // тип ответа (по умолчанию jsonp)  
  dataType: 'jsonp',  
  // URL (или базовый URL) службы  
  url: 'http://odata.netflix.com/v2/Catalog/Titles?&'  
},
```

3. Настройте отображение результатов библиотекой.

Укажите библиотеке число элементов, отображаемых на странице, текущую страницу, диапазон номеров страниц и другие параметры.

```
paginator_ui: {  
  // наименьший индекс страницы, к которому разрешает доступ ваш API  
  firstPage: 1,  
  // начальная страница для пагинатора  
  // (а также его текущая страница)  
  currentPage: 1,  
  // число элементов на странице  
  perPage: 3,  
  // общее число запрашиваемых страниц по умолчанию  
  // для случаев, когда API или служба не  
  // предоставляют его нам.  
  // В нашем случае по умолчанию запрашивается 10 страниц.  
  totalPages: 10,  
  // Общее число страниц, отображаемых в виде списка пагинации,  
  // составляет (pagesInRange * 2) + 1.  
  pagesInRange: 4  
},
```

4. Настройте параметры, которые будут отправлены серверу.

В большинстве случаев базового URL недостаточно, поэтому передайте серверу дополнительные параметры. Обратите внимание, что вместо констант можно использовать функции, а также ссылаться на значения, указанные в `paginator_ui`.

```
server_api: {
    // поле запроса
    '$filter': 'substringof(\'america\',Name)',
    // число возвращаемых элементов на один запрос/страницу
    '$top': function() { return this.perPage },
    // число результатов, которые запрос должен пропустить.
    // Для Netflix API было необходимо пропустить число
    // результатов, равное произведению номера текущей страницы
    // и числа результатов на одной странице
    '$skip': function() { return this.currentPage * this.perPage },
    // поле для сортировки
    '$orderby': 'ReleaseYear',
    Paginator.clientPager | 215
    // формат, в котором будут возвращены результаты
    '$format': 'json',
    // нестандартные параметры
    '$inlinecount': 'allpages',
    '$callback': 'callback'
},
```

5. Наконец, настройте метод `Collection.parse()` — и все готово!

Последнее, что нам нужно сделать — настроить метод `parse()`, который не учитывает общее число страниц результатов, переданных сервером, поскольку мы используем свой собственный счетчик страниц с данными, формируемыми в пользовательском интерфейсе при пагинации.

```
parse: function (response) {
    var tags = response.d.results;
    return tags;
}
});
```

Вспомогательные методы

Как уже говорилось, ваши представления могут использовать ряд вспомогательных методов для навигации по данным, разбитым на страницы в пользовательском интерфейсе. Для объекта `clientPager` эти методы следующие:

○ `Collection.goTo(n, options)`

Переход на заданную страницу.

○ `Collection.prevPage(options)`

Переход на предыдущую страницу.

○ `Collection.nextPage(options)`

Переход на следующую страницу.

○ `Collection.howManyPer(n)`

Задание числа элементов, отображаемых на странице.

○ `Collection.setSort(sortBy, sortDirection)`

Обновление режима сортировки в текущем представлении. Процедура сортировки автоматически определяет, пытаетесь ли вы упорядочить числа (даже если они хранятся в виде строк), и корректно сортирует данные.

○ `Collection.setFilter(filterFields, filterWords)`

Фильтрация текущего представления. Фильтрация поддерживает множественные слова и не учитывает их порядок, поэтому в вашем распоряжении фактически есть функция полнотекстового поиска. Вы можете передавать этому методу как одно поле модели, так и массив полей, над каждым из которых выполняется фильтрация. Последний передаваемый параметр — это объект, содержащий метод и правила сравнения. В настоящее время поддерживается только метод Левенштейна (Levenshtein). Расстояние Левенштейна — это разность между двумя строками, она фактически представляет собой минимальное количество изменений, которое нужно выполнить, чтобы превратить одно слово в другое.

Функции `goTo()`, `prevPage()` и `nextPage()` не требуют использования параметра `options`, поскольку выполняются синхронно. Тем не менее если вы указываете обратный вызов, который запускается при успешном выполнении какой-либо из этих функций, то он активизируется раньше, чем функция завершает работу. Например:

```
nextPage(); // это просто работает!
nextPage({success: function() { }}); // здесь при успешном выполнении метода
// вызывается "функция успеха"
```

Параметр `options` обеспечивает некоторую унификацию интерфейсов объектов `requestPaginator` и `clientPaginator`, что позволяет взаимозаменять использовать их в представлениях типа `Backbone.View`.

```
this.collection.setFilter(
  {'Name': {cmp_method: 'levenshtein', max_distance: 7}}
  , "American P" // Обратите внимание на то, что буквы 'r' и 'e' меняются
    // местами, и на букву 'P' из слова 'Pie'
);
```

Также обратите внимание, что необходимо загрузить и включить плагин с методом Левенштейна с помощью переменной `useLevenshteinPlugin`. Не менее важна еще одна деталь: сравнение Левенштейна возвращает разность двух строк и не позволяет осуществлять *поиск* в длинном тексте.

Разность двух строк — это число символов, которые необходимо добавить, удалить или сдвинуть вправо/влево для того, чтобы строки стали одинаковыми. Разница строк «Something» и «This is a test that could show something» равна 32; это больше, чем разница между строками «Something» и «ABCDEFG», которая равна 9. Применяйте метод Левенштейна только к коротким текстам, таким как заголовки, имена и т. п.

○ `Collection.doFakeFilter(filterFields, filterWords)`

Возвращает число моделей после ложного применения метода `Collection.setFilter`.

○ `Collection.setFieldFilter(rules)`

Фильтрация всех значений всех моделей в соответствии с правилами, указанными в аргументе `rules`.

Предположим, что у вас имеется коллекция книг, у каждой из которых есть год издания и автор. Вы можете сначала отфильтровать только книги, выпущенные между 1999 и 2003 годами, а затем добавить правило, которое фильтрует книги, имя автора которых начинается на букву «A». Поддерживаются следующие правила: `function`, `required`, `min`, `max`, `range`, `minLength`, `maxLength`, `rangeLength`, `oneOf`, `equalTo`, `containsAllOf`, `pattern`. Передача этому методу пустого набора правил удаляет примененные правила `FieldFilter`.

```
my_collection.setFieldFilter([
  {field: 'release_year', type: 'range', value:
    {min: '1999', max: '2003'}},
  {field: 'author', type: 'pattern', value: new RegExp('A*', 'igm')}
]);
//Правила:
//
//var my_var = 'green';
//
//{{field: 'color', type: 'equalTo', value: my_var}
//{{field: 'color', type: 'function', value: function(field_value){
return field_value == my_var; } }
//{{field: 'color', type: 'required'}
//{{field: 'number_of_colors', type: 'min', value: '2'}
//{{field: 'number_of_colors', type: 'max', value: '4'}
//{{field: 'number_of_colors', type: 'range', value: {min: '2', max: '4'} }
//{{field: 'color_name', type: 'minLength', value: '4'}
//{{field: 'color_name', type: 'maxLength', value: '6'}
//{{field: 'color_name', type: 'rangeLength', value: {min: '4', max: '6'} }
//{{field: 'color_name', type: 'oneOf', value: ['green', 'yellow']}}
```

продолжение ↗

```
//{field: 'color_name', type: 'pattern', value: new RegExp('gre*', 'ig')}
//{field: 'color_name', type: 'containsAllOf', value:
['green', 'yellow', 'blue']}
```

○ **Collection.doFakeFieldFilter(rules)**

Возвращает число моделей после ложного применения метода `Collection.setFieldFilter`.

Замечания о реализации

Для отражения текущего состояния пагинатора вы можете использовать в своем представлении ряд переменных:

○ **totalUnfilteredRecords**

Содержит количество записей, считая записи, фильтрованные любым из способов (доступна только в `clientPager`).

○ **totalRecords**

Содержит число записей.

○ **currentPage**

Текущая страница, на которой находится пагинатор.

○ **perPage**

Число записей, отображаемых пагинатором на одной странице.

○ **totalPages**

Общее число страниц.

○ **startRecord**

Позиция первой записи, отображенной на текущей странице: например, от 41 до 50 из 2000 записей (доступна только в `clientPager`).

○ **endRecord**

Позиция последней записи, отображенной на текущей странице: например, от 41 до 50 из 2000 записей (доступна только в `clientPager`).

○ **pagesInRange**

Число страниц, которые должны быть отображены на каждой стороне текущей страницы. Если `pagesInRange` имеет значение 3, а `currentPage` — значение 13, то вы получите номера 10, 11, 12, 13 (выбран), 14, 15, 16.

```
<!--пример шаблона для пользовательского интерфейса с пагинацией -->
<script type="text/html" id="tmpServerPagination">
  <div class="row-fluid">
```

```
<div class="pagination span8">
  <ul>
    <% _.each (pageSet, function (p) { %>
      <% if (currentPage == p) { %>
        <li class="active"><span><%= p %></span></li>
      <% } else { %>
        <li><a href="#" class="page"><%= p %></a></li>
      <% } %>
    <% }); %>
  </ul>
</div>
<div class="pagination span4">
  <ul>
    <% if (currentPage > firstPage) { %>
      <li><a href="#" class="serverprevious">Previous</a></li>
    <% } else{ %>
      <li><span>Previous</span></li>
    <% } %>
    <% if (currentPage < totalPages) { %>
      <li><a href="#" class="servernext">Next</a></li>
    <% } else { %>
      <li><span>Next</span></li>
    <% } %>
    <% if (firstPage != currentPage) { %>
      <li><a href="#" class="serverfirst">First</a></li>
    <% } else { %>
      <li><span>First</span></li>
    <% } %>
    <% if (totalPages != currentPage) { %>
      <li><a href="#" class="serverlast">Last</a></li>
    <% } else { %>
      <li><span>Last</span></li>
    <% } %>
  </ul>
</div>
<span class="cell serverhowmany"> Show <a href="#" class="selected">18</a> | <a href="#" class="">9</a> | <a href="#" class="">12</a> per page
</span>
<span class="divider">/</span>
<span class="cell first records">
  Page: <span class="label"><%= currentPage %></span> of
  <span class="label"><%= totalPages %></span> shown
</span>
</script>
```

Плагины

Плагин *Diacritic.js* для Backbone.Paginator заменяет диакритические символы (‘, ”, ~ и др.) на символы, наиболее похожие на них, как показано на рис. 10.4. Это особенно полезно для фильтрации.

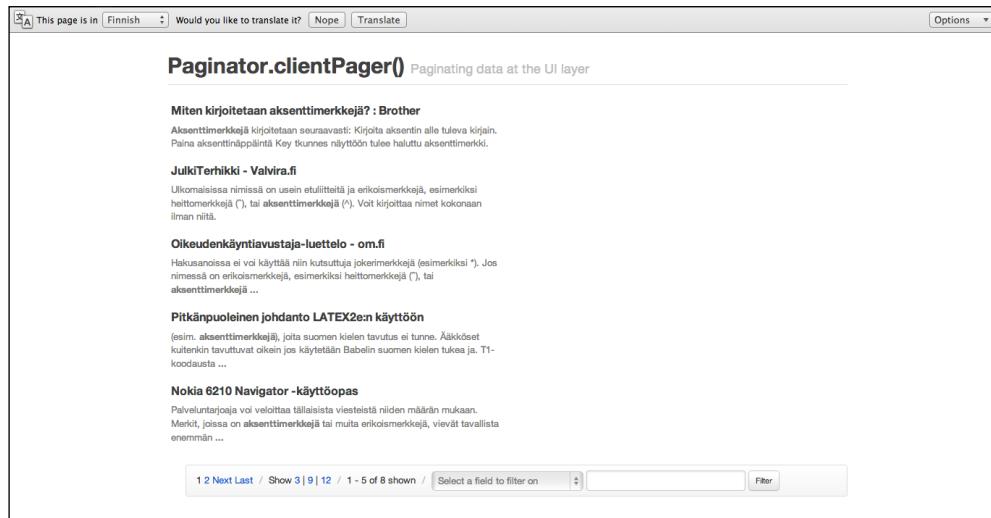


Рис. 10.4. Плагин Diacritic используется для корректного отображения специальных символов с помощью clientPager

Чтобы включить этот плагин, установите переменную `this.useDiacriticsPlugin` в значение `true`, как показано в этом примере:

```
Paginator.clientPager = Backbone.Collection.extend({
  // значения по умолчанию, используемые при сортировке и/или фильтрации
  initialize: function(){
    this.useDiacriticsPlugin = true; // использовать плагин для работы
                                    // с диакритическими знаками, если возможно
  ...
});
```

Инициализация

По умолчанию `clientPager` и `requestPager` посылают серверу начальный запрос, чтобы заполнить свои внутренние данные о страницах. Если вы хотите избежать этого дополнительного запроса, то экземпляр `Backbone.Paginator` можно инициализировать данными, которые уже есть в DOM, как показано ниже в `Backbone.Paginator.clientPager`.

```
// Расширение Backbone.Paginator.clientPager с помощью ваших
// собственных конфигурационных параметров
var MyClientPager = Backbone.Paginator.clientPager.extend({paginator_ui: {}});
// создание экземпляра вашего класса и его заполнение
// всеми моделями вашей коллекции
```

```
var aClientPager = new MyClientPager([{id: 1, title: 'foo'},
{id: 2, title: 'bar'}]);
// Вызов функции инициализации
aClientPager.bootstrap();
```



Если вы хотите инициализировать clientPager, то нет необходимости указывать объект paginator_core в конфигурации (поскольку вы уже должны были заполнить объект clientPager всеми необходимыми данными), как показано ниже в Backbone.Paginator.requestPager.

```
// Расширение Backbone.Paginator.requestPager с помощью ваших собственных
// конфигурационных параметров
var MyRequestPager = Backbone.Paginator.requestPager.extend({paginator_ui: {}});
// Создание экземпляра вашего класса с первой страницей данных
var aRequestPager = new MyRequestPager([{id: 1, title: 'foo'},
{id: 2, title: 'bar'}]);
// Вызов функции инициализации и конфигурирование requestPager
// с использованием 'totalRecords'
aRequestPager.bootstrap({totalRecords: 50});
```



Функции инициализации объектов clientPager и requestPager принимают параметр options, который будет расширен с помощью экземпляра Backbone.Paginator. Свойство totalRecords будет задано неявно объектом clientPager.

Более подробную информацию об инициализации Backbone вы найдете на веб-сайте Рико СтаКруза (Rico Sta Cruz).

Стили

Разумеется, вы можете настраивать общий вид пагинаторов по своему усмотрению. По умолчанию во всех приложениях-примерах для стилизации ссылок, кнопок и выпадающих меню используется *Twitter Bootstrap*.

Классы CSS позволяют задавать стили для количества записей, фильтров, сортировки и других элементов, как показано на рис. 10.5.

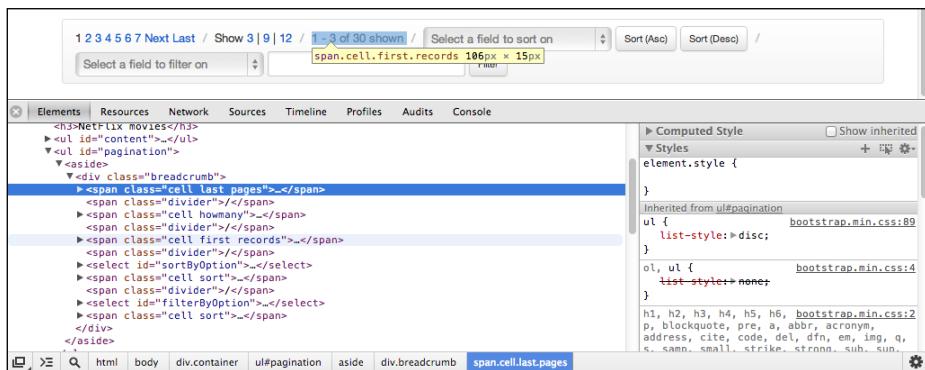


Рис. 10.5. Изучение пагинатора с помощью инструментальной консоли Chrome дает информацию о поддержке стилей в некоторых его классах

С помощью классов можно задавать стили и для более мелких элементов, таких как счетчик страниц, внутри breadcrumb > pages (например, .page, .page selected), как показано на рис. 10.6.

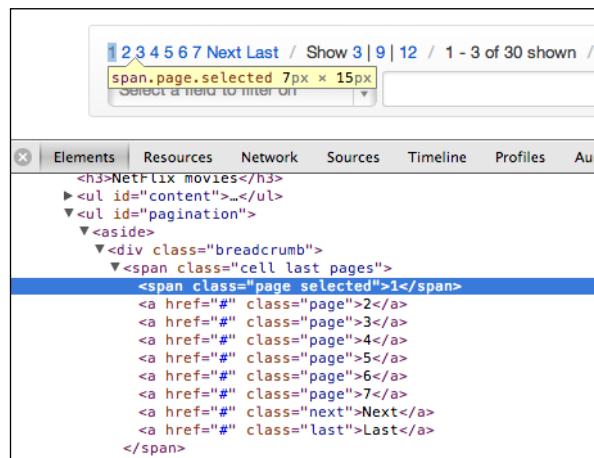


Рис. 10.6. Задание стиля навигационной строки с помощью Twitter Bootstrap

Возможности стилизации очень гибки, и если вы хорошо владеете шаблонами, то можете придать вашим пагинаторам желаемый внешний вид — от простого до сложного.

Заключение

Вы, конечно, можете самостоятельно создать классы для пагинации, работающие с коллекциями Backbone, но значительная часть этой работы уже проделана за вас в наборе Backbone.Paginator.

Backbone.Paginator обладает богатыми возможностями конфигурации, что избавляет от необходимости создавать свой собственный механизм постраничного представления коллекций данных, получаемых из базы данных или с помощью API. Используйте этот плагин для преобразования больших объемов данных в более управляемые и легкодоступные страничные списки.

Если у вас возникли вопросы или предложения об улучшении Backbone.Paginator, напишите об этом в службу поддержки проекта.

11

Backbone Boilerplate и Grunt-BBB

Стандартные заготовки кода являются отправной точкой при работе над проектами. Они представляют собой минимальный необходимый фундамент, на котором строится единый функциональный набор. При разработке нового Backbone-приложения создать новую модель, как правило, можно при помощи нескольких строк кода.

Конечно, одной модели недостаточно, поскольку вам также потребуется коллекция для группировки моделей, представление для их отображения и, возможно, маршрутизатор, если вы захотите создать фиксированные URL для некоторых представлений данных из ваших коллекций. Если вы начинаете работать над совершенно новым проектом, то вам также может понадобиться процесс сборки: он генерирует оптимизированную версию вашего приложения, которую можно поместить в производственную среду.

Для выполнения этих задач полезны стандартные решения. Вместо того чтобы вручную создавать начальный код каждого Backbone-приложения, воспользуйтесь готовым клише, которое также позаботится и о процессе сборки приложения.

Именно этим занимается Backbone Boilerplate (или просто BB) – великолепный набор полезных наработок и утилит для сборки приложений на основе Backbone.js, созданный одним из разработчиков Backbone Тимом Браньеном (Tim Branyen). Тим создал BB на основе собственного опыта, с учетом всех ошибок, «ловушек» и типовых задач, с которыми ему пришлось столкнуться при активной разработке приложений с помощью Backbone.

Grunt-BBB (Boilerplate Build Buddy, помощник в сборке клише) – это инструмент, дополняющий BB и обеспечивающий возможности скаффолдинга, просмотра файлов и сборки. Grunt-BBB используется совместно с BB и позволяет быстро запускать новые Backbone-приложения (рис. 11.1).

```

addyo at addyo-macbookair3 in ~/projects
$ bbb

Boilerplate Build Buddy
bbb Version - 0.2.0-alpha-5

Usage: bbb <task_name>

clean      Clear files and folders.
concat     Concatenate files.
copy       Copy files.
debug      Alias for "clean lint jst requirejs concat styles" tasks.
handlebars Compile handlebars templates and partials.
init       Generate project scaffolding from a predefined template.
jslint    Install, ls, remove, upgrade, search, or rebuild packages.
jasmine   Run Jasmine specs in a headless PhantomJS instance.
jslinter  Compile underscore templates to JS file.
less      Compile LESS files to CSS.
lint      Validate files with JSHint.
list      Show module dependencies.
min      Minify files with UglifyJS.
mincss   Minify CSS files.
qunit    Run QUnit unit tests in a headless PhantomJS instance.
release  Alias for "debug min mincss" tasks.
repl     Run app through REPL.
requirejs Build a RequireJS project.
server   Run development server.
styles   Compile project styles.
stylus   Compile Stylus files into CSS.
targethtml Produces html_output depending on grunt release version.
test     Run unit tests with nodeunit.
watch   Run predefined tasks whenever watched files change.

```

Рис. 11.1. Инструмент Grunt-BBB, работающий в командной строке

ВВ и Grunt-BBB содержат следующие штатные возможности:

- Backbone; Lo-Dash — альтернатива Underscore.js; jQuery с Boilerplate на основе HTML5.
- Поддержка кодовых клише и скаффолдинга, позволяющая разработчику за минимальное время создавать типовые коды модулей, коллекций и других компонентов.
- Инструмент сборки, выполняющий предкомпиляцию шаблонов, конкатенацию и минимизацию библиотек, кода приложения и таблиц стилей.
- Легковесный веб-сервер Node.js.

Процесс сборки включает в себя следующие шаги:

- Предкомпиляция шаблона. Обычно использование шаблонизатора, такого как Underscore (работающего с микрошаблонами) или Handlebars.js, состоит из трех этапов: (1) чтения исходного шаблона, (2) его компиляции в JavaScript-функцию и (3) запуска скомпилированного шаблона, заполненного нужными вам данными. Предкомпиляция исключает из процесса выполнения приложения второй шаг и переносит его в шаг сборки.
- Конкатенация — это объединение множества ресурсов (в нашем случае файлов сценариев) в меньшее число файлов (или в единственный файл) с целью сокращения числа HTTP-запросов, необходимых для их получения.

- Минимизация — это процесс удаления из кода лишних символов (например, пробелов, переходов на следующую строку, комментариев) и сжатие кода для сокращения размера сценариев.

Начало работы

Для начала установим инструмент Grunt-BBB, включающий Backbone Boilerplate и необходимые сторонние зависимости, такие как утилита сборки Grunt.

Мы можем установить Grunt-BBB с помощью прм следующей командой:

```
npm install -g bbb
```

Вот и всё! Можете начинать работу.

Ниже приведена типичная процедура работы с Grunt-BBB, которой мы воспользуемся позже:

1. Инициализируйте новый проект (`bbb init`).
2. Добавьте новые модули и шаблоны (`bbb init:module`).
3. Предварительно изучите изменения с помощью встроенного сервера (`bbb server`.)
4. Запустите инструмент сборки (`bbb build`).
5. Скомпонуйте JavaScript, скомпилируйте шаблоны, соберите ваше приложение с использованием r.js и минимизируйте CSS и JavaScript (с помощью `bbb release`).

Создание нового проекта

Теперь давайте создадим новый каталог для нашего проекта и запустим команду `bbb init`, чтобы сгенерировать его начальное содержимое. Как показано ниже, будет создан ряд подкаталогов и файлов:

```
$ bbb init
Running "init" task
This task will create one or more files in the current directory, based on the
environment and the answers to a few questions. Note that answering "?" to any
question will show question-specific help and answering "none" to most questions
will leave its value blank.
"bbb" template notes:
This tool will help you install, configure, build, and maintain your Backbone
Boilerplate project.
```

```
Writing app/app.js...OK
Writing app/config.js...OK
Writing app/main.js...OK
Writing app/router.js...OK
Writing app/styles/index.css...OK
Writing favicon.ico...OK
Writing grunt.js...OK
Writing index.html...OK
Writing package.json...OK
Writing readme.md...OK
Writing test/jasmine/index.html...OK
Writing test/jasmine/spec/example.js...OK
Writing test/jasmine/vendor/jasmine-html.js...OK
Writing test/jasmine/vendor/jasmine.css...OK
Writing test/jasmine/vendor/jasmine.js...OK
Writing test/jasmine/vendor/jasmine_favicon.png...OK
Writing test/jasmine/vendor/MIT.LICENSE...OK
Writing test/qunit/index.html...OK
Writing test/qunit/tests/example.js...OK
Writing test/qunit/vendor/qunit.css...OK
Writing test/qunit/vendor/qunit.js...OK
Writing vendor/h5bp/css/main.css...OK
Writing vendor/h5bp/css/normalize.css...OK
Writing vendor/jam/backbone/backbone.js...OK
Writing vendor/jam/backbone/package.json...OK
Writing vendor/jam/backbone.layoutmanager/backbone.layoutmanager.js...OK
Writing vendor/jam/backbone.layoutmanager/package.json...OK
Writing vendor/jam/jquery/jquery.js...OK
Writing vendor/jam/jquery/package.json...OK
Writing vendor/jam/lodash/lodash.js...OK
Writing vendor/jam/lodash/lodash.min.js...OK
Writing vendor/jam/lodash/lodash.underscore.min.js...OK
Writing vendor/jam/lodash/package.json...OK
Writing vendor/jam/require.config.js...OK
Writing vendor/jam/require.js...OK
Writing vendor/js/libs/almond.js...OK
Writing vendor/js/libs/require.js...OK
Initialized from template "bbb".
Done, without errors.
```

А теперь изучим то, что получилось в результате генерации.

index.html

Это вполне стандартное урезанное клише на HTML5; единственное, на что следует обратить внимание, — это включение библиотеки RequireJS в конце листинга.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
```

продолжение ↗

```

<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<meta name="viewport" content="width=device-width,initial-scale=1">
<title>Backbone Boilerplate</title>
<!-- Стили приложения. -->
<!--(if target dummy)><!-->
<link rel="stylesheet" href="/app/styles/index.css">
<!--<!(endif)-->
</head>
<body>
    <!-- Контейнер приложения. -->
    <main role="main" id="main"></main>
    <!-- Исходный код приложения. -->
    <!--(if target dummy)><!-->
    <script data-main="/app/config" src="/vendor/js/libs/require.js"></script>
    <!--<!(endif)-->
</body>
</html>

```

RequireJS, AMD-модуль и загрузчик сценариев помогут управлять модулями приложения. Мы уже рассмотрели их в главе 10, но сейчас кратко перечислим, что происходит в данном клише:

```
<script data-main="/app/config" src="/vendor/js/libs/require.js"></script>
```

Атрибут `data-main` дает библиотеке RequireJS указание загрузить файл `app/config.js` (конфигурационный объект) после того, как завершится загрузка самой библиотеки. Обратите внимание, что здесь мы опустили расширение `.js`, поскольку библиотека RequireJS может добавить его автоматически; но, если вы укажете это расширение, ошибки не произойдет. Теперь давайте посмотрим на то, как происходит обращение к конфигурационному файлу.

config.js

Конфигурационный объект RequireJS позволяет указывать псевдонимы и пути для зависимостей, к которым предполагается частое обращение (например, `jQuery`), начальные параметры, такие как базовый URL приложения, и библиотеки-адаптеры, необеспечивающие штатной поддержки AMD. Конфигурационный файл в Backbone Boilerplate выглядит следующим образом:

```

// задайте конфигурации require.js для вашего приложения.
require.config({
    // инициализируйте приложение, указав его главный файл и JamJS
    // сгенерированный конфигурационный файл.
    deps: ["../vendor/jam/require.config", "main"],
    paths: {
        // Здесь укажите пути.
    },
},

```

```
shim: {  
  // Здесь укажите адаптеры.  
}  
});
```

Первый параметр, определенный в этом конфигурационном файле, — `deps: ["../vendor/jam/require.config", "main"]`. Он указывает библиотеке RequireJS загрузить дополнительную конфигурацию, а также файл `main.js`, который считается точкой входа в наше приложение.

Обратите внимание, что мы не указали никаких других путей для `main`. Библиотека RequireJS определит значение по умолчанию для параметра `baseUrl`, используя путь из атрибута `data-main` в файле `index.html`. Другими словами, наш параметр `baseUrl` имеет значение `app/`, и остальные необходимые сценарии будут загружаться относительно этого каталога. Если бы мы захотели использовать другое местоположение, то могли бы переопределить его, задав соответствующее значение параметра `baseUrl`.

С помощью следующего блока, `paths`, мы укажем пути относительно `baseUrl`, а также пути/ псевдонимы для зависимостей, к которым собираемся часто обращаться.

Затем следует адаптер — важная часть конфигурации RequireJS, она позволяет загружать библиотеки, не совместимые с AMD. Основная идея в том, что `shim` избавляет нас от сложной работы по реализации поддержки AMD во всех используемых нами библиотеках.

Вернемся к параметру `deps`; содержимое файла `require.config` выглядит следующим образом:

```
var jam = {  
  "packages": [  
    {  
      "name": "backbone",  
      "location": "../vendor/jam/backbone",  
      "main": "backbone.js"  
    },  
    {  
      "name": "backbone.layoutmanager",  
      "location": "../vendor/jam/backbone.layoutmanager",  
      "main": "backbone.layoutmanager.js"  
    },  
    {  
      "name": "jquery",  
      "location": "../vendor/jam/jquery",  
      "main": "jquery.js"  
    },  
    {  
      "name": "lodash",  
      "location": "../vendor/jam/lodash",  
      "main": "lodash.js"  
    }  
  ],  
  "shim": {  
    // Здесь укажите адаптеры.  
  }  
};
```

продолжение ➔

```

        "location": "../vendor/jam/lodash",
        "main": "./lodash.js"
    }
],
"version": "0.2.11",
"shim": {
    "backbone": {
        "deps": [
            "jquery",
            "lodash"
        ],
        "exports": "Backbone"
    },
    "backbone.layoutmanager": {
        "deps": [
            "jquery",
            "backbone",
            "lodash"
        ],
        "exports": "Backbone.LayoutManager"
    }
}
);

```

Объект `jam` конфигурирует менеджер пакетов *Jam*, который предназначен для внешних интерфейсов и помогает устанавливать, обновлять и настраивать зависимости вашего проекта. В настоящее время Backbone Boilerplate использует Jam для управления пакетами.

Массив `packages` содержит набор зависимостей, которые необходимо включить в проект, в том числе Backbone, плагин `Backbone.LayoutManager`, `jQuery` и `Lo-Dash`.

`Backbone.LayoutManager` — это плагин для библиотеки Backbone, который предоставляет базовые возможности для сборки макетов и представлений внутри Backbone.

Для установки дополнительных пакетов с помощью Jam необходимо добавить соответствующие записи в параметр `packages`.

main.js

Следующий файл, `main.js`, определяет точку входа в наше приложение. Мы используем глобальный метод `require()`, чтобы загрузить массив, содержащий другие необходимые сценарии, например наше приложение `app.js` и главный маршрутизатор `router.js`. Обратите внимание, что мы обычно используем метод `require()` только для инициализации приложения, а во всех остальных случаях пользуемся аналогичным методом `define()`.

Функция, определенная после нашего массива зависимостей, — это обратный вызов, который не срабатывает до тех пор, пока перечисленные сценарии не будут загружены. Обратите внимание, что для удобства мы можем создать локальные псевдонимы для приложения и маршрутизатора — соответственно `app` и `Router`.

```
require([
    // Приложение.
    "app",
    // Главный маршрутизатор.
    "router"
],
function(app, Router) {
    // Определение главного маршрутизатора в пространстве имен приложения
    // и запуск всей навигации из этого экземпляра.
    app.router = new Router();
    // Запуск начального маршрута и включение поддержки HTML5 History API,
    // задание '/' в качестве корневого каталога по умолчанию.
    // Измените его в app.js.
    Backbone.history.start({ pushState: true, root: app.root });
    // Вся относительная навигация должна передаваться методу navigate
    // для обработки маршрутизатором. Если у ссылки есть атрибут `data-bypass`,
    // то полностью исключите делегирование.
    $(document).on("click", "a[href]:not([data-bypass])", function(evt) {
        // Получение абсолютного href якоря.
        var href = { prop: $(this).prop("href"), attr: $(this).attr("href") };
        // Получение абсолютного корня.
        var root = location.protocol + "//" + location.host + app.root;
        // Корень должен входить в href якоря, указывая на его относительность.
        if (href.prop.slice(0, root.length) === root) {
            // подавление события по умолчанию, чтобы ссылка
            // не приводила к обновлению страницы.
            evt.preventDefault();
            // `Backbone.history.navigate` достаточен для всех маршрутизаторов
            // и генерирует корректные события. Внутренний метод
            // `navigate` маршрутизатора
            // всегда вызывает его. Этот фрагмент взят из корня.
            Backbone.history.navigate(href.attr, true);
        }
    });
});
```

Backbone Boilerplate содержит встроенный стандартный код, активизирующий поддержку HTML5 History API при инициализации нашего маршрутизатора и обрабатывающий другие действия при навигации; по этой причине нам не нужно реализовать эти функции самостоятельно.

app.js

Теперь рассмотрим модуль `app.js`. Обычно в приложениях, не использующих Backbone Boilerplate, файл `app.js` содержит ключевую логику или ссылки на модули, необходимые для запуска приложения.

В данном примере этот файл содержит определения конфигурационных параметров шаблонов и макетов, а также утилиты для использования макетов. Код такого объема может показаться чересчур громоздким, однако хорошая новость заключается в том, что для простых приложений этот код, скорее всего, не потребует существенной модификации. Вам придется в большей степени сконцентрироваться на модулях вашего приложения, которые мы изучим далее.

```
define([
    "backbone.layoutmanager"
], function() {
    // создание глобального места для хранения конфигурационных параметров
    // и создания модулей.
    var app = {
        // Корневой путь для запуска приложения.
        root: "/"
    };
    // Локализация или создание нового объекта JavaScript-шаблона.
    var JST = window.JST = window.JST || {};
    // Конфигурирование менеджера макетов значениями
    // по умолчанию Backbone Boilerplate.
    Backbone.LayoutManager.configure({
        // разрешаем менеджеру макетов LayoutManager расширить
        // Backbone.View.prototype.
        manage: true,
        prefix: "app/templates/",
        fetch: function(path) {
            // добавление расширения файла.
            path = path + ".html";
            // использовать скомпилированный шаблон, если путь кэширован
            if (JST[path]) {
                return JST[path];
            }
            // Перевод fetch в асинхронный режим.
            var done = this.async();
            // асинхронный поиск шаблона.
            $.get(app.root + path, function(contents) {
                done(JST[path] = _.template(contents));
            });
        }
    });
    // Примешивание класса Backbone.Events, модулей и управления макетами
    // к объекту приложения.
    return _.extend(app, {
        // создание настраиваемого объекта с вложенным объектом Views.
        module: function(additionalProps) {
            return _.extend({ Views: {} }, additionalProps);
        },
        // вспомогательная функция для использования макетов.
        useLayout: function(name, options) {
            // Включаем арность переменной, разрешая первому аргументу
            // быть объектом options
            // и опуская аргумент name.
        }
    });
});
```

```

if (_.isObject(name)) {
    options = name;
}
// гарантируем, что options - это объект.
options = options || {};
// Если свойство name было указано, то используем его
// в качестве шаблона.
if (_.isString(name)) {
    options.template = name;
}
// создаем новый макет с параметрами.
var layout = new Backbone.Layout(_.extend({
    el: "#main"
}, options));
// кэшируем ссылку.
return this.layout = layout;
},
}, Backbone.Events);
});

```



JST является сокращением от *JavaScript templates* (шаблоны JavaScript) и обычно ссылается на шаблоны, которые были (или будут) предкомпилированы в процессе сборки. При выполнении команды `bbb release` или `bbb debug` шаблоны Underscore/Lo-dash предварительно компилируются для того, чтобы их не нужно было компилировать в браузере во время выполнения приложения.

Создание модулей Backbone Boilerplate

Модуль Backbone Boilerplate (не путайте его с простым AMD-модулем) представляет собой сценарий, состоящий из:

- модели;
- коллекции;
- представлений (не обязательно).

Мы можем создать новый модуль Boilerplate с помощью программы `grunt-bbb`, еще раз воспользовавшись командой `init`:

```

# Create a new module
$ bbb init:module
# Grunt prompt
Please answer the following:
[?] Module Name foo
[?] Do you need to make any changes to the above before continuing? (y/N)
Writing app/modules/foo.js...OK
Writing app/styles/foo.styl...OK

```

продолжение ↗

```
Writing app/templates/foo.html...OK
Initialized from template "module".
Done, without errors.
```

В результате будет сгенерирован следующий модуль foo.js:

```
// Демонстрационный модуль
define([
    // Приложение.
    "app"
],
// отображение зависимостей из массива выше.
function(app) {
    // создание нового модуля.
    var Foo = app.module();
    // модель по умолчанию.
    Foo.Model = Backbone.Model.extend({
    });
    // коллекция по умолчанию.
    Foo.Collection = Backbone.Collection.extend({
        model: Foo.Model
    });
    // представление по умолчанию.
    Foo.Views.Layout = Backbone.Layout.extend({
        template: "foo"
    });
    // возврат модуля для совместимости с AMD.
    return Foo;
});
```

Обратите внимание, как мы избавились от необходимости писать типовой код для модели, коллекции и представления.

Возможно, в приложении нам также понадобится указать ссылки на плагины, например плагин локального хранилища или оффлайновый адаптер. Один из способов явно включить плагин в приведенном выше коде выглядит так:

```
// Демонстрационный модуль
define([
    // Приложение.
    "app",
    // Плагины
    'plugins/backbone-localstorage'
],
// отображение зависимостей из массива выше.
function(app) {
    // Создание нового модуля.
    var Foo = app.module();
    // Модель по умолчанию.
    Foo.Model = Backbone.Model.extend({
        // Save all of the items under the `foo` namespace.
        localStorage: new Store('foo-backbone'),
```

```

});  

// Коллекция по умолчанию.  

Foo.Collection = Backbone.Collection.extend({  

    model: Foo.Model  

});  

// Представление по умолчанию.  

Foo.Views.Layout = Backbone.Layout.extend({  

    template: "foo"  

});  

// Возврат модуля для совместимости с AMD.  

return Foo;  

});

```

router.js

Наконец, рассмотрим маршрутизатор нашего приложения, он обрабатывает навигацию. Backbone Boilerplate создает маршрутизатор по умолчанию, который включает разумные параметры «из коробки» и может быть легко расширен.

```

define([
    // Приложение.  

    "app"
],  

function(app) {  

    // Определение маршрутизатора приложения;  

    // здесь можно подключить субмаршрутизаторы.  

    var Router = Backbone.Router.extend({  

        routes: {  

            "" : "index"  

        },  

        index: function() {  

        }  

    });
    return Router;
});

```

Если необходимо выполнить какую-либо специфичную для модуля логику, то при загрузке страницы (например, когда пользователь переходит по маршруту по умолчанию) мы можем подгрузить модуль как зависимость и дополнительно воспользоваться менеджером макетов Backbone, чтобы подключить представления к нашему макету, как показано ниже:

```

define([
    // Приложение.  

    'app',
    // Модули.  

    'modules/foo'
],  

function(app, Foo) {  

    // Определение маршрутизатора приложения;  

    // здесь можно подключить субмаршрутизаторы.

```

продолжение ↗

```
var Router = Backbone.Router.extend({
  routes: {
    '' : 'index'
  },
  index: function() {
    // Создание новой коллекции
    var collection = new Foo.Collection();
    // Использование и конфигурирование макета 'main'
    app.useLayout('main').setViews({
      // Включение представления строки в представление содержимого
      '.bar': new Foo.Views.Bar({
        collection: collection
      })
    }).render();
  }
});
// Получение данных (например, из локального хранилища)
collection.fetch();
return Router;
});
```

Другие полезные инструменты и проекты

При работе с Backbone вам, как правило, необходимо создать для своего приложения несколько различных классов и файлов. Вспомогательные инструменты вроде Grunt-BBB автоматизируют этот процесс, генерируя стандартный код для нужных вам файлов.

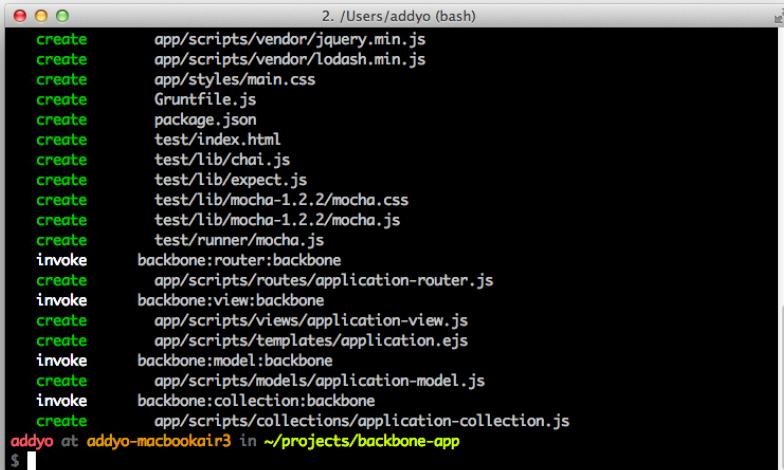
Yeoman

Если вам понравилось работать с Grunt-BBB, но вы хотите изучить инструмент, предназначенный для более широкого круга проектных задач, то я рекомендую воспользоваться инструментальным комплектом Yeoman, в создании которого я участвовал (рис. 11.2).

Yeoman представляет собой набор инструментов и методов, повышающих эффективность разработки приложений. В состав Yeoman входит вспомогательный инструмент *yo* (см. рис. 11.2), программа сборки *Grunt* и менеджер пакетов *Bower*, работающий на клиентской стороне (рис. 11.3).

Основная цель Grunt-BBB заключается в том, чтобы дать уверенный старт Backbone-проектам, комплект Yeoman обеспечивает скаффолдинг приложений с помощью Backbone (или других фреймворков), предоставляет доступ к плагинам Backbone из командной строки и компилирует код на *CoffeeScript*, *Sass* и других абстрактных языках без дополнительных усилий.

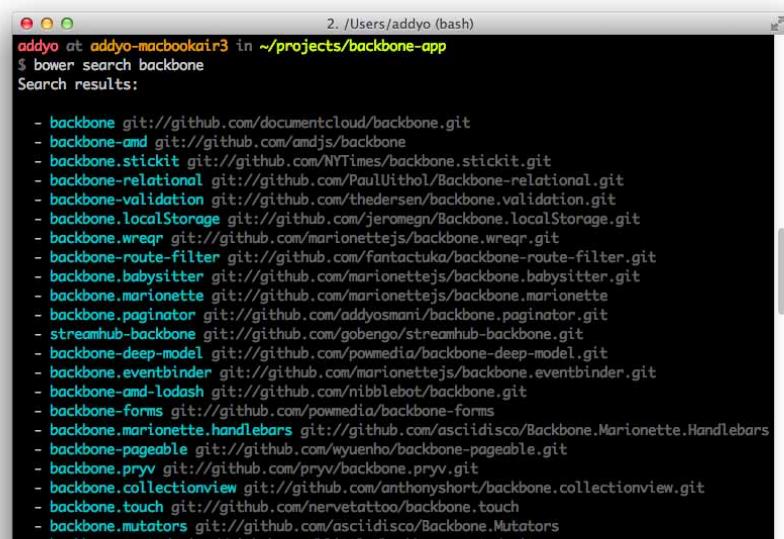
Вам также может быть интересен *Brunch* — похожий проект, в котором генерация новых приложений осуществляется на основе каркасов из базового кода.



2. /Users/addyo (bash)

```
create    app/scripts/vendor/jquery.min.js
create    app/scripts/vendor/lodash.min.js
create    app/styles/main.css
create    Gruntfile.js
create    package.json
create    test/index.html
create    test/lib/chai.js
create    test/lib/expect.js
create    test/lib/mocha-1.2.2/mocha.css
create    test/lib/mocha-1.2.2/mocha.js
create    test/runner/mocha.js
invoke  backbone:router:backbone
create    app/scripts/routes/application-router.js
invoke  backbone:view:backbone
create    app/scripts/views/application-view.js
create    app/scripts/templates/application.ejs
invoke  backbone:model:backbone
create    app/scripts/models/application-model.js
invoke  backbone:collection:backbone
create    app/scripts/collections/application-collection.js
addyo at addyo-macbookair3 in ~/projects/backbone-app
```

Рис. 11.2. Инструмент «уо», помогающий создать новое Backbone-приложение



2. /Users/addyo (bash)

```
addyo at addyo-macbookair3 in ~/projects/backbone-app
$ bower search backbone
Search results:
```

- backbone git://github.com/documentcloud/backbone.git
- backbone-amd git://github.com/amdjs/backbone
- backbone-stickit git://github.com/NTYTimes/backbone.stickit.git
- backbone-relational git://github.com/PaulUithol/Backbone-relational.git
- backbone-validation git://github.com/thedersen/backbone.validation.git
- backbone.localStorage git://github.com/jeromegn/Backbone.localStorage.git
- backbone.wreqr git://github.com/marionettejs/backbone.wreqr.git
- backbone-route-filter git://github.com/fantactuka/backbone-route-filter.git
- backbone.babysitter git://github.com/marionettejs/backbone.babysitter.git
- backbone.marionette git://github.com/marionettejs/backbone.marionette
- backbone.paginator git://github.com/addyosmani/backbone.paginator.git
- streamhub-backbone git://github.com/gobengo/streamhub-backbone.git
- backbone-deep-model git://github.com/powmedia/backbone-deep-model.git
- backbone.eventbinder git://github.com/marionettejs/backbone.eventbinder.git
- backbone-amd-lodash git://github.com/nibblebot/backbone.git
- backbone-forms git://github.com/powmedia/backbone-forms
- backbone.marionette.handlebars git://github.com/asciidisco/Backbone.Marionette.Handlebars
- backbone-pageable git://github.com/wyuenho/backbone-pageable.git
- backbone.pryv git://github.com/pryv/backbone.pryv.git
- backbone.collectionview git://github.com/anthonyshort/backbone.collectionview.git
- backbone.touch git://github.com/nervetattoo/backbone.touch
- backbone.mutators git://github.com/asciidisco/Backbone.Mutators
- backbone-nested git://github.com/blittle/backbone-nested.git

Рис. 11.3. Список плагинов и расширений Backbone, доступных через менеджер пакетов Bower

Backbone DevTools

Для выполнения рутинных отладочных действий при разработке Backbone-приложений существует специальный инструментарий.

К примеру, комплект Backbone DevTools является расширением Chrome DevTools и позволяет анализировать события, синхронизацию, связи между представлениями и DOM-элементами, а также наблюдать за экземплярами объектов (рис. 11.4).

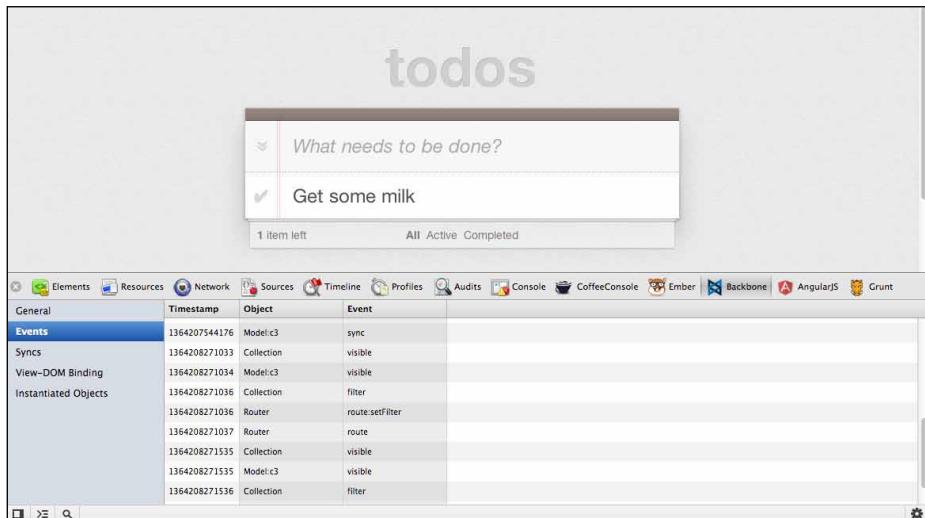


Рис. 11.4. Расширение Backbone DevTools используется для отладки приложения, управляющего задачами, которое мы разработали ранее в этой книге

В панели Elements (Элементы) отображается полезная иерархия представлений. Когда вы инспектируете DOM-элемент, ближайшее представление будет отображено в консоли через `$view`.

На момент написания книги этот проект был доступен на GitHub.

Заключение

В этой главе мы рассмотрели комплект Backbone Boilerplate и научились пользоваться инструментом BBB для скраффолдинга нашего приложения.

Для получения дополнительной информации о том, как этот проект позволяет структурировать приложения, в составе BBB имеются встроенные примеры приложений, которые легко сгенерировать и изучить.

Среди этих примеров – базовый учебный проект (`bbb init:tutorial`) и реализация приложения для управления задачами (`bbb init:todomvc`). Я рекомендую изучить их, поскольку они сформируют у вас более полное представление о том, какое место Backbone Boilerplate и его шаблоны занимают в общей процедуре создания веб-приложения.

Дополнительную информацию о Grunt-BBB можно получить в официальном репозитории этого проекта. Также существует презентация о Grunt-BBB, которая будет полезна заинтересованным читателям.

12

Backbone и jQuery Mobile

Разработка мобильных приложений с помощью jQuery Mobile

Веб-навигация на мобильных устройствах получила очень широкое распространение и продолжает быстро набирать популярность. Этому сопутствует большое разнообразие устройств и браузеров. По этой причине разработка кроссплатформенных приложений, способных работать на мобильных устройствах, — одновременно важная и сложная задача. Создавать «родные» приложения дорого; это требует большого количества времени и, как правило, обширного опыта разработчиков в таких языках программирования, как Objective C , C#, Java и JavaScript, чтобы приложение могло работать в различных средах исполнения.

HTML, CSS и JavaScript позволяют разрабатывать единственное приложение для общедоступной среды исполнения — браузера. Такой подход дает возможность создавать программы для широкого круга мобильных устройств, в том числе для планшетных компьютеров, смартфонов и карманных компьютеров, а также традиционных ПК.

Сложность разработки состоит не только в том, чтобы адаптировать текст и картинки к экранам с различными расширениями, но и в том, чтобы обеспечить один и тот же пользовательский интерфейс в различных операционных системах. Фреймворк jQuery Mobile (или jQM), как и jQueryUI, позволяет разрабатывать пользовательские интерфейсы на основе jQuery и работает на всех популярных платформах для мобильных телефонов, планшетных компьютеров, электронных книг и настольных ПК. jQuery Mobile задуман как легкодоступная и универсальная технология.

Основная идея фреймворка — сделать возможной разработку мобильных приложений исключительно средствами HTML. Знание языков программирования

не требуется; не нужно создавать и сложные, специфичные для устройства CSS-таблицы. Чтобы интегрировать jQMobile в Backbone, сначала следует усвоить два основных принципа: *прогрессивное улучшение* и *адаптивный веб-дизайн*.

Принцип прогрессивного улучшения виджетов в jQMobile

Фреймворк jQuery Mobile следует принципам прогрессивного улучшения¹ и адаптивного веб-дизайна² с использованием определений и конфигураций разметки HTML5.

В jQuery Mobile страница состоит из элемента с атрибутом `data-role="page"`. Внутри контейнера страницы используется любая корректная HTML-разметка, однако в jQM-страницах непосредственными дочерними элементами чаще всего являются теги `div` с атрибутами `data-role="header"`, `data-role="content"` и `data-role="footer"`. Единственное жесткое требование — поддержка системы навигации оберткой страницы; все остальное является необязательным.

Исходная HTML-страница выглядит следующим образом:

```
<!DOCTYPE html>
<html>
<head>
    <title>Page Title</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
        href="http://code.jquery.com/mobile/1.3.0/jquery.mobile-1.3.0.min.css" />
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script src="http://code.jquery.com/mobile/1.3.0/jquery.mobile-1.3.0.min.js">
    </script>
</head>
<body>
<div data-role="page">
    <div data-role="header">
        <h1>Page Title</h1>
    </div>
    <div data-role="content">
```

продолжение ↗

¹ Принцип прогрессивного улучшения разделяет возможности веб-платформы на уровни и обеспечивает пользователю доступ к базовому содержимому и функциям страницы даже при отключенном JavaScript и устаревшем браузере. При просмотре страницы в более современном браузере с включенным JavaScript пользователю доступен более широкий функционал страницы.

² Адаптивный веб-дизайн — это разработка веб-страниц, структура которых изменяется в зависимости от среды просмотра и придает им оптимальный внешний вид. В таких страницах часто используются медиазапросы CSS.

```

<p>Page content goes here.</p>
<form>
    <label for="slider-1">Slider with tooltip:</label>
    <input type="range" name="slider-1" id="slider-1" min="0"
           max="100" value="50"
           data-popup-enabled="true">
</form>
</div>
<div data-role="footer">
    <h4>Page Footer</h4>
</div>
</div>
</body>
</html>

```

jQuery Mobile преобразует это HTML-определение в визуализированный HTML и CSS с помощью API прогрессивного улучшения виджетов. Он также запускает JavaScript-сценарии, указанные в конфигурациях, свойствах атрибутов и настройках этапа исполнения. Результат показан на рис. 12.1.

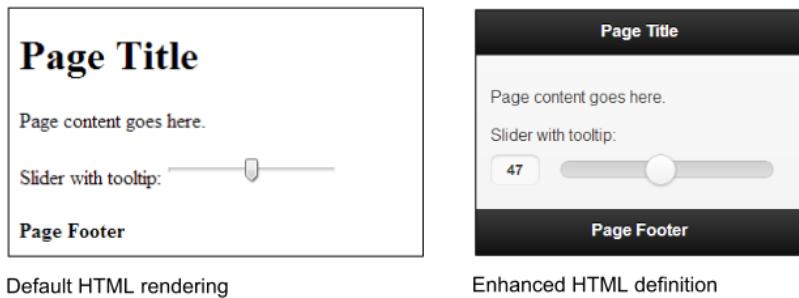


Рис. 12.1. Сравнение пользовательского интерфейса с HTML-элементами по умолчанию и расширенной версии на основе jQuery Mobile

Это означает, что при любом добавлении или изменении HTML-содержимого оно должно быть обработано системой прогрессивного улучшения виджетов фреймворка jQuery Mobile.

Система навигации jQuery Mobile

Система навигации jQuery Mobile управляет жизненным циклом приложения; она автоматически перехватывает стандартные ссылки и отправляемые формы и преобразует их в AJAX-запросы. Событие, возникающее при щелчке по ссылке или отправке формы, автоматически перехватывается, и вместо перезагрузки страницы создается AJAX-запрос на основе атрибута `href` или действия формы.

При запросе страницы jQuery Mobile ищет в ее документе все элементы с атрибутом `data-role="page"`, разбирает его содержимое и вставляет код в DOM исходной страницы. Как только новая страница сформирована, JavaScript фреймворка jQuery Mobile генерирует переход, который отображает новую страницу и скрывает HTML-код предыдущей страницы в DOM.

Затем к виджетам новой страницы применяются стили и добавляется логика. Остальная часть страницы отбрасывается, поэтому сценарии, таблицы стилей и другая информация не включаются в нее.

Благодаря функции *многостраницных шаблонов* вы можете добавлять в тег `<body>` HTML-файла любое количество страниц, задавая теги `divs` с атрибутами `data-role="page"` или `data-role="dialog"`, а также `id`, который можно использовать в ссылках (начинающихся с `#`):

```
<html>
  <head>...</head>
  <body>
    ...
    <div data-role="page" id="firstpage">
      ...
      <div data-role="content">
        <a href="#secondpage">go to secondpage</a>
      </div>
    </div>
    <div data-role="page" id="secondpage">
      ...
      <div data-role="content" >
        <a href="#firstdialog" data-rel="dialog" >open a page as a dialog</a>
      </div>
    </div>
    <div data-role="dialog" id="firstdialog">
      ...
      <div data-role="content">
        <a href="#firstpage">leave dialog and go to first page</a>
      </div>
    </div>
  </body>
</html>
```

Чтобы, к примеру, перейти на страницу `secondpage`, которая будет отображена в модальном диалоговом окне с использованием эффекта `fade-transition`, достаточно добавить в тег якоря атрибуты `data-rel="dialog"`, `datatransition="fade"` и `href="#index.html#secondpage"`.

jQuery Mobile обладает своим собственным циклом обработки событий и, можно сказать, является маленьким MVC-фреймворком, содержащим возможности прогрессивного улучшения виджетов, ранней загрузки, кэширования и создания многостраницных шаблонов, которые реализуются путем

конфигурирования HTML-кода. Разработчик, использующий Backbone.js, не обязан знать внутреннее устройство jQuery Mobile, но должен понимать, как задать HTML-конфигурации, участвующие в обработке событий. В разделе «Перехват событий jQuery Mobile» подробно рассматриваются ситуации, в которых требуется филигранная настройка JavaScript.

Для получения дополнительной вводной информации о фреймворке jQuery Mobile и подробного ознакомления с принципами его работы посетите следующие ссылки:

- <http://view.jquerymobile.com/1.3.0/docs/intro/>;
- <http://view.jquerymobile.com/1.3.0/docs/widgets/pages/>;
- <http://view.jquerymobile.com/1.3.0/docs/intro/rwd.php>.

Настройка Backbone-приложения для работы с jQuery Mobile

Первая проблема, с которой разработчики обычно сталкиваются при создании приложений на основе jQuery Mobile и MV*-фреймворка, заключается в том, что оба фреймворка «хотят» обрабатывать навигацию по приложению.

Чтобы использовать Backbone совместно с jQuery Mobile, сначала нужно отключить систему навигации и прогрессивного улучшения jQuery Mobile, а на следующем шаге воспользоваться его API, чтобы применить конфигурации и улучшить компоненты приложения уже в рамках жизненного цикла Backbone.

Мобильное приложение, представленное на рис. 12.2, основано на кодовой базе примера *Backbone-Require.js*, управляющего задачами (см. главу 8), и расширено поддержкой jQuery Mobile.

В данной реализации используется инструмент Grunt-BBB и библиотека *Handlebars.js*. Будут предоставлены и другие утилиты, полезные при разработке мобильных приложений, которые можно легко сочетать друг с другом и расширять, как показано на рис. 12.3 (см. главы 6 и 11).

Библиотека *Require.js* загружает файлы в следующем порядке:

- 1) jQuery;
- 2) Underscore/Lo-Dash;
- 3) handlebars.compiled;
- 4) TodoRouter (создает экземпляры определенных представлений);
- 5) jQueryMobile;

- 6) `jQueryMobileCustomInitConfig`;
- 7) Создание маршрутизатора Backbone.

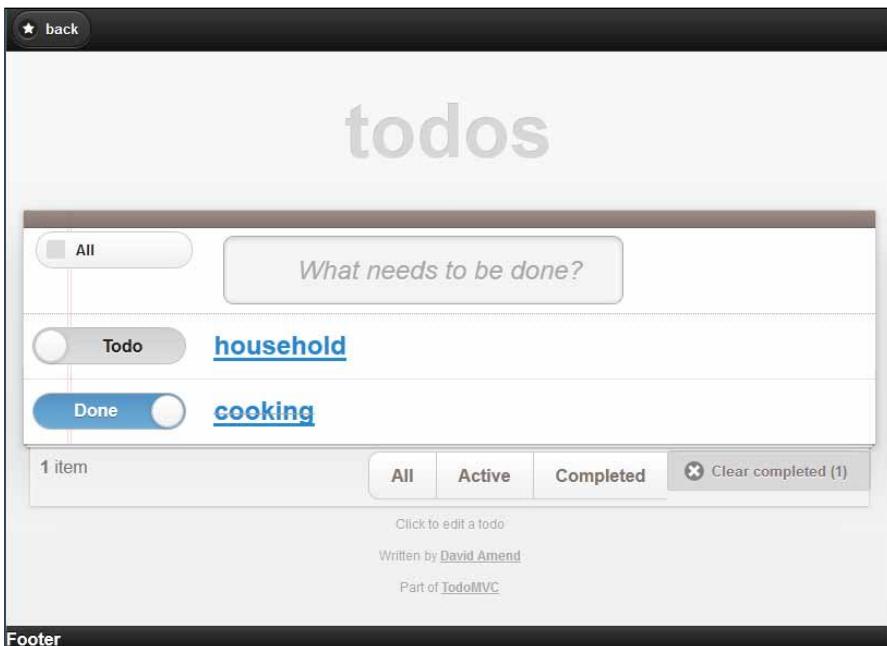


Рис. 12.2. Приложение, управляющее задачами, с поддержкой jQuery Mobile

Есть открыть консоль в директории проекта и запустить команду `grunt handlebars` или `grunt watch`, то все файлы шаблонов будут объединены и скомпилированы в файл `dist/debug/handlebars_package`. Чтобы запустить приложение, выполните команду `grunt server`.

Когда маршрутизатор Backbone выполняет перенаправление, создаются следующие файлы:

- `BasicView.js` и `basic_page_simple.template`. Представление `BasicView` отвечает за обработку многостраничных шаблонов Handlebars. Его метод `render` вызывает метод `$.mobile.changePage` jQuery Mobile API, чтобы обработать переход на страницу и выполнить прогрессивное улучшение виджетов.
- *Фиксированное представление с его частью шаблона — например, `EditTodoPage.js` и `editTodoView.template_partial`.* Заголовочный раздел файла `index.html` должен загрузить файл `jquerymobile.css`, файл `base.css`, который используется всеми приложениями Todo-MVC, а также файл `index.css`, в котором содержатся специфичные для проекта таблицы стилей.

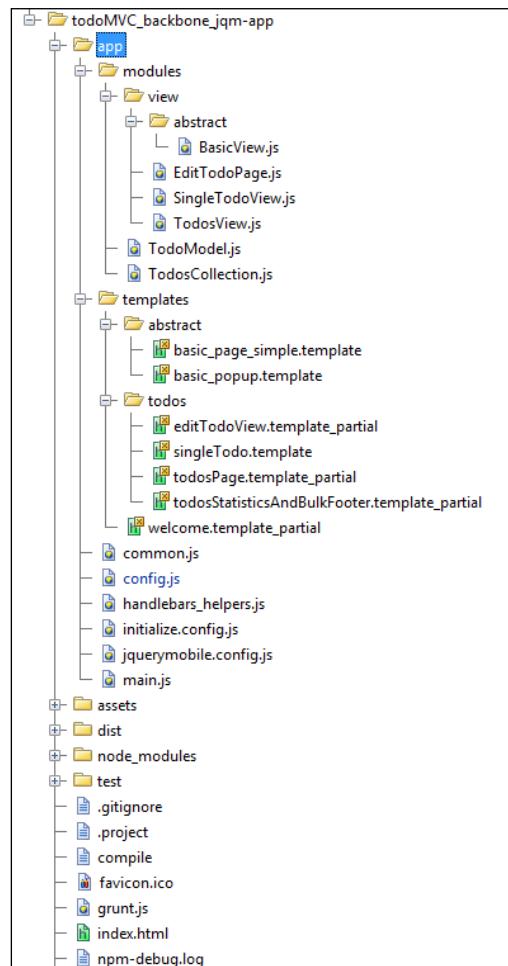


Рис. 12.3. Рабочая область приложения TodoMVC с jQueryMobile и Backbone

```

<html>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>TodoMVC Jquery Mobile</title>
    <!--виджеты и стили адаптивного дизайна -->
    <link rel="stylesheet" href="/assets/css/jquerymobile.css">
    <!--используется всеми приложениями TodoMVC -->
    <link rel="stylesheet" href="/assets/css/base.css">
    <!-- пользовательские css -->
    <link rel="stylesheet" href="/assets/css/index.css">

```

```
</head>
<body>
  <script data-main="/app/config" src="/assets/js/libs/require.js"></script>
</body>
</html>
```

Рабочий цикл приложения на основе Backbone и jQueryMobile

Делегирование функций маршрутизации и навигации фреймворка jQuery Mobile библиотеке Backbone дает нам возможность с ее помощью четко структурировать приложение, чтобы впоследствии легко разделить его логику между «настольной» веб-страницей, планшетными компьютерами и мобильными приложениями.

Теперь необходимо преодолеть различия в механизмах обработки запросов Backbone и jQuery Mobile. Класс Backbone.Router позволяет разработчику явно задавать собственные маршруты навигации, а jQuery Mobile использует локальные URL-ссылки для обращения к страницам или представлениям, находящимся в одном документе.

В прошлом для устранения этой проблемы предлагалось вручную модифицировать Backbone и jQuery Mobile. Решение, которое будет продемонстрировано ниже, не только упрощает обработку события инициализации компонента jQuery Mobile, но и дает возможность использовать существующие обработчики Backbone-маршрутизатора.

Для передачи управления навигацией от фреймворка jQuery Mobile к библиотеке Backbone необходимо сначала задать несколько специфических настроек для события `mobileinit`, которое генерируется после загрузки фреймворка; это даст маршрутизатору возможность принять решение о том, какую страницу загрузить.

Данные настройки находятся в файле `jquerymobile.config.js`, который делегирует библиотеке Backbone функции навигации jQM, а также разрешает создание виджетов вручную:

```
$(document).bind("mobileinit", function(){
// Отключение jQM-событий маршрутизации и создания компонентов
  // отключение маршрутизации #-путей
  $.mobile.hashListeningEnabled = false;
  // отключение контроля якорей
  $.mobile.linkBindingEnabled = false;
  // может привести к двукратному созданию объекта
  // и проблемы с кнопкой возврата решены
  $.mobile.ajaxEnabled = false;
  // в противном случае после mobileinit выполняется попытка
```

продолжение ↗

```
// загрузить исходную страницу
$.mobile.autoInitializePage = false;
// мы хотим самостоятельно обрабатывать кэширование и очистку DOM
$.mobile.page.prototype.options.domCache = false;
// вызвано проблемами совместимости
// не поддерживается всеми браузерами
$.mobile.pushStateEnabled = false;
// решает проблему с кнопкой возврата в phonegap
$.mobile.phonegapNavigationEnabled = true;
// встроенный календарь не будет конфликтовать с компонентом jQM
$.mobile.page.prototype.options.degradeInputs.date = true;
});
```

Ниже рассматривается поведение и использование нового механизма со следующим разделением функций:

- 1) маршрутизации страниц с фиксированными представлениями;
- 2) управление шаблонами мобильных страниц;
- 3) управление DOM;
- 4) `$.mobile.changePage`.

Приведенные на рис. 12.4 шаги 1–11 относятся к новому рабочему циклу мобильного приложения.

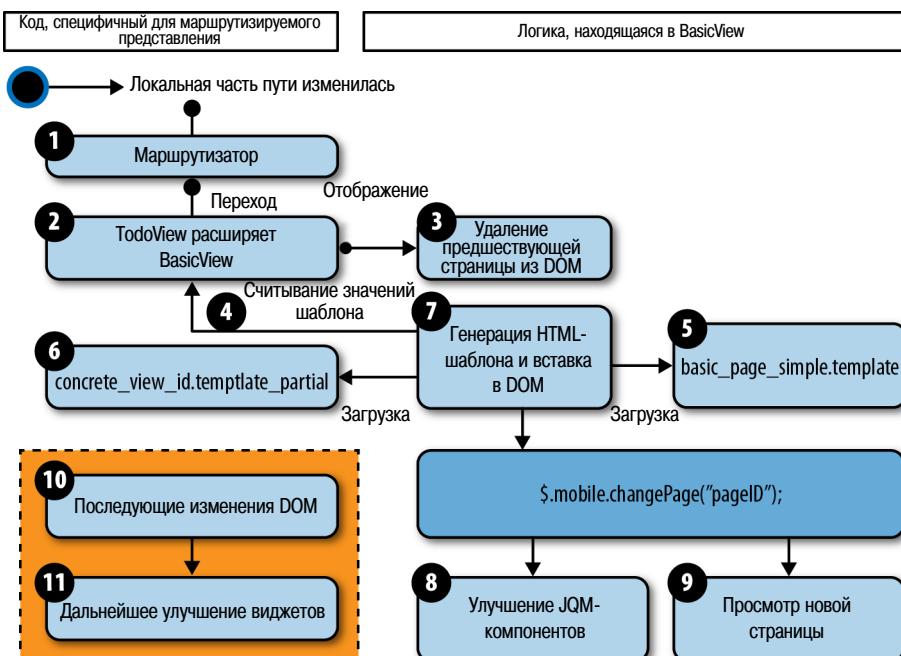


Рис. 12.4. Рабочий цикл приложения TodoMVC, использующего Backbone и jQueryMobile

Переход на страницу с фиксированным представлением, наследование класса BasicView

При изменении URL (например, из-за щелчка по ссылке) приведенная выше конфигурация предотвращает генерацию jQM-событий. Вместо этого маршрутизатор Backbone прослушивает изменения локальных URL и определяет, какое представление необходимо запросить.

При разработке мобильных страниц хорошей практикой является создание базовых прототипов для ключевых компонентов jQM, таких как базовые страницы, всплывающие и диалоговые окна, а также для использования плагина валидации jQuery. Это значительно упрощает изменение специфичной для устройства логики представлений на этапе выполнения приложения, а также реализацию общих стратегий. Кроме того, это поможет добавить синтаксис и обеспечить поддержку множественных цепочек наследования прототипов при использовании JavaScript и Backbone.

Создавая суперкласс BasicView, мы даем всем дочерним страницам представлений возможность работать с jQM единым способом, а также использовать общий шаблонизатор и механизмы взаимодействия с представлениями.

При сборке проекта с помощью инструментов Grunt/Yeoman компиляция семантических шаблонов выполняется библиотекой Handlebar.js, а файлы AMD-шаблонов объединяются в один файл. Включение всех определений страниц в единый файл приложения позволяет пользоваться им в оффлайн-режиме, что важно для приложений, предназначенных для мобильных устройств.

Управление шаблонами мобильных страниц

Внутри фиксированной страницы представления можно переопределять статические свойства и функции, заменяя их динамическими значениями суперкласса BasicView. Позже BasicView обработает эти значения и создаст HTML-код страницы jQuery Mobile с помощью библиотеки Handlebars.

Дополнительные динамические параметры шаблонов (например, информация о Backbone-моделях) будут взяты из определенного представления и объединены с параметрами класса BasicView.

Фиксированное представление может иметь следующий вид (файл *EditTodoPage.js*):

```
define([
    "backbone", "modules/view/abstract/BasicView"],
    function (Backbone, BasicView) {
        return BasicView.extend({
            id : "editTodoView",
            getHeaderTitle : function () {
                return "Edit Todo";
            },
        });
    }
);
```

продолжение ↗

```

        getSpecificTemplateValues : function () {
            return this.model.toJSON();
        },
        events : function () {
            // объединенные события BasicView (чтобы добавить
            // исправление функций кнопки возврата)
            return _.extend({
                'click #saveDescription' : 'saveDescription'
            }, this.constructor.__super__.events);
        },
        saveDescription : function (clickEvent) {
            this.model.save({
                title : $("#todoDescription", this.el).val()
            });
            return true;
        }
    );
});

```

По умолчанию класс `BasicView` использует Handlebars-шаблон `basic_page_simple.template`. Если вам необходимо задать собственный шаблон или создать абстрактное представление *Super* с другим шаблоном, переопределите функцию `getTemplateID`:

```

getTemplateID : function(){
    return "custom_page_template";
}

```

Согласно принятому правилу, атрибут `id` используется в качестве идентификатора jQM-страницы и имени файла соответствующего шаблона, который будет вставлен в шаблон `basic_page_simple.template`.

Каждая фиксированная страница станет частью элемента `datarole="content"`, где находится параметр `templatePartialPageID`.

Позднее результат функции `getHeaderTitle` из `EditTodoPage` заменит `headerTitle` в абстрактном шаблоне (`basic_page_simple.template`).

```

<div data-role="header">
    {{whatis "Specific loaded Handlebars parameters:"}}
    {{whatis this}}
    <h2>{{headerTitle}}</h2>
    <a id="backButton" href="javascript:history.go(-1);">
        data-icon="star" data-rel="back" >back</a>
</div>
<div data-role="content">
    {{whatis "Template page trying to load:"}}
    {{whatis templatePartialPageID}}
    {{> templatePartialPageID}}
</div>

```

```
<div data-role="footer">
  {{footerContent}}
</div>
```



Помощник представления `whatis` библиотеки Handlebars ведет простой журнал параметров.

Все дополнительные параметры, возвращаемые методом `getSpecificTemplateValues`, вставляются в фиксированный шаблон `editTodoPage.template_partial`.

Поскольку считается, что элемент `footerContent` используется редко, его содержимое возвращается методом `getSpecificTemplateValues`.

В ситуации с представлением `EditTodoPage` возвращается вся информация модели, а заголовок используется в фиксированной странице-части:

```
<div data-role="fieldcontain">
  <label for="todoDescription">Todo Description</label>
  <input type="text" name="todoDescription" id="todoDescription"
    value="{{title}}" />
</div>
<a id="saveDescription" href="#" data-role="button" data-mini="true">Save</a>
```

При вызове метода `render` загружаются шаблоны `basic_page_simple.template` и `editTodoView.template_partial`, а параметры `EditTodoPage` и `BasicView` объединяются и генерируются библиотекой Handlebars для создания следующего текста:

```
<div data-role="header">
  <h2>Edit Todo</h2>
  <a id="backButton" href="javascript:history.go(-1);"
    data-icon="star" data-rel="back" >back</a>
</div>
<div data-role="content">
  <div data-role="fieldcontain">
    <label for="todoDescription">Todo Description</label>
    <input type="text" name="todoDescription" id="todoDescription"
      value="Cooking" />
  </div>
  <a id="saveDescription" href="#" data-role="button" data-mini="true">Save
  </a>
</div>
<div data-role="footer">
  Footer
</div>
```

В следующем разделе рассматривается сбор параметров шаблона классом `BasicView` и загрузка определения HTML.

Управление DOM и методом \$.mobile.changePage

При выполнении метода `render` (строка 29 в листинге исходного кода следующего примера) представление `BasicView` сначала очищает DOM, удаляя предыдущую страницу (строка 70). Для удаления элементов из DOM нельзя использовать метод `$.remove`, однако можно воспользоваться методом `$previousEl.detach()`, поскольку `detach` не удаляет события и данные, связанные с элементом.

Это важный момент, поскольку библиотеке jQuery Mobile все еще нужна данная информация (например, для запуска действий при переключении на другую страницу). Помните, что впоследствии данные и события DOM также должны быть очищены во избежание проблем с производительностью приложения.

Для очистки DOM могут применяться стратегии, отличные от той, которая используется в функции `cleanupPossiblePageDuplicationInDOM`. Чтобы избежать дублирования DOM, можно удалить только старую страницу с таким же `id`, как у текущей страницы, после того как к ней уже были обращения. В некоторых приложениях можно заменить страницу с помощью механизма кэширования.

Далее, `BasicView` собирает все параметры шаблонов из фиксированного представления и вставляет HTML-код запрошенной страницы в тело шаблона. Это делается на этапах 4–7 на рис. 12.4 (строки 23–51 в листинге исходного кода).

На странице jQuery Mobile также будет установлен атрибут `data-role`. Его типичными значениями являются `page`, `dialog` и `popup`.

Как видно из файла `BasicView.js` (начиная со строки 74), функция `goBackInHistory` содержит вручную реализованную обработку кнопки возврата. В некоторых ситуациях функционал кнопки возврата в `jQuery Mobile` не работал с устаревшими версиями и блокировал работу системы навигации `jOMobile`.

```
1 define([
2     "lodash",
3     "backbone",
4     "handlebars",
5     "handlebars_helpers"
6 ],
7
8 function (_, Backbone, Handlebars) {
9     var BasicView = Backbone.View.extend({
10         initialize: function () {
11             _.bindAll();
12             this.render();
13         },
14         events: {
15             "click #backButton": "goBackInHistory"
16         }
17     });
18
19     return BasicView;
20 })(_, Backbone, Handlebars);
```

```
17      role: "page",
18      attributes: function () {
19          return {
20              "data-role": this.role
21          };
22      },
23      getHeaderTitle: function () {
24          return this.getSpecificTemplateValues().headerTitle;
25      },
26      getTemplateID: function () {
27          return "basic_page_simple";
28      },
29      render: function () {
30          this.cleanupPossiblePageDuplicationInDOM();
31          $(this.el).html(this.getBasicPageTemplateResult());
32          this.addPageToDOMAndRenderJQM();
33          this.enhanceJQMCComponentsAPI();
34      },
35 // Генерация HTML с использованием Handlebars-шаблонов
36      getTemplateResult: function (templateDefinitionID, templateValues) {
37          return window.JST[templateDefinitionID](templateValues);
38      },
39 // Сбор и слияние всех параметров шаблонов
40      getBasicPageTemplateResult: function () {
41          var templateValues = {
42              templatePartialPageID: this.id,
43              headerTitle: this.getHeaderTitle()
44          };
45          var specific = this.getSpecificTemplateValues();
46          $.extend(templateValues, this.getSpecificTemplateValues());
47          return this.getTemplateResult(this.getTemplateID(),
48              templateValues);
49      },
50      getRequestedPageTemplateResult: function () {
51          this.getBasicPageTemplateResult();
52      },
53      enhanceJQMCComponentsAPI: function () {
54 // changePage
55         $.mobile.changePage("#" + this.id, {
56             changeHash: false,
57             role: this.role
58         });
59 // добавление страницы в DOM
60         addPageToDOMAndRenderJQM: function () {
61             $("body").append($(this.el));
62             $("#" + this.id).page();
63         },
64 // Стратегия очистки DOM
65         cleanupPossiblePageDuplicationInDOM: function () {
66             // можно также переместить в событие "pagehide": или "onPageHide"
67             var $previousEl = $("#" + this.id);
68             var alreadyInDom = $previousEl.length >= 0;
```

продолжение ↗

```

69         if (alreadyInDom) {
70             $previousEl.detach();
71         }
72     },
73 // всегда поддерживать кнопку возврата при отключенной навигации
74     goBackInHistory: function (clickEvent) {
75         history.go(-1);
76         return false;
77     }
78 });
79
80     return BasicView;
81 });

```

После того как динамический HTML-код добавлен в DOM, необходимо применить метод `$.mobile.changePage` на шаге 8 (строка 54).

Это самый важный вызов API, поскольку он приводит к созданию компонента jQuery Mobile для текущей страницы.

Далее страница отображается пользователю на шаге 9 (рис. 12.5).

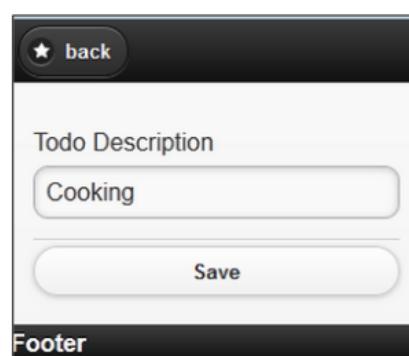
```

<a data-mini="true" data-role="button" href="#" id="saveDescription"
    data-corners="true"
    data-shadow="true" data-iconshadow="true" data-wrapperels="span" data-theme="c"
    class="ui-btn ui-shadow ui-btn-corner-all ui-mini ui-btn-up-c">
    <span class="ui-btn-inner">
        <span class="ui-btn-text">Save</span>
    </span>
</a>

```



Отображение по умолчанию HTML-кода, написанного вручную



Пользовательский интерфейс с таким же HTML-кодом, улучшенный с помощью jQM

Рис. 12.5. Внешний вид написанного вручную HTML-кода и страницы описания задачи, использующей jQuery Mobile

Улучшение пользовательского интерфейса происходит в методе `enhanceJQM-ComponentsAPI` (строка 52):

```
$.mobile.changePage("#" + this.id, {  
    changeHash: false,  
    role: this.role  
});
```

Чтобы сохранить управление локальными маршрутами, установим параметр `changeHash` в значение `false` и зададим подходящий параметр `role`, это обеспечит надлежащий внешний вид страницы. Наконец, метод `changePage` отобразит пользователю новую страницу с заданным переходом.

В простейших случаях рекомендуется создавать одно представление для каждой страницы, а если требуется выполнить улучшение виджетов, то всегда повторно отображать страницу целиком, вызывая метод `$.mobile.changePage`.

Для корректного отображения мобильных компонентов следует пользоваться сложными методами последовательного улучшения новых HTML-фрагментов. Будьте очень внимательны при создании частей HTML-кода и обновлении значений элементов пользовательского интерфейса. В следующем разделе мы расскажем о том, как это сделать.

Использование сложных методов jQM в Backbone

Динамическое изменение DOM

Описанное выше решение устраняет проблемы маршрутизации в Backbone вызовом метода `$.mobile.changePage('pageID')` и гарантирует, что вся HTML-страница будет улучшена с помощью разметки для фреймворка jQuery Mobile.

Следующий «трюк» с jQuery Mobile заключается в динамическом изменении определенного содержимого DOM (например, после загрузки содержимого с помощью Ajax). Я рекомендую использовать этот метод только в ситуациях, когда он явно приводит к существенному повышению производительности приложения.

jQM в текущей версии (1.3) предоставляет три способа, информация о которых есть в официальном API, на форумах и в блогах:

- `$(pageId).trigger('pagecreate')`

Создает разметку содержимого верхнего и нижнего колонтитулов.

○ \$(anyElement).trigger(create)

Создает разметку элемента и его дочерних элементов:

- \$(myListElement).listview(refresh);
- \$('[type=radio]).checkboxradio();
- \$('[type=text]).textInput();
- \$('[type=button]).button();
- \$('[data-role=navbar]).navbar();
- \$('[type=range]).slider();
- \$(select).selectmenu().



Каждый компонент jQM предоставляет плагинам методы, с помощью которых можно обновлять состояние определенных элементов пользовательского интерфейса.

Иногда при создании компонента с нуля вы встретите ошибку «Cannot call methods on ListView prior to initialization» («Невозможно вызывать методы ListView до инициализации»). Чтобы избежать ее, инициализируйте компонент до улучшения разметки следующим образом:

```
$('#mylist').listview().listview('refresh')
```

Для получения дополнительных сведений о создании страниц с использованием фреймворка JQM ознакомьтесь с его API и периодически читайте примечания к версиям:

- jQuery Mobile: Page Scripting (jQuery Mobile: создание сценариев для страниц);
- jQuery Mobile: Document Ready vs. Page Events (jQuery Mobile: Document Ready или страницные события);
- StackOverflow: Markup Enhancement of Dynamically Added Content (Переполнение стека: улучшение разметки динамически добавляемого содержимого).

Если вы собираетесь использовать **Model-Binding Plugin**, то вам необходимо создать механизм автоматического улучшения отдельных компонентов.

Теперь, после близкого знакомства с динамическим использованием DOM в сценариях, если вам не хочется повторно создавать компонент (например, **Listview**) с нуля, поскольку его загрузка требует больше времени, и, кроме того, вы желали бы упростить делегирование событий, воспользуйтесь специализированными

плагинами, которые будут обновлять лишь нужные HTML-компоненты и таблицы стилей.

В случае с `ListView` для обновления списка добавленных, отредактированных и удаленных записей вызовите следующую функцию:

```
$('#mylist').listview()
```

Чтобы принять решение о том, какой метод плагина вызвать, вам нужен способ определения типа компонента. Такой способ предоставляет адаптер jQuery Mobile под названием `Angular.js`.

Пример связывания моделей с использованием jQuery Mobile имеется на GitHub.

Перехват сообщений jQuery Mobile

В определенных ситуациях вам потребуется обрабатывать события, генерируемые jQuery Mobile. Это можно сделать следующим образом:

```
$('#myPage').live('pagebeforecreate', function(event){  
    console.log('page was inserted into the DOM');  
    // запустите здесь свои сценарии для улучшения элементов...  
    // запретите манипуляции страничного плагина  
    return false;  
});  
$('#myPage').live('pagecreate', function(event){  
    console.log('page was enhanced by jQM');  
});
```

В таких ситуациях важно знать, когда происходят события jQuery Mobile. На рис. 12.6 показан цикл события (страница А выгружается, а страница Б загружается).

Альтернативой является проект `jQuery Mobile Router`, с помощью которого можно заменить маршрутизатор `Backbone`. Этот проект предоставляет широкие возможности по перехвату и маршрутизации различных событий `jQM`. Он является расширением `jQuery Mobile` и может быть использован независимо.

Имейте в виду, что `jQM Router` лишен некоторых возможностей `Backbone`. `Router` и тесно связан с фреймворком `jQuery Mobile`. По этим причинам мы не использовали этот маршрутизатор в приложении `TodoMVC`. Если вы собираетесь воспользоваться им, имеет смысл создать собственную сборку `Backbone`, удалив из нее код маршрутизатора. Это позволит сэкономить до 25 % объема, составляющего 17,1 Кбайт.

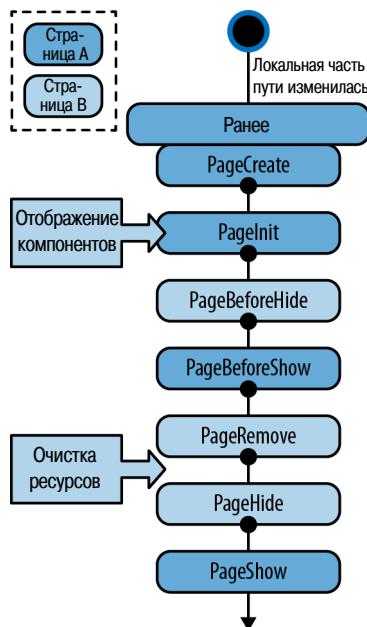


Рис. 12.6. Цикл события jQuery Mobile

Ознакомьтесь со сборщиком конфигураций Backbone.

Производительность

Производительность — важный аспект мобильных устройств. jQuery Mobile предоставляет различные инструменты для создания журналов производительности, с помощью которых вы получите наглядное представление о том, сколько времени занимает маршрутизация, улучшение компонентов и визуальные эффекты.

В зависимости от устройства переходы могут занимать до 90 % времени загрузки. Чтобы запретить все переходы, укажите режим переходов `none` при вызове метода `$.mobile.changePage()` в блоке кода, выполняющего конфигурирование:

```

$(document).bind("mobileinit", function(){
...
// в противном случае будет потрачено до 90 % времени загрузки
$.mobile.defaultPageTransition = "none";
$.mobile.defaultDialogTransition = "none";
});
}
  
```

или добавьте настройки, специфичные для устройства. Например:

```
$(document).bind("mobileinit", function(){
    var iosDevice =((navigator.userAgent.match(/iPhone/i))
    || (navigator.userAgent.match(/iPod/i))) ? true : false;
    $.extend( $.mobile , {
        slideText : (iosDevice) ? "slide" : "none",
        slideUpText : (iosDevice) ? "slideup" : "none",
        defaultPageTransition:(iosDevice) ? "slide" : "none",
        defaultDialogTransition:(iosDevice) ? "slide" : "none"
    });
});
```

Кроме того, подумайте над перспективой создания собственного механизма кэширования улучшенных страниц jQuery Mobile.

Новые релизы jQuery Mobile API регулярно совершенствуются с точки зрения технологий повышения производительности. Мы рекомендуем изучить самую свежую версию API и выбрать оптимальную стратегию кэширования с помощью динамических сценариев, которая наилучшим образом удовлетворяет требованиям вашего приложения.

Дополнительные материалы по производительности:

- инструменты профилирования jQuery Mobile;
- конфигурирование jQuery Mobile с учетом специфики устройства;
- средства отладки jQuery Mobile;
- функции кэширования jQuery Mobile.

Эффективная поддержка многоплатформенности

На сегодняшний день у большинства компаний уже имеется готовая веб-страница, и руководство принимает решение предоставить клиентам дополнительное мобильное приложение. Коды веб-страницы и мобильного приложения становятся независимыми друг от друга, и изменение их содержания или функций начинает отнимать значительно больше времени, чем требовалось для одной веб-страницы.

Тенденция к расширению разнообразия мобильных платформ лишь увеличивает затраты, необходимые для их поддержки. Кроме того, не всегда имеет смысл создавать приложения под каждое устройство. Но «веяния времени» требуют, чтобы пользователи имели доступ к содержимому приложения независимо от того, какую платформу или браузер они используют. Обязательно имейте в виду этот принцип при проектировании приложения.

Для решения проблем разработки многоплатформенных приложений существует два подхода — *адаптивный дизайн* и «*сначала мобильные*».

Архитектура мобильного приложения, представленная в этой главе, фактически обеспечивает значительную часть решения этих проблем, поскольку предоставляет готовую поддержку адаптивных макетов и даже браузеров, которые не способны обрабатывать медиазапросы. Возможно, это покажется неочевидным, но jQM представляет собой фреймворк для разработки пользовательских интерфейсов, не так уж сильно отличающийся от jQuery UI. Фреймворк jQuery Mobile использует фабрику виджетов и подходит не только для мобильных сред.

Для поддержки многоплатформенных браузеров с помощью jQuery Mobile и Backbone вы можете структурировать приложение следующими способами (перечислены в порядке возрастания затрат времени и усилий):

1. В идеале — один проект с кодом; для различных устройств отличаются только таблицы стилей.
2. Один проект с кодом; во время работы приложения используются HTML-шаблоны и суперклассы, соответствующие типу устройства, на котором оно выполняется.
3. Один проект с кодом; многократное использование API адаптивного дизайна и большинства виджетов jQuery Mobile. При запуске в «настольном» браузере добавляются некоторые компоненты другого фреймворка виджетов (например, *jQuery UI* или *Twitter Bootstrap*), которые управляются, например, HTML-шаблонами.
4. Один проект с кодом; во время работы приложения jQuery Mobile полностью заменяется другим фреймворком виджетов (например, *jQuery UI* или *Twitter Bootstrap*). Суперклассы и конфигурации, а также фиксированные элементы кода *Backbone.View* должны заменяться.
5. Различные проекты с кодом, которые используют общие модули.
6. Полнотью отдельный проект с кодом для настольного приложения. Такой подход может использоваться из-за абсолютно разных языков программирования или фреймворков, отсутствия знаний об адаптивном дизайне или доставшегося в наследство хаоса.

Идеальное решение заключается в создании симпатичного настольного приложения с использованием единственного мобильного фреймворка, и это возможно!

Взгляните в браузере на страницу, созданную с помощью jQuery Mobile API, вы увидите, что она совсем не похожа на мобильное приложение (рис. 12.7).

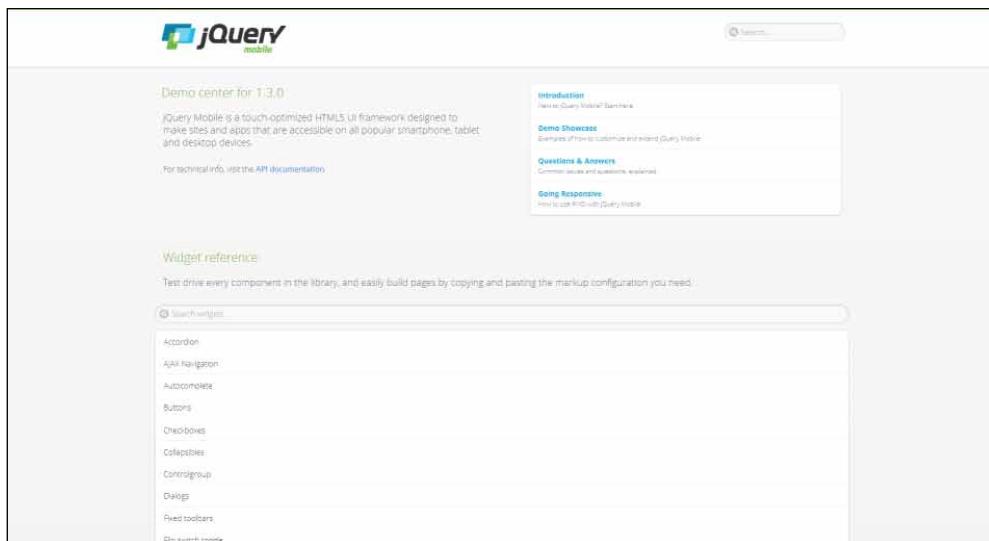


Рис. 12.7. Внешний вид приложения, созданного с помощью jQuery Mobile API и Docs, на настольном компьютере

Это же относится и к примерам приложений, в которых возможности пользовательского интерфейса расширены с помощью фреймворка jQuery Mobile (рис. 12.8).

Аккордеоны, календари, слайдеры и другие элементы пользовательского интерфейса повторно используют функционал, который библиотека jQM предоставляет пользователям мобильных устройств. Например, атрибут `data-mini="true"` придаст мобильным виджетам более элегантный вид на «настольном» браузере.

Более подробную информацию о мини-виджетах для «настольных» приложений вы можете найти в материалах, посвященных jQuery Mobile.

Благодаря некоторым медиазапросам «настольный» пользовательский интерфейс способен оптимально использовать свободное пространство, расширять блоки компонентов и применять альтернативные макеты, при этом продолжая использовать jQM в качестве компонентного фреймворка.

Преимущество такого интерфейса в том, что для получения этих возможностей не нужно задействовать отдельный дополнительный фреймворк виджетов (например, jQuery UI). Благодаря ThemeRoller компоненты могут иметь желаемый внешний вид; в то же время пользователи приложения, работающие в системе с низким разрешением экрана, получают jQM-интерфейс и jQM-подобный интерфейс во всех остальных случаях.

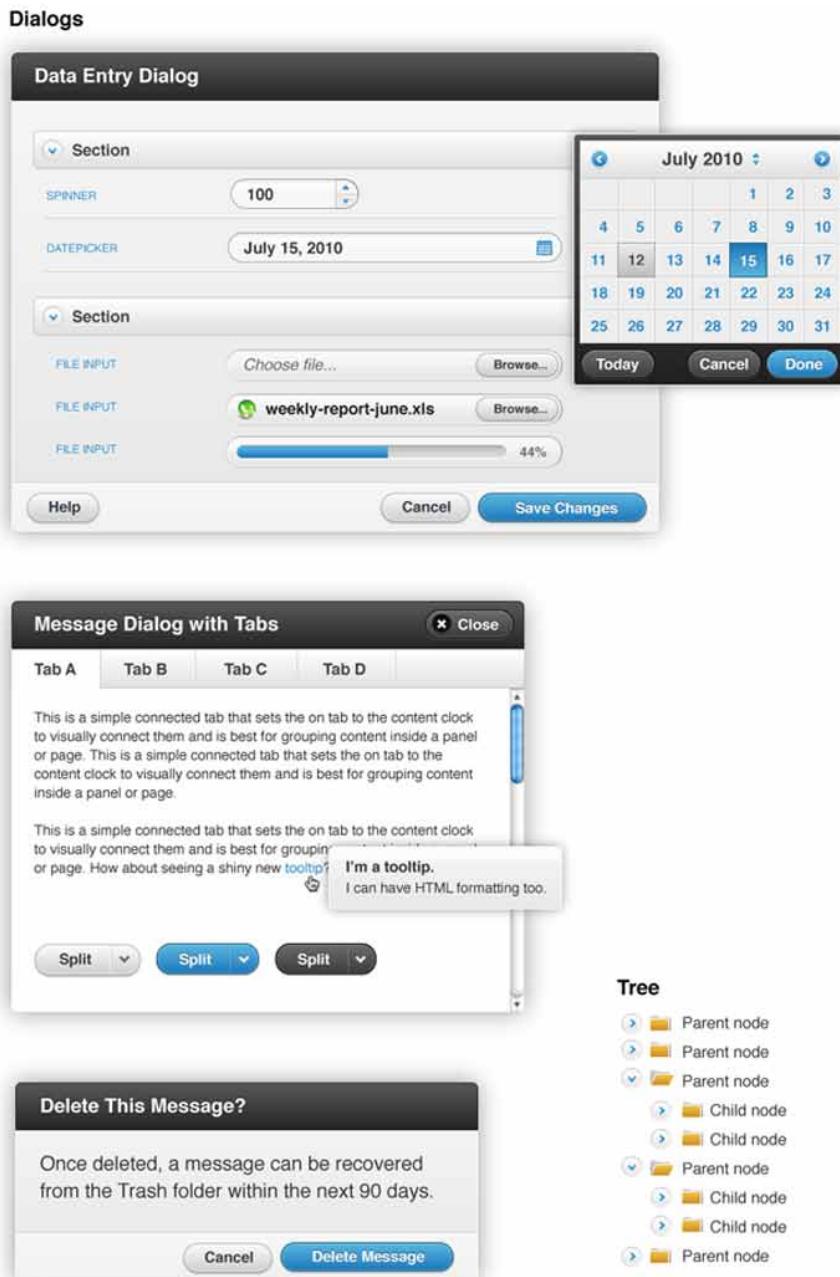


Рис. 12.8. Пример интерфейса на основе jQuery Mobile для настольной системы

Резюме: если вы уже не «бьетесь» над приложением, в котором загрузка сценариев и стилей выполняется в зависимости от разрешения экрана (с помощью matchMedia.js и других средств), воспользуйтесь более простым решением для придания компонентам желаемого вида в зависимости от целевого устройства. Всегда есть смысл воспользоваться API адаптивного дизайна фреймворка jQuery Mobile, который имеется в версиях 1.3.0 и старше, поскольку он работает как на мобильных, так и на стационарных устройствах.

Таким образом вы сможете управлять компонентами jQuery Mobile так, что пользователи настольных систем не заметят разницы. Дополнительную информацию об адаптивном дизайне с использованием jQuery Mobile можно получить по адресу <http://view.jquerymobile.com/1.3.0/docs/intro/rwd.php>.

Если вы достигнете пределов в использовании стилей CSS и конфигурировании компонентов jQuery Mobile для настольных систем в вашем приложении, то всегда есть возможность с минимальными дополнительными усилиями воспользоваться jQuery Mobile совместно с Twitter Bootstrap. Если «настольный» браузер запрашивает страницу при загруженном Twitter Bootstrap, то мобильному приложению TodoMVC понадобится условный код, который отключит API прогрессивного улучшения виджетов фреймворка jQM (показанный в разделе «Динамическое изменение DOM») в реализации Backbone.View. Таким образом, как было объяснено в предыдущих разделах, мы рекомендуем генерировать улучшения виджетов с помощью метода `$.mobile.changePage` только один раз для загрузки полной страницы.

На рис. 12.9 показан пример такого гибридного использования виджетов.

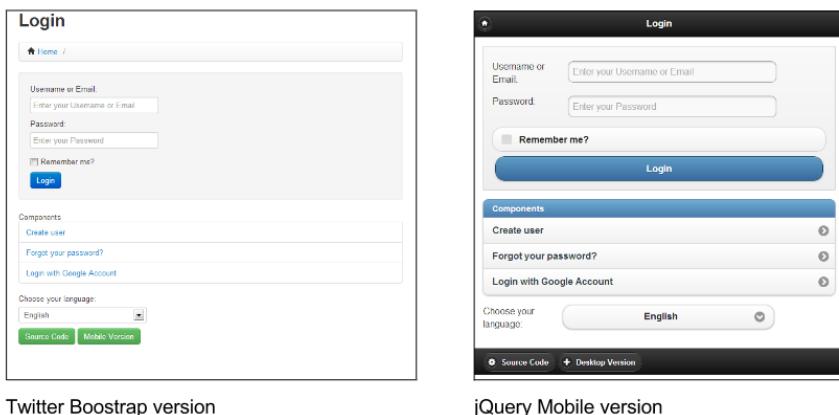


Рис. 12.9. Внешний вид заготовки приложения на настольной и мобильной системе

Несмотря на то что здесь используются серверные технологии шаблонизации с использованием языка программирования Python, принципы активизации прогрессивного улучшения виджетов во время загрузки страницы те же, что и в методе `$mobile.changePage`.

Как видите, JavaScript и даже таблицы стилей остаются неизменными. Различия и специфика этих реализаций в отношении устройств заключается в импортировании надлежащих фреймворков в HTML-шаблоне:

```
...
  {% if is_mobile %}
    <link rel="stylesheet" href="/mobile/jquery.mobile-1.1.0.min.css" />
  {% else %}
    <link rel="apple-touch-icon" href="/apple-touch-icon.png" />
    <link rel="stylesheet" href="/css/style.css" />
    <link rel="stylesheet" href="/css/bootstrap.min.css">
    <link rel="stylesheet" href="/css/bootstrap-responsive.min.css">
  {% endif %}
    <link rel="stylesheet" href="/css/main.css" />
  {% block mediaCSS %}{% endblock %}
...
  {% if is_mobile %}
    <script src="/mobile/jquery.mobile-1.1.0.min.js"></script>
  {% else %}
    <script src="/js/libs/bootstrap.min.js"></script>
  {% endif %}
...
...
```

13 **Jasmine**

Одно из определений гласит: *модульное тестирование* — это выделение в приложении минимального фрагмента кода, который можно протестировать, его изоляция от остальной кодовой базы и проверка корректности его выполнения.

Чтобы хорошо протестировать приложение, для каждой его функции нужно создать отдельные модульные тесты, проверяющие функцию в различных рабочих условиях. Прежде чем функционал объявляется готовым, он должен успешно пройти все тесты. Это дает возможность разработчикам модифицировать элементы кода и их зависимости и узнавать о наличии или отсутствии ошибок во внесенных изменениях.

Простейший пример модульного тестирования: разработчик хочет убедиться в том, что передача определенных значений в функцию суммирования приводит к правильному результату. Еще один пример, который имеет отношение к теме этой книги: проверка, что при добавлении новой задачи в список в коллекции задач корректно создается модель определенного типа.

При создании современных веб-приложений рекомендуется выполнять автоматическое модульное тестирование в процессе разработки. В следующих главах мы рассмотрим три различных решения для модульного тестирования Backbone-приложений: *Jasmine*, *QUnit* и *SinonJS*.

Разработка через реализацию поведения

В этой главе мы изучим модульное тестирование Backbone-приложений с помощью популярного тестового фреймворка для JavaScript под названием *Jasmine*, разработанного компанией Pivotal Labs.

Согласно собственному описанию, *Jasmine* — это фреймворк, предназначенный для тестирования JavaScript-кода и основанный на принципе «разработка через

реализацию поведения» (behavior-driven development, BDD). Перед тем как мы погрузимся в изучение этого фреймворка, давайте разберемся, что такое BDD.

BDD – это подход второго поколения к тестированию, впервые описанный экспертом Дэном Нортом (Dan North) и сконцентрированный на проверке поведения программного обеспечения. Термин «второе поколение» означает, что этот подход появился в результате слияния идей проблемно-ориентированного проектирования (domain-driven design, DDD) и бережливой разработки программного обеспечения (lean software development). BDD помогает командам разработчиков создавать высококачественное ПО, отвечая на самые нетривиальные вопросы на ранних стадиях гибкого процесса. Как правило, часть этих вопросов касается документирования и тестирования.

Если вы читали какую-нибудь книгу о BDD, то, скорее всего, в ней BDD характеризуется терминами «изнутри наружу» и «запросно-ориентированный». Дело в том, что BDD заимствовал идею запроса функций из бережливого производства, в котором эффективная разработка требуемых программ достигается за счет следующих принципов:

- концентрации на требуемых от системы результатах;
- гарантировании достижения этих результатов.

BDD учитывает, что в проекте существует много сторон, а не только абстрактный пользователь системы. Различные группы людей используют разные подходы к разрабатываемому ПО и вкладывают различный смысл в понятие качества системы. По этой причине важно четко представлять себе адресную аудиторию ПО и ее конкретные запросы.

Наконец, в BDD применяется автоматизация. Как только вы определитесь с качеством решения, возникнет необходимость в проверке разрабатываемого функционала и сравнения его работы с ожидаемыми результатами. Чтобы этот процесс был эффективным, его необходимо автоматизировать. BDD в значительной степени опирается на автоматическое тестирование соблюдения спецификаций, и инструмент Jasmine способствует выполнению этой задачи.

BDD помогает разработчикам и нетехническим участникам проекта:

- лучше понимать и представлять модели решаемых задач;
- объяснять тестовые сценарии языком, понятным непрограммистам;
- сводить к минимуму перевод между техническим языком программного кода и языком предметной области, на котором говорят бизнес-специалисты.

Другими словами, разработчики должны иметь возможность показать модульные тесты Jasmine участнику проекта, который благодаря общей терминологической базе сможет в общих чертах понять, что делает код.

Разработчики часто реализуют BDD одновременно с другой парадигмой — разработкой через тестирование (test-driven development, TDD). Основная идея TDD состоит в использовании следующего процесса разработки:

1. Создание модульных тестов, описывающих функциональность, которую должен обеспечивать код.
2. Регистрация случаев неудачного выполнения этих тестов (из-за того, что не написан соответствующий код).
3. Написание кода, который успешно проходит тестирование.
4. Расширение функционала приложения путем повторения этих действий.

В этой главе мы будем разрабатывать модульные тесты для Backbone-приложения с помощью BDD и TDD.



Многие разработчики также создают тесты, проверяющие поведение кода, после написания самого кода. Хотя такой подход работает, у него могут быть изъяны: например, может оказаться, что он тестирует только функции, которые поддерживает имеющийся код, а не весь функционал, требуемый для решения задачи.

Тестовые наборы, спецификации и агенты

При использовании Jasmine вы будете создавать тестовые наборы и спецификации. Наборы описывают ситуации, а спецификации — действия, которые можно выполнить в этих ситуациях.

Каждая спецификация представляет собой JavaScript-функцию, описываемую вызовом `it()`, которому передается строка описания и функция. Описание должно определять поведение некоторого фрагмента кода и (принимая во внимание принципы BDD) быть выразительным. Вот простой пример спецификации:

```
it('should be incrementing in value', function(){
  var counter = 0;
  counter++;
});
```

Сама по себе спецификация не особо полезна, пока не определено ожидаемое поведение кода. Ожидания задаются в спецификациях с помощью функции `expect()` и обнаружителя совпадений — например, `toEqual()`, `toBeTruthy()`, `toContain()`. Пример с обнаружителем совпадений выглядит так:

```
it('should be incrementing in value', function(){
    var counter = 0;
    counter++;
    expect(counter).toEqual(1);
});
```

В этом коде задано ожидание равенства переменной `counter` числу 1. Обратите внимание, как легко читается последняя строка — скорее всего, вы поняли ее смысл без всяких пояснений.

Спецификации группируются в наборы, которые описываются Jasmine-функцией `describe()`; в нее, как и в вызов `it()`, передается строка с описанием и функция. Имя/описание набора, как правило, относится к тестируемому компоненту или модулю.

Библиотека Jasmine использует описание в качестве имени группы при составлении отчета о запущенных спецификациях. Простой набор, содержащий нашу спецификацию-пример, выглядит следующим образом:

```
describe('Stats', function(){
    it('can increment a number', function(){
        ...
    });
    it('can subtract a number', function(){
        ...
    });
});
```

Наборы совместно используют область видимости функций, поэтому можно объявлять внутри блока `describe` переменные и функции, доступные внутри спецификаций:

```
describe('Stats', function(){
    var counter = 1;
    it('can increment a number', function(){
        // значение counter было равно 1
        counter = counter + 1;
        expect(counter).toEqual(2);
    });
    it('can subtract a number', function(){
        // значение counter было равно 2
        counter = counter - 1;
        expect(counter).toEqual(1);
    });
});
```



Наборы исполняются в порядке их описания; этот принцип важно иметь в виду, если вы хотите, чтобы результаты тестирования определенных компонентов вашего приложения были помещены в начало отчета.

Jasmine также поддерживает *агентов*, которые обеспечивают «заглушки», имитируют действия и поставляют информацию о поведении приложения при модульном тестировании. Агенты заменяют собой функцию, за которой они «следят», и позволяют симулировать ее поведение, то есть проводить тестирование в условиях, когда функция фактически не реализована.

В следующем примере мы «следим» за методом `setComplete` «заглушки» `Todo`, чтобы проверить, что ему можно корректно передать аргументы.

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

Todo.prototype.setComplete = function (arg){
    return arg;
}

describe('a simple spy', function(){
    it('should spy on an instance method of a Todo', function(){
        var myTodo = new Todo();
        spyOn(myTodo, 'setComplete');
        myTodo.setComplete('foo bar');
        expect(myTodo.setComplete).toHaveBeenCalledWith('foo bar');
        var myTodo2 = new Todo();
        spyOn(myTodo2, 'setComplete');
        expect(myTodo2.setComplete).not.toHaveBeenCalled();
    });
});
```

Вы, скорее всего, будете использовать агентов при тестировании асинхронных действий приложения, таких как AJAX-запросы. Jasmine поддерживает следующие возможности:

- написание тестов, имитирующих AJAX-запросы, с помощью агентов. Это позволяет тестировать как код, инициирующий AJAX-запрос, так и код, выполняемый после обработки запроса. Можно также имитировать реакцию сервера. Преимущество такого тестирования в быстроте, поскольку не происходит реальных обращений к серверу. Возможность имитировать любую реакцию сервера также представляет значительную ценность;
- асинхронные тесты, не использующие агентов.

Пример теста первого типа демонстрирует, как сымитировать AJAX-запрос и проверить, что запрос обратился к правильному URL и выполнил обратный вызов там, где он был указан:

```
it('the callback should be executed on success', function () {
    // `andCallFake()` вызывает переданную функцию при
    // вызове агента
    spyOn($, 'ajax').andCallFake(function(options) {
        options.success();
    });
});
```

продолжение ↗

```

// создание нового агента
var callback = jasmine.createSpy();
// исполнение обратного вызова агента, если
// запрос к задаче 15 успешен
getTodo(15, callback);
// проверка, что URL последнего вызова
// соответствует нужной задаче.
expect($.ajax.mostRecentCall.args[0]['url']).toEqual('/todos/15');
// `expect(x).toHaveBeenCalled()` будет успешно пройден, если
// `x` - это агент, который был вызван.
expect(callback).toHaveBeenCalled();
});

function getTodo(id, callback) {
  $.ajax({
    type: 'GET',
    url: '/todos/' + id,
    dataType: 'json',
    success: callback
  });
}

```

Все обнаружители совпадений, использованные в агентах в этом примере, документированы на вики-странице [Jasmine](#).

В teste второго (асинхронного) типа мы идем дальше и используем три других метода, поддерживаемых [Jasmine](#) (в соответствии с документацией на GitHub):

waits(timeout)

Штатный тайм-аут, после которого запускается следующий блок.

waitsFor(function, optional message, optional timeout)

Задержка спецификации до окончания другого действия. Здесь [Jasmine](#) ждет возврата из указанной функции, а затем переходит к следующему блоку.

runs(function)

Выполнение блока так, как если бы он был вызван напрямую. Этот метод позволяет тестировать асинхронные процессы.

```

it('should make an actual AJAX request to a server', function () {
  // создание нового агента
  var callback = jasmine.createSpy();
  // выполнение обратного вызова агента, если
  // запрос к задаче 16 был успешен
  getTodo(16, callback);
  // приостановить спецификацию, пока счетчик обратных вызовов
  // не станет положительным
  waitsFor(function() {
    return callback.callCount > 0;
  });
  // по завершении ожидания блок runs()
}

```

```
// проверит, был ли выполнен обратный вызов
// нашего агента
runs(function() {
    expect(callback).toHaveBeenCalled();
});
});

function getTodo(id, callback) {
$.ajax({
    type: 'GET',
    url: 'todos.json',
    dataType: 'json',
    success: callback
});
}
```



Помните, что реальные запросы к веб-серверу при модульном тестировании могут существенно замедлить выполнение тестов (в силу многих факторов, в том числе задержки обработки запросов сервером). Поскольку при таком подходе модульное тестирование приобретает внешнюю зависимость, которую можно (и следует) минимизировать, настоятельно рекомендуется пользоваться агентами, не зависящими от веб-сервера.

beforeEach() и afterEach()

Фреймворк Jasmine также позволяет задавать код, который выполняется до и после каждого теста, с помощью функций соответственно `beforeEach()` и `afterEach()`. В следующем примере функция `beforeEach` создает новую модель задачи, которую спецификации могут использовать для тестирования.

```
beforeEach(function(){
    this.todo = new Backbone.Model({
        text: 'Buy some more groceries',
        done: false
    });
});
it('should contain a text value if not the default value', function(){
    expect(this.todo.get('text')).toEqual('Buy some more groceries');
});
```

Каждый вложенный вызов `describe()` в тестах имеет свои собственные методы `beforeEach()` и `afterEach()`, обеспечивающие создание и удаление ресурсов, относящиеся к определенному тестовому набору.

Функции `beforeEach()` и `afterEach()` используются совместно для создания тестов, проверяющих, что маршруты Backbone корректно срабатывают при переходе к определенному URL. Начнем с действия `index`:

```

describe('Todo routes', function(){
  beforeEach(function(){
    // создание нового маршрутизатора
    this.router = new App.TodoRouter();
    // создание нового агента
    beforeEach() and afterEach() | 271
    this.routerSpy = jasmine.spy();
    // начало наблюдения за событиями изменения локального URL
    try{
      Backbone.history.start({
        silent:true,
        pushState: true
      });
    }catch(e){
      // ...
    }
    // навигация к URL
    this.router.navigate('/js/spec/SpecRunner.html');
  });
  afterEach(function(){
    // обратная навигация к URL
    this.router.navigate('/js/spec/SpecRunner.html');
    // временное запрещение журнала Backbone.history.
    // это не слишком полезно в реальных приложениях,
    // но хорошо подходит для тестирования маршрутизаторов
    Backbone.history.stop();
  });
  it('should call the index route correctly', function(){
    this.router.bind('route:index', this.routerSpy, this);
    this.router.navigate('', {trigger: true});
    // если все, что указано в вызовах beforeEach() и afterEach()
    // выполнено корректно, то
    // следующие проверки должны пройти успешно.
    expect(this.routerSpy).toHaveBeenCalledWithOnce();
    expect(this.routerSpy).toHaveBeenCalledWith();
  });
});

```

Настоящий маршрутизатор TodoRouter, для которого написан предыдущий тест, выглядит следующим образом:

```

var App = App || {};
App.TodoRouter = Backbone.Router.extend({
  routes:{
    '' : 'index'
  },
  index: function(){
    //...
  }
});

```

Общая область видимости

Представим, что у нас есть тестовый набор, в котором мы хотим проверить существование нового экземпляра задачи. Сделаем это, продублировав спецификацию следующим образом:

```
describe("Todo tests", function(){
    // спецификация
    it("Should be defined when we create it", function(){
        // задача, которую мы тестируем
        var todo = new Todo("Get the milk", "Tuesday");
        expect(todo).toBeDefined();
    });
    it("Should have the correct title", function(){
        // здесь мы дублируем код
        var todo = new Todo("Get the milk", "Tuesday");
        expect(todo.title).toBe("Get the milk");
    });
});
```

Ка видите, мы дублировали код, которому в идеале следует придать более компактную структуру. Для этого воспользуемся *функциональной областью видимости* набора Jasmine.

Все спецификации набора имеют *общую функциональную область видимости*, то есть переменные, объявленные в самом наборе, доступны во всех спецификациях этого набора. Это позволяет избавиться от дублирования путем перемещения действий по созданию задач в общую функциональную область видимости:

```
describe("Todo tests", function(){
    // экземпляр задачи, который мы хотим протестировать,
    // теперь находится в общей функциональной области видимости
    var todo = new Todo("Get the milk", "Tuesday");
    // спецификация
    it("should be correctly defined", function(){
        expect(todo).toBeDefined();
    });
    it("should have the correct title", function(){
        expect(todo.title).toBe("Get the milk");
    });
});
```

Вы могли заметить, что в предыдущем примере мы сначала объявили `this.todo` в области видимости вызова `beforeEach()`, а затем продолжали использовать эту ссылку в `afterEach()`.

Это также относится к общей функциональной области видимости, которая позволяет создавать объявления, доступные во всех блоках (в том числе `runs()`).

Переменные, объявленные вне общей области видимости (в локальной области `var todo=...`), не являются общедоступными.

Подготовка к тестированию

Итак, мы изучили основы фреймворка `Jasmine`, а теперь займемся скачиванием и установкой инструментов, необходимых для написания тестов.

Автономную версию фреймворка `Jasmine` можно скачать с официальной страницы с его релизами.

Помимо релиза вам понадобится файл `SpecRunner.html`. Его можно получить на [GitHub](#) или из полного репозитория `Jasmine`. Кроме того, полный репозиторий `Jasmine` можно скачать с [GitHub](#) с помощью команды `git clone`.

Рассмотрим файл `SpecRunner.html.jst`.

Вначале в нем выполняется включение `Jasmine` и CSS-таблиц, необходимых для создания отчетов:

```
<link rel="stylesheet" type="text/css"
      href="lib/jasmine-<%= jasmineVersion %>/jasmine.css">
<script src="lib/jasmine-<%= jasmineVersion %>/jasmine.js"></script>
<script src="lib/jasmine-<%= jasmineVersion %>/jasmine-html.js"></script>
<script src="lib/jasmine-<%= jasmineVersion %>/boot.js"></script>
```

Далее указывается тестируемый исходный код:

```
<!-- укажите здесь файлы с исходным кодом... -->
<script src="src/Player.js"></script>
<script src="src/Song.js"></script>
```

Затем следует несколько тестов-примеров:

```
<!-- укажите здесь файлы спецификаций... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
```



За этим разделом `SpecRunner` следует код, отвечающий за фактическое выполнение тестов.

Поскольку мы не будем изменять этот код, я не стану пояснять его содержание. Тем не менее рекомендую ознакомиться с содержимым файлов `PlayerSpec.js`

и SpecHelper.js. Это полезные базовые примеры, демонстрирующие работу минимальных тестовых наборов.

Также имейте в виду, что в некоторых вводных примерах этой главы будут тестироваться функции самой библиотеки Backbone.js, чтобы наглядно продемонстрировать работу Jasmine. Как правило, вам не нужно писать тесты, проверяющие корректность работы фреймворка.

Разработка через тестирование с использованием Backbone

При разработке приложений с помощью Backbone может возникнуть необходимость тестировать как отдельные модули кода, так и модели, представления, коллекции и маршрутизаторы. Воспользуемся TDD-подходом к тестированию и рассмотрим несколько спецификаций для тестирования этих компонентов Backbone на примере популярного Backbone-приложения для управления задачами.

Модели

Сложность Backbone-моделей вариируется в зависимости от целей приложения. В следующем примере мы протестируем значения по умолчанию, атрибуты, изменения состояний и правила валидации.

Начнем набор для тестирования моделей с функции describe():

```
describe('Tests for Todo', function() {
```

Атрибуты модели должны иметь значения по умолчанию. Если при создании экземпляра такой модели не указывается значение для какого-либо ее атрибута, то используется его значение по умолчанию (например, пустая строка). Смысл такого механизма — исключить некорректные действия при взаимодействии приложения с моделями.

В следующей спецификации создается новая задача без передачи ей каких-либо атрибутов, а затем проверяется значение атрибута text. Поскольку ему не задано никакое значение, при считывании мы должны получить значение по умолчанию:

```
it('Can be created with default values for its attributes.', function() {
  var todo = new Todo();
  expect(todo.get('text')).toBe('');
});
```

Если вы протестируете эту спецификацию раньше, чем создадите модели, то тест, как и следует ожидать, завершится неудачно. Чтобы тест прошел успешно, спецификации требуется значение атрибута `text` по умолчанию. Зададим его и несколько других полезных значений по умолчанию (которыми мы вскоре воспользуемся) в модели задачи следующим образом:

```
window.Todo = Backbone.Model.extend({
  defaults: {
    text: '',
    done: false,
    order: 0
  }
})
```

Далее, в модели включают валидационную логику для проверки корректности данных, вводимых пользователями или получаемых от других модулей приложения.

В приложении, управляющем задачами, может потребоваться проверка наличия нецензурной лексики во введенном тексте. Если признак завершения задачи хранится в виде булевой величины, то нам необходимо убедиться, что передаваемые значения имеют логический тип, а не являются произвольными строками.

В следующей спецификации воспользуемся тем, что в случае неуспешной валидации метода `model.validate()` генерируется событие-ошибка. Оно дает возможность проверить, что валидация действительно завершается неудачей при вводе некорректных данных.

Создадим агент `errorCallback` встроенным методом `createSpy()` фреймворка `Jasmine`, позволяющим наблюдать за событием-ошибкой, следующим образом:

```
it('Can contain custom validation rules, and will trigger an invalid event on failed validation.', function() {
  var errorCallback = jasmine.createSpy('-invalid event callback-');
  var todo = new Todo();
  todo.on('invalid', errorCallback);
  // что необходимо указать в свойствах задачи,
  // чтобы валидация завершилась неудачно?
  todo.set({done: 'a non-boolean value'});
  var errorArgs = errorCallback.mostRecentCall.args;
  expect(errorArgs).toBeDefined();
  expect(errorArgs[0]).toBe(todo);
  expect(errorArgs[1]).toBe('Todo.done must be a boolean value.');
});
```

Код, с помощью которого предыдущий неуспешный тест поддерживает валидацию, довольно прост. Мы перегружаем метод `validate()` модели (согласно рекомендациям документации `Backbone`) и проверяем, что у модели есть свойство

done, которое имеет корректное логическое значение, прежде чем разрешаем его передачу.

```
validate: function(attrs) {
  if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
    return 'Todo.done must be a boolean value.';
  }
}
```

Ниже приведен окончательный код модели задачи:

```
window.Todo = Backbone.Model.extend({
  defaults: {
    text: '',
    done: false,
    order: 0
  },
  initialize: function() {
    this.set({text: this.get('text')}, {silent: true});
  },
  validate: function(attrs) {
    if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
      return 'Todo.done must be a boolean value.';
    }
  },
  toggle: function() {
    this.save({done: !this.get('done')});
  }
});
```

Коллекции

Теперь определим спецификации для тестирования Backbone-коллекции задач (TodoList). Коллекции отвечают за ряд действий над списком задач, в том числе за сортировку и фильтрацию.

Вот несколько отдельных спецификаций, которые можно использовать в работе с коллекциями:

- проверка возможности добавлять новые модели задач как в виде объектов, так и в виде массивов;
- проверка корректности значений атрибутов, например базового URL коллекции;
- намеренное добавление задач со статусом `done:true` и проверка количества задач, которые коллекция считает завершенными и незавершенными.

В этом разделе мы рассмотрим первые две спецификации, а третью предлагаем вам самостоятельно реализовать в качестве упражнения.

Проверить, что модели задач можно добавлять в коллекцию как объекты и массивы, несложно. Сначала инициализируем новую коллекцию `TodoList` и убедимся в том, что ее длина (то есть количество моделей, которые она содержит) равна 0. Затем добавляем новые задачи как объекты и как массивы, и на каждом шаге проверяем свойство `length` коллекции, чтобы убедиться в его правильности:

```
describe('Tests for TodoList', function() {
    it('Can add Model instances as objects and arrays.', function() {
        var todos = new TodoList();
        expect(todos.length).toBe(0);
        todos.add({ text: 'Clean the kitchen' });
        // сколько задач добавлено к текущему моменту?
        expect(todos.length).toBe(1);
        todos.add([
            { text: 'Do the laundry', done: true },
            { text: 'Go to the gym'}
        ]);
        // сколько всего существует задач?
        expect(todos.length).toBe(3);
    });
    ...
});
```

Протестировать атрибуты коллекции, как и атрибуты модели, совсем несложно. Вот спецификация, которая проверяет корректность URL коллекции (ссылки на расположение коллекции на сервере):

```
it('Can have a url property to define the basic url structure for all contained
models.', function() {
    var todos = new TodoList();
    // какой базовый URL задан в модели?
    expect(todos.url).toBe('/todos/');
});
```

Что касается третьей спецификации (которую вы создадите самостоятельно в качестве упражнения), то реализация нашей коллекции будет содержать методы `done()` и `remaining()`, фильтрующие завершенные и незавершенные задачи соответственно. Подумайте над написанием спецификации, создающей новую коллекцию и добавляющей в нее одну модель, в которой атрибут `done` имеет явно заданное значение `true`, и две модели, в которых атрибут `done` имеет значение по умолчанию (`false`). Проверка длин результатов, возвращаемых методами `done()` и `remaining()`, даст ответ на вопрос, работает ли управление состояниями задач правильно или же нуждается в корректировке.

Окончательная реализация коллекции `TodoList` выглядит следующим образом:

```
window.TodoList = Backbone.Collection.extend({
    model: Todo,
    url: '/todos/',
```

```
done: function() {
    return this.filter(function(todo) { return todo.get('done'); });
},
remaining: function() {
    return this.without.apply(this, this.done());
},
nextOrder: function() {
    if (!this.length) {
        return 1;
    }
    return this.last().get('order') + 1;
},
comparator: function(todo) {
    return todo.get('order');
}
});
```

Представления

Перед тем как изучать тестирование представлений Backbone, сначала кратко рассмотрим плагин jQuery, он поможет нам создавать для них Jasmine-спецификации.

Поскольку наше приложение, управляющее задачами, будет осуществлять манипуляции с DOM при помощи библиотеки jQuery, воспользуемся удобным jQuery-плагином под названием `jasmine-jquery`, чтобы упростить BDD-тестирование отображений, выполняемых представлениями.

Этот плагин предоставляет ряд дополнительных обнаружителей совпадений для Jasmine, позволяющих тестировать наборы, «обернутые» jQuery:

○ `toBe(jQuerySelector)`

Пример:

```
expect($('<div id="some-id"></div>')).toBe('div#some-id').
```

○ `toBeChecked()`

Пример:

```
expect($('<input type="checkbox" checked="checked"/>')).toBeChecked().
```

○ `toBeSelected()`

Пример:

```
expect($('<option selected="selected"></option>')).toBeSelected().
```

Плагин jasmine-jquery поддерживает и многие другие обнаружители совпадений, полный список которых можно найти на домашней странице проекта. Обратите внимание, что перечисленные обнаружители, как и стандартные обнаружители Jasmine, можно инвертировать с помощью префикса .not (например, expect(x).not.toBe(y)).

```
expect($('<div>I am an example</div>')).not.toHaveText(/other/)
```

Плагин jasmine-jquery также включает в себя модуль фикстур, с помощью которого можно загружать произвольное HTML-содержимое и использовать его в тестах.

Добавим HTML-код во внешний фикстурный файл some.fixture.html:

```
<div id="sample-fixture">some HTML content</div>
```

Затем загрузим его в реальный тест следующим образом:

```
loadFixtures('some.fixture.html')
$('#some-fixture').myTestedPlugin();
expect($('#some-fixture')).to<the rest of your matcher would go here>
```

По умолчанию плагин jasmine-jquery загружает фикстуры из директории spec/javascripts/fixtures. Чтобы сконфигурировать этот путь, воспользуйтесь конструкцией jasmine.getFixtures().fixturesPath = 'требуемый_путь'.

Наконец, плагин jasmine-jquery обеспечивает поддержку наблюдения за jQuery-событиями без каких-либо доводок. Для этого воспользуйтесь функциями spyOnEvent() и assert(eventName).toHaveBeenTriggered(selector). Например:

```
spyOnEvent($('#el'), 'click');
$('#el').click();
expect('click').toHaveBeenTriggeredOn($('#el'));
```

Тестирование представлений

В этом разделе мы изучим написание спецификаций для трех аспектов, касающихся представлений Backbone: начальной настройки, отображения представлений и работы с шаблонами. Два последних аспекта тестируются наиболее часто, однако скоро мы увидим причины, по которым полезно создавать спецификации и для инициализации представлений.

Начальная настройка

Спецификации для представлений Backbone должны в первую очередь проверять, что представления корректно привязаны к определенным элементам

DOM и базируются на корректных моделях данных. Такие спецификации способны выявлять проблемы, которые впоследствии приводят к ошибкам в более сложных тестах. Общая ценность этих спецификаций дополняется еще и тем, что их очень легко написать.

Чтобы создать целостные настройки для тестирования спецификаций, мы добавляем в DOM пустой элемент `` (#todoList) с помощью метода `beforeEach()` и инициализируем новый экземпляр `TodoView` пустой моделью задачи. Метод `afterEach()` удаляет предыдущий элемент #todoList `` и экземпляр представления.

```
describe('Tests for TodoView', function() {
  beforeEach(function() {
    $('body').append('<ul id="todoList"></ul>');
    this.todoView = new TodoView({ model: new Todo() });
  });
  afterEach(function() {
    this.todoView.remove();
    $('#todoList').remove();
  });
  ...
});
```

Первая спецификация, которую следует создать, проверяет, что созданное представление `TodoView` использует правильное имя тега `tagName` (имя элемента или класса). Цель этого теста — удостовериться, что в процессе создания представление было корректно привязано к элементу DOM.

Обычно представления Backbone при инициализации создают пустые элементы DOM; тем не менее эти элементы не привязываются к видимой DOM, чтобы их можно было сформировать, не затрачивая ресурсы на отображение.

```
it('Should be tied to a DOM element when created, based off the property provided.', function() {
  //какое имя тега у html-элемента, соответствующего этому представлению?
  expect(todoView.el.tagName.toLowerCase()).toBe('li');
});
```

Как и ранее, тестирование спецификации завершится неудачно, если представление `TodoView` не написано. К счастью, решить эту проблему просто: нужно создать новый экземпляр класса `Backbone.View` и задать в нем атрибут `tagName`.

```
var todoView = Backbone.View.extend({
  tagName: 'li'
});
```

Если вместо `tagName` вы захотите протестировать атрибут `className`, воспользуйтесь обнаружителем совпадений `toHaveClass()` плагина `jasmine-jquery`:

```
it('Should have a class of "todos"', function(){
  expect(this.view.$el).toHaveClass('todos');
});
```

Обнаружитель совпадений `toHaveClass()` работает с объектами `jQuery`, и если этот плагин не используется, то генерируется исключение. Если вы не пользуетесь `jasmine-jquery`, то проверить `className` также можно через доступ к элементу `el.className`.

Возможно, вы обратили внимание, что в методе `beforeEach()` мы передали нашему начальному (хотя и незаполненному) представлению модель задачи. В основе представления должен быть экземпляр модели, который представляет данные. Поскольку это существенно влияет на работоспособность нашего представления, мы можем написать спецификацию, которая проверяет, определена ли модель (воспользовавшись для этого обнаружителем совпадений `toBeDefined()`), а затем тестирует наличие корректных значений по умолчанию у атрибутов модели.

```
it('Is backed by a model instance, which provides the data.', function() {
  expect(todoView.model).toBeDefined();
  // какое значение здесь имеет Todo.get('done')?
  expect(todoView.model.get('done')).toBeFalsy();
});
```

Отображение представлений

Рассмотрим написание спецификаций для отображения представлений. С их помощью мы проверим, что элементы `TodoView` отображаются так, как нам требуется.

Разработчики, которые только начинают знакомиться с BDD, могут утверждать, что в небольших приложениях визуальная оценка отображения представлений способна заменить модульное тестирование. На самом деле, если вы создаете приложение, которое может разрастись до большого числа представлений, то имеет смысл в самом начале разработки максимально автоматизировать тестирование. Как мы скоро увидим, существуют и аспекты отображения, которые не поддаются проверке путем визуальной оценки экрана и требуют дополнительной верификации.

Начнем тестирование представлений с написания двух спецификаций. Первая спецификация проверит, что метод `render()` корректно возвращает экземпляр представления (это необходимо для формирования цепочек). Вторая спецификация проверит, что генерируемый HTML-код в точности соответствует свойствам экземпляра модели, связанный с представлением.

В отличие от некоторых спецификаций, рассмотренных ранее, в этом разделе метод `beforeEach()` будет использоваться чаще, чтобы продемонстрировать работу с вложенными тестовыми наборами и создать целостное множество условий

для наших спецификаций. В первом примере мы создадим модель-пример (на основе задачи), а затем — экземпляр представления TodoView с ее помощью.

```
describe('TodoView', function() {
  beforeEach(function() {
    this.model = new Backbone.Model({
      text: 'My Todo',
      order: 1,
      done: false
    });
    this.view = new TodoView({model:this.model});
  });
  describe('Rendering', function() {
    it('returns the view object', function() {
      expect(this.view.render()).toEqual(this.view);
    });
    it('produces the correct HTML', function() {
      this.view.render();
      // метод toContain() плагина jasmine-jquery позволяет
      // не тестировать все содержимое разметки задачи
      expect(this.view.el.innerHTML)
        .toContain('<label class="todo-content">My Todo</label>');
    });
  });
});
```

При запуске этих спецификаций неудачно завершается только вторая из них (которая проверяет корректность HTML-кода). Первая спецификация, проверяющая объект представления TodoView, возвращающий метод render(), завершается успешно, поскольку метод render(), который мы используем, предоставлен библиотекой Backbone по умолчанию, и мы еще не переопределили его.



Для удобочитаемости кода все примеры шаблонов в этом разделе будут использовать минимальный шаблон представления задачи. При необходимости польуйтесь его содержимым, приведенным ниже.

```
<div class="todo <%= done ? 'done' : '' %>">
  <div class="display">
    <input class="check" type="checkbox" <%= done ?
      'checked="checked"' : '' %> />
    <label class="todo-content"><%= text %></label>
    <span class="todo-destroy"></span>
  </div>
  <div class="edit">
    <input class="todo-input" type="text" value="<%= content %>" />
  </div>
</div>
```

Следующая спецификация завершается неудачно со следующим сообщением:

```
Expected '' to contain '<label class="todo-content">My Todo</label>'
```

Это объясняется тем, что метод `render()` по умолчанию не создает какую-либо разметку.

Теперь напишем замену для стандартного метода `render()`, чтобы устраниить эту ошибку:

```
render: function() {
  var template = '<label class="todo-content">+++PLACEHOLDER+++</label>';
  var output = template
    .replace('+++PLACEHOLDER+++', this.model.get('text'));
  this.$el.html(output);
  return this;
}
```

Предшествующий код определяет встроенный строковый шаблон и заменяет поля, обнаруженные в шаблоне, блоками `+++PLACEHOLDER+++` с соответствующими значениями из связанной с ним модели. Поскольку мы также возвращаем из метода экземпляра `TodoView`, первая спецификация завершается успешно.

Модульное тестирование невозможно обсуждать без фикстур. Фикстуры обычно содержат тестовые данные (например, HTML-код), которые загружаются по необходимости (локально или из внешнего файла) для тестирования модуля. До настоящего момента мы определяли ожидания jQuery на основе свойства `el` представления. В некоторых случаях это работает; но иногда необходимо отображать разметку в документ. Оптимальный способ сделать это — использовать фикстуры (еще одну возможность плагина `jasmine-jquery`).

Перепишем последнюю спецификацию с применением фикстур:

```
describe('TodoView', function() {
  beforeEach(function() {
    ...
    setFixtures('<ul class="todos"></ul>');
  });
  ...
  describe('Template', function() {
    beforeEach(function() {
      $('.todos').append(this.view.render().el);
    });
    it('has the correct text content', function() {
      expect($('.todos').find('.todo-content'))
        .toHaveText('My Todo');
    });
  });
});
```

В этой спецификации мы добавляем в фикстуру отображенную задачу. Затем задаем ожидания для этой фикстуры, что желательно при создании представления для элемента, уже присутствующего в DOM. Нам необходимо указать фикстуру и протестировать элемент `el`, корректно считав ожидаемый элемент при создании экземпляра представления.

Отображение с использованием шаблонизации

Если пользователь помечает задачу как завершенную, в качестве ответа желательно применить визуальный эффект (например, зачеркнуть текст задачи), чтобы ее можно было отличить от невыполненных задач. Для этого назначьте элементу задачи новый класс. Начнем с написания теста:

```
describe('When a todo is done', function() {
  beforeEach(function() {
    this.model.set({done: true}, {silent: true});
    $('.todos').append(this.view.render().el);
  });
  it('has a done class', function() {
    expect($('.todos .todo-content:first-child'))
      .toHaveClass('done');
  });
});
```

Этот тест завершится неудачно с выдачей следующего сообщения:

```
Expected '<label class="todo-content">My Todo</label>' to have class 'done'.
```

Исправим эту ошибку, изменив метод `render()`:

```
render: function() {
  var template = '<label class="todo-content">' +
    '<%= text %></label>';
  var output = template
    .replace('<%= text %>', this.model.get('text'));
  this.$el.html(output);
  if (this.model.get('done')) {
    this$('.todo-content').addClass('done');
  }
  return this;
}
```

Но такой код способен быстро стать громоздким. С ростом сложности шаблона и расширением его логики тестирование шаблона также усложнится. Этот процесс можно упростить с помощью современных шаблонизаторов, многие из которых хорошо зарекомендовали себя в работе со средствами тестирования вроде Jasmine.

JavaScript-шаблонизаторы, например *Handlebars*, *Mustache* и *Underscore* (поддерживающего микрошаблоны), позволяют использовать условную логику в строках шаблонов. Это означает, что мы можем встраивать в код шаблонов if/else-выражения и тернарные операции, которые по необходимости оцениваются, и тем самым мы получаем возможность строить еще более сложные шаблоны.

В данном примере воспользуемся микрошаблонами библиотеки Underscore.js, поскольку для этого не потребуются никакие дополнительные файлы, и мы сможем без больших усилий изменить существующие спецификации, чтобы задействовать в них микрошаблоны.

Пусть наш шаблон будет определен с помощью тега `<script>` с идентификатором `myTemplate`:

```
<script type="text/template" id="myTemplate">
  <div class="todo <%= done ? 'done' : '' %>">
    <div class="display">
      <input class="check" type="checkbox"
        <%= done ? 'checked="checked"' : '' %> />
      <label class="todo-content"><%= text %></label>
      <span class="todo-destroy"></span>
    </div>
    <div class="edit">
      <input class="todo-input" type="text" value="<%= content %>" />
    </div>
  </div>
</script>
```

Underscore-шаблоны можно использовать в представлении `TodoView` следующим образом:

```
var TodoView = Backbone.View.extend({
  tagName: 'li',
  template: _.template($('#myTemplate').html()),
  initialize: function(options) {
    // ...
  },
  render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
  },
  ...
});
```

Итак, что здесь происходит? Сначала мы определяем шаблон в теге `<script>` с особым типом (например, `type="text/template"`). Поскольку этот тип не распознается каждым браузером, он игнорируется; тем не менее обращение к этому сценарию по атрибуту `id` позволяет хранить шаблон отдельно от других частей страницы.

В нашем представлении используется Underscore-метод `_template()`, компилирующий шаблон в функцию, которой мы позже передадим данные модели. Вызов `this.model.toJSON()` возвращает копию атрибутов модели в виде JSON-строки методу `template`, создавая блок HTML-кода, который можно добавить в DOM.

Обратите внимание, что в идеале вся логика шаблона должна находиться за пределами спецификаций в отдельных файлах шаблонов или встраиваться посредством тегов `<script>` в файл SpecRunner. Обычно код с такой структурой легче поддерживать.

Если вы работаете с гораздо более компактными шаблонами и не пользуетесь этим подходом, я подскажу вам полезный трюк, позволяющий автоматически создавать или расширять шаблоны в общей функциональной области видимости Jasmine для каждого теста.

Мы создадим новую директорию (скажем, `templates`) в папке `spec` и включим в `SpecRunner.html` новый файл сценария с приведенным ниже содержанием; это позволит вручную добавлять собственные атрибуты для небольших шаблонов, которые мы хотим использовать:

```
: beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content">' +
      '<%= text %>' +
      '</label>'
  });
});
```

Наконец, мы обновляем существующую спецификацию, включая в нее обращение к шаблону при создании экземпляра `TodoView`:

```
describe('TodoView', function() {
  beforeEach(function() {
    ...
    this.view = new TodoView({
      model: this.model,
      template: this.templates.todo
    });
    ...
  });
});
```

При использовании этого подхода существующие спецификации, рассмотренные нами, по-прежнему успешно проходят тестирование, и теперь мы можем добавить в шаблон условную логику, работающую со статусом завершения задач:

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
```

продолжение ↗

```
    todo: '<label class="todo-content <%= done ? \'done\' : \'\' %>"' +
          '<%= text %>' +
          '</label>'
    });
});
});
```

Этот код тоже пройдет тест успешно, однако, как мы уже говорили, последний подход имеет смысл только тогда, когда вы работаете с небольшими и динамическими шаблонами.

Упражнение

В качестве упражнения я рекомендую изучить «коаны Jasmine» в папке `practicals\jasminekoans` и попробовать исправить специально сделанные ошибки, приводящие к неудачам при тестировании. Это отличный способ изучения спецификаций и тестовых наборов Jasmine; кроме того, работа над примерами «без шпаргалки» улучшит ваши навыки использования Backbone.

Дополнительная литература

- Джеймс Ньюберри. Тестирование Backbone-приложений с помощью SinonJS (James Newberry, Testing Backbone Apps with SinonJS);
- Крис Стром. Еще раз о Jasmine и Backbone.js (Chris Strom, Jasmine Backbone.js Revisited);
- Крис Стром. Phantom.js, Backbone.js и require.js (Chris Strom, Phantom.js and Backbone.js (and require.js)).

Заключение

Мы изучили написание Jasmine-тестов для моделей, коллекций и представлений библиотеки Backbone.js. Несмотря на то что иногда желательно тестировать маршрутизацию, разработчики предпочитают делать это с помощью сторонних инструментов, таких как Selenium.

14 QUnit

QUnit представляет собой тестовый комплект для языка JavaScript, который написан участником команды разработчиков jQuery Джерном Зефферером (Jörn Zaefferer) и используется для тестирования во многих крупных проектах с открытым исходным кодом (таких, как jQuery и Backbone.js). QUnit позволяет тестировать как стандартный JavaScript-код в браузере, так и код на стороне сервера, где используются такие среды, как Rhino, V8 и SpiderMonkey. Эти возможности делают QUnit прекрасным решением для широкого круга задач.

Моя личная рекомендация: сравните оба фреймворка и выберите тот, с которым вам удобнее работать.

Настройка

К счастью, подготовка QUnit к работе проста и занимает меньше пяти минут.

Сначала создадим тестовую среду, состоящую из трех файлов:

- HTML-структура для отображения результатов тестирования.
- Файл `qunit.js`, формирующий тестовый фреймворк.
- Файл `qunit.css` для стилевого оформления результатов тестирования.

Последние два файла можно скачать с сайта QUnit.

Для изучения комплекта QUnit вы можете воспользоваться онлайн-версией его исходных файлов; она доступна по адресу <https://github.com/jquery/qunit/>.

Пример HTML-кода с QUnit-совместимой разметкой

```
<!DOCTYPE html>
<html>
<head>
```

продолжение ➔

```

<title>QUnit Test Suite</title>
<link rel="stylesheet" href="qunit.css">
<script src="qunit.js"></script>
<!-- Ваше приложение -->
<script src="app.js"></script>
<!-- Ваши тесты -->
<script src="tests.js"></script>
</head>
<body>
    <h1 id="qunit-header">QUnit Test Suite</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests">test markup, hidden.</ol>
</body>
</html>

```

Рассмотрим элементы, идентификаторы которых содержат QUnit. Во время работы фреймворка QUnit происходит следующее:

- qunit-header отображает название тестового набора;
- qunit-banner отображается красным, если какой-либо тест не пройден, и зеленым, если все тесты пройдены;
- qunit-testrunner-toolbar содержит дополнительные параметры настройки отображения тестов;
- qunit-userAgent отображает свойство navigator.userAgent;
- qunit-tests является контейнером результатов тестов.

Корректно работающий исполнитель тестов выглядит так, как показано на рис. 14.1.



Рис. 14.1. Исполнитель тестов QUnit, выполняющий модульные тесты Backbone в браузере

Числа через запятую (а, б, в) после имени каждого теста соответствуют: а) количеству неудачных проверок; б) количеству удачных проверок; в) общему числу проверок. Щелчок по имени теста раскрывает информацию обо всех проверках соответствующего тестового варианта. Зеленым цветом отображены успешно пройденные проверки (рис. 14.2).

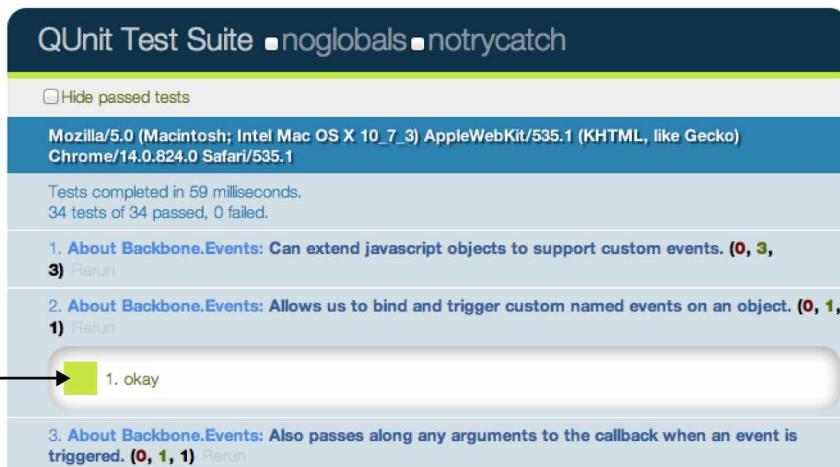


Рис. 14.2. Успешно пройденные проверки помечены зеленым маркером

Если какие-либо тесты завершаются неудачно, то соответствующий тест подсвечивается, а заголовок QUnit наверху становится красным, как показано на рис. 14.3.

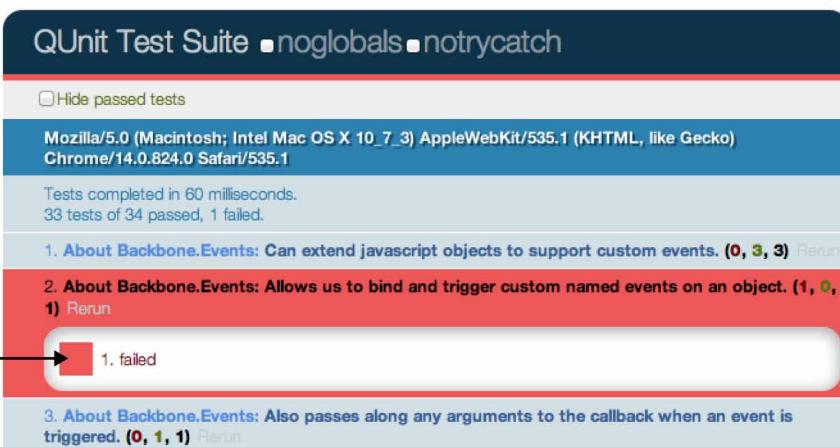


Рис. 14.3. Неудачно завершенные тесты отображаются в исполнителе тестов красным цветом

Операторы контроля

QUnit поддерживает набор базовых операторов контроля, которые используются в тестах для проверки правильности результата, возвращаемого кодом. Если оператор контроля генерирует ошибку, то программа действует некорректно. QUnit, как и Jasmine, позволяет легко выполнять регрессионное тестирование; иначе говоря, при некорректном поведении программы разработчик может написать оператор контроля, чтобы проверить наличие ошибки, затем написать исправление и опубликовать его вместе с проверкой. Если последующие изменения в коде приведут к провалу теста, то найти и устранить его причину будет проще.

В первую очередь мы рассмотрим следующие операторы контроля, поддерживаемые QUnit:

○ `ok(state, message)`

Завершается успешно, если первый аргумент истинен.

○ `equal(actual, expected, message)`

Простое сравнение с приведением типов.

○ `notEqual(actual, expected, message)`

Действует противоположно `equal()`.

○ `expect(amount)`

Количество операторов контроля, которые должны быть выполнены в каждом teste.

○ `strictEqual(actual, expected, message)`

Выполняет более строгое сравнение, чем `equal()`, и считается наилучшим методом проверки равенства, поскольку позволяет избежать нетривиальных ошибок при приведении типов.

○ `deepEqual(actual, expected, message)`

Аналогичен `strictEqual`; сравнивает содержимое заданных объектов, массивов и примитивов (с помощью `==`).

Простой тестовый вариант с использованием функции `test`

Создавать новые тестовые варианты с помощью QUnit несложно; для этого используется функция `test()` — она конструирует тест и принимает в качестве первого аргумента название, под которым будут отображены его результаты, а в качестве второго аргумента — функцию обратного вызова, содержащую операторы контроля. Она вызывается при запуске QUnit.

```
var myString = 'Hello Backbone.js';
test( 'Our first QUnit test - asserting results', function(){
    // ok( boolean, message )
    ok( true, 'the test succeeds');
    ok( false, 'the test fails');
    // equal( actualValue, expectedValue, message )
    equal( myString, 'Hello Backbone.js', 'Expected value: Hello Backbone.js!' );
});
```

Здесь мы определяем переменную с заданным значением, а затем проверяем, соответствует ли значение этой переменной нашим ожиданиям. Для этого воспользуемся сравнительным оператором контроля `equal()`, первым аргументом которого является тестируемое значение, а вторым аргументом — ожидаемое значение. Мы также использовали оператор `ok()`, позволяющий легко выполнять сравнение с функциями и переменными, которые можно привести к логическому значению.



В этом тестовом варианте мы могли бы передать функции `test()` ожидаемое значение, задав количество операторов контроля, которые собираемся выполнить. Это можно сделать в виде вызова `test(name, [expected], test)`; или вручную задав ожидание в начале тестирующей функции, например, так: `expect(1)`. Я рекомендую вам выработать привычку всегда задавать количество операторов контроля, которые вы собираетесь выполнить. Более подробно об этом речь пойдет позже.

Сравнение фактического и ожидаемого результата функции

Поскольку протестировать простую статическую переменную очень просто, пойдем дальше и займемся тестированием функций. В следующем примере мы проверяем результат функции, которая инвертирует строку, с помощью операторов `equal()` и `notEqual()`:

```
function reverseString( str ){
    return str.split('').reverse().join('');
}
test( 'reverseString()', function() {
    expect( 5 );
    equal( reverseString('hello'), 'olleh', 'The value expected was olleh' );
    equal( reverseString('foobar'), 'raboof', 'The value expected was raboof' );
    equal( reverseString('world'), 'dlrow', 'The value expected was dlrow' );
    notEqual( reverseString('world'), 'dlroo', 'The value was expected to not be dlroo' );
    equal( reverseString('bubble'), 'double', 'The value expected was elbbub' );
})
```

Запустив эти тесты в исполнителе тестов QUnit (который появится после загрузки тестовой HTML-страницы), мы увидим, что четыре оператора контроля отрабатывают успешно, а последний — нет. Проверка строки 'double' завершается неудачей потому, что в ней намеренно сделана опечатка. Если в ваших проектах тест не проходит, а операторы контроля верны, то, скорее всего, вы обнаружили ошибку.

Структурирование операторов контроля

Если вы поместите все операторы контроля в один тестовый вариант, то работать с ним будет неудобно. К счастью, фреймворк QUnit позволяет четко структурировать блоки операторов контроля с помощью метода `module()`, который позволяет группировать тесты. Типичный подход к группировке тестов состоит в том, что в одну группу (модуль) помещают множество тестов, относящихся к определенному методу.

Простейшие модули QUnit

```
module( 'Module One' );
test( 'first test', function() {} );
test( 'another test', function() {} );
module( 'Module Two' );
test( 'second test', function() {} );
test( 'another test', function() {} );
module( 'Module Three' );
test( 'third test', function() {} );
test( 'another test', function() {} );
```

Усовершенствуем этот пример, добавив в него обратные вызовы `setup()` и `teardown()`, которые выполняются соответственно до и после каждого теста.

Использование методов `setup()` и `teardown()`

```
module( 'Module One', {
    setup: function() {
        // запустить до
    },
    teardown: function() {
        // запустить после
    }
});
test('first test', function() {
    // запустить первый тест
});
```

С помощью этих обратных вызовов можно определять (или очищать) любые компоненты, создаваемые для использования в одном или нескольких тестах. Как мы скоро увидим, этот механизм идеально подходит для определения в проекте новых экземпляров представлений, коллекций, моделей и маршрутизаторов, которыми можно пользоваться в различных тестах.

Создание и очистка объектов с помощью методов `setup()` и `teardown()`

```
// Определение простой модели и коллекции,
// соответствующих магазину и списку магазинов
var Store = Backbone.Model.extend({});  
var StoreList = Backbone.Collection.extend({
    model: Store,
    comparator: function( Store ) { return Store.get('name') }  
});  
// Определение группы тестов
module( 'StoreList sanity check', {
    setup: function() {
        this.list = new StoreList;
        this.list.add(new Store({ name: 'Costcutter' }));
        this.list.add(new Store({ name: 'Target' }));
        this.list.add(new Store({ name: 'Walmart' }));
        this.list.add(new Store({ name: 'Barnes & Noble' }));
    },
    teardown: function() {
        window.errors = null;
    }
});  
// Тестирование порядка добавленных элементов
test( 'test ordering', function() {
    expect( 1 );
    var expected = ['Barnes & Noble', 'Costcutter', 'Target', 'Walmart'];
    var actual = this.list.pluck('name');
    deepEqual( actual, expected, 'is maintained by comparator' );
});
```

Здесь список магазинов создается и сохраняется в методе `setup()`. Обратный вызов `teardown()` используется для очистки списка ошибок, который хранится в области видимости окна, но не нужен за ее пределами.

Примеры использования операторов контроля

Перед тем как двигаться дальше, рассмотрим несколько дополнительных примеров корректного использования различных операторов контроля QUnit при написании тестов:

○ equal

Сравнительный оператор контроля; он завершается успешно, если `actual == expected`.

```
test( 'equal', 2, function() {
    var actual = 6 - 5;
    equal( actual, true, 'passes as 1 == true' );
    equal( actual, 1, 'passes as 1 == 1' );
});
```

○ notEqual

Сравнительный оператор контроля; он завершается успешно, если `actual != expected`.

```
test( 'notEqual', 2, function() {
    var actual = 6 - 5;
    notEqual( actual, false, 'passes as 1 != false' );
    notEqual( actual, 0, 'passes as 1 != 0' );
});
```

○ strictEqual

Сравнительный оператор контроля; он завершается успешно, если `actual === expected`.

```
test( 'strictEqual', 2, function() {
    var actual = 6 - 5;
    strictEqual( actual, true, 'fails as 1 !== true' );
    strictEqual( actual, 1, 'passes as 1 === 1' );
});
```

○ notStrictEqual

Сравнительный оператор контроля; он завершается успешно, если `actual != expected`.

```
test('notStrictEqual', 2, function() {
    var actual = 6 - 5;
    notStrictEqual( actual, true, 'passes as 1 !== true' );
    notStrictEqual( actual, 1, 'fails as 1 === 1' );
});
```

○ deepEqual

Рекурсивный сравнительный оператор контроля. В отличие от `strictEqual()` он работает с объектами, массивами и примитивами.

```
test('deepEqual', 4, function() {
    var actual = {q: 'foo', t: 'bar'};
    var el = $('div');
```

```
var children = $('div').children();
equal( actual, {q: 'foo', t: 'bar'}, 'fails - objects are not equal
using equal()' );
deepEqual( actual, {q: 'foo', t: 'bar'}, 
'passes - objects are equal' );
equal( el, children, 'fails - jQuery objects are not the same' );
deepEqual(el, children, 'fails - objects not equivalent' );
});
```

○ notDeepEqual

Сравнительный оператор контроля; он возвращает результат, противоположный deepEqual.

```
test('notDeepEqual', 2, function() {
  var actual = {q: 'foo', t: 'bar'};
  notEqual( actual, {q: 'foo', t: 'bar'}, 'passes - objects are not equal' );
  notDeepEqual( actual, {q: 'foo', t: 'bar'}, 'fails - objects are
equivalent' );
});
```

○ raises

Этот оператор контроля проверяет, генерирует ли обратный вызов какие-либо исключения.

```
test('raises', 1, function() {
  raises(function() {
    throw new Error( 'Oh no! It` s an error!' );
  }, 'passes - an error was thrown inside our callback');
});
```

Фикстуры

Иногда необходимо создавать тесты, модифицирующие DOM. Удаление этих изменений между тестами может оказаться очень утомительным, однако фреймворк QUnit решает эту проблему с помощью `#qunit-fixture`:

```
<!DOCTYPE html>
<html>
<head>
  <title>QUnit Test</title>
  <link rel="stylesheet" href="qunit.css">
  <script src="qunit.js"></script>
  <script src="app.js"></script>
  <script src="tests.js"></script>
</head>
<body>
```

продолжение ➔

```

<h1 id="qunit-header">QUnit Test</h1>
<h2 id="qunit-banner"></h2>
<div id="qunit-testrunner-toolbar"></div>
<h2 id="qunit-userAgent"></h2>
<ol id="qunit-tests"></ol>
<div id="qunit-fixture"></div>
</body>
</html>

```

Мы можем поместить статическую разметку в фикстуру либо просто вставить/добавить в нее любые необходимые DOM-элементы. QUnit будет автоматически сбрасывать элемент `innerHTML` фикстуры в исходное значение после каждого теста. Если вы используете фреймворк jQuery, помните, что QUnit проверяет его доступность и по возможности пытается использовать элемент `$(el).html()`, что также приводит к очистке всех обработчиков событий jQuery.

Пример с фикстурами

Теперь изучим более полный пример использования фикстур. Большинство из нас привыкли пользоваться библиотекой jQuery для работы со списками, с помощью которых часто определяется разметка меню, сеток и различных других компонентов. Не исключено, что вы пользовались плагинами jQuery до того, как обрабатывали списки, поэтому полезно убедиться, что конечный (обработанный) вывод плагина соответствует вашим ожиданиям.

В следующем примере мы воспользуемся плагином `$.enumerate()`, написанным Беном Олменом (Ben Alman). Этот плагин позволяет добавлять в начало каждого элемента списка индекс, с помощью которого можно задать первый элемент списка. Ниже приведен фрагмент кода плагина и пример генерированного им вывода:

```

$.fn.enumerate = function( start ) {
    if ( typeof start !== 'undefined' ) {
        // Поскольку параметр `start` указан, выполнить перечисление
        // и вернуть начальный jQuery-объект, чтобы обеспечить возможность создания
        // цепочек.
        return this.each(function(i){
            $(this).prepend( '<b>' + ( i + start ) + '</b> ' );
        });
    } else {
        // Поскольку аргумент `start` отсутствует, выполнить чтение
        // и вернуть соответствующее значение из
        // первого выбранного элемента.
        var val = this.eq( 0 ).children( 'b' ).eq( 0 ).text();
        return Number( val );
    }
};

```

```
/*
<ul>
  <li>1. hello</li>
  <li>2. world</li>
  <li>3. i</li>
  <li>4. am</li>
  <li>5. foo</li>
</ul>
*/
```

Теперь напишем несколько тестов для этого плагина. Сначала зададим разметку для списка с примерами элементов в компоненте `qunit-fixture`:

```
<div id="qunit-fixture">
  <ul>
    <li>hello</li>
    <li>world</li>
    <li>i</li>
    <li>am</li>
    <li>foo</li>
  </ul>
</div>
```

Затем мы должны решить, что именно следует протестировать. Плагин `$.enumerate()` поддерживает несколько вариантов использования, в том числе следующие:

- *Не передано никаких аргументов*

```
$(el).enumerate()
```

- *Передан аргумент 0*

```
$(el).enumerate(0)
```

- *Передан аргумент 1*

```
$(el).enumerate(1)
```

Поскольку текстовое значение каждого элемента списка имеет формат `n. item-text`, и это необходимо только для сверки с ожидаемым выводом, мы можем получить доступ к содержимому с помощью элемента `$(el).eq(index).text()` (более подробную информацию об `.eq()` можно найти по адресу <http://api.jquery.com/eq/>).

Итак, далее приведены тестовые варианты:

```
module('jQuery#enumerate');
test( 'No arguments passed', 5, function() {
```

продолжение ↗

```

var items = $('#qunit-fixture li').enumerate(); // 0
equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});
test( '0 passed as an argument', 5, function() {
    var items = $('#qunit-fixture li').enumerate( 0 );
    equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
    equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
    equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
    equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
    equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});
test( '1 passed as an argument', 3, function() {
    var items = $('#qunit-fixture li').enumerate( 1 );
    equal( items.eq(0).text(), '1. hello', 'first item should have index 1' );
    equal( items.eq(1).text(), '2. world', 'second item should have index 2' );
    equal( items.eq(2).text(), '3. i', 'third item should have index 3' );
    equal( items.eq(3).text(), '4. am', 'fourth item should have index 4' );
    equal( items.eq(4).text(), '5. foo', 'fifth item should have index 5' );
});

```

Асинхронный код

Фреймворк QUnit, как и Jasmine, позволяет выполнять синхронные тесты с минимальными усилиями. А как обстоит дело с тестами, в которых требуются асинхронные обратные вызовы (например, при проверке трудоемких процессов, Ajax-запросов и т. п.)? При работе с асинхронным кодом мы не передаем фреймворку QUnit управление при запуске очередного теста, а даем ему указание остановиться и дождаться возможности продолжения тестирования.

Запомните: выполнение асинхронного кода без учета его особенностей может привести к некорректным операциям контроля в других тестах, поэтому с таким кодом следует обращаться очень осторожно. Создавать QUnit-тесты для асинхронного кода можно с помощью методов `start()` и `stop()`, которые программно задают точки запуска и останова в таких тестах. Ниже приведен простой пример:

```

test('An async test', function(){
    stop();
    expect( 1 );
    $.ajax({
        url: '/test',
        dataType: 'json',
        success: function( data ){
            deepEqual(data, {

```

```
        topic: 'hello',
        message: 'hi there!'
    });
ok(true, 'Asynchronous test passed!');
start();
}
});
});
```

jQuery-вызов `$.ajax()` подключается к тестируемому ресурсу и проверяет корректность возвращаемых данных. В этом примере метод `deepEqual()` позволяет сравнивать различные типы данных (такие, как объекты и массивы) и проверяет соответствие возвращаемых данных ожидаемому результату. Мы знаем, что наш Ajax-вызов является асинхронным, поэтому сначала вызываем метод `stop()`, затем запускаем код, выполняющий запрос, и в самом конце обратного вызова информируем QUnit о том, что можно продолжать выполнение других тестов.



Мы можем по своему усмотрению заменить метод `stop()` методами `test()` и `asyncTest()`. Это повышает читабельность кода тестовых наборов, которые одновременно содержат в себе как синхронные, так и асинхронные тесты.

Хотя такая структура подходит для многих вариантов использования, нет гарантии, что обратный вызов в запросе `$.ajax()` будет выполнен. Чтобы учесть это, еще раз воспользуйтесь методом `expect()` и задайте количество операторов контроля, которые вы ожидаете увидеть внутри нашего теста. Это хорошая мера предосторожности: если на момент окончания теста выполнено недостаточное число операторов контроля, то вы узнаете о том, что тестирование прошло некорректно, и сможете внести необходимые исправления.

15 SinonJS

Итак, мы почти готовы воспользоваться полученными знаниями и написать несколько QUnit-тестов для приложения, управляющего задачами.

Перед тем как начать, сделаем небольшое отступление. Возможно, вы обратили внимание, что фреймворк QUnit не поддерживает тестовые агенты — функции, которые фиксируют аргументы, исключения и значения, возвращаемые их вызовами. Обычно агенты применяются для проверки обратных вызовов и использования функций в тестируемом приложении. В тестовых фреймворках агенты обычно являются анонимными функциями или «обертывают» существующие функции.

Что такое SinonJS?

Для обеспечения поддержки агентов в QUnit воспользуемся имитационным фреймворком под названием *SinonJS*, написанным Кристианом Йохансеном (Christian Johansen). Мы также применим адаптер, обеспечивающий бесшовную (то есть с минимумом необходимых настроек) стыковку между SinonJS и QUnit. SinonJS полностью независим от тестовых фреймворков; его легко использовать с любым тестовым фреймворком, что идеально подходит для наших задач.

SinonJS поддерживает три функции, которыми мы воспользуемся при модульном тестировании приложения:

- анонимные агенты;
- наблюдение за существующими методами;
- богатый тестовый интерфейс.

Базовые агенты

Вызов `this.spy()` без аргументов создает анонимный агент; его можно сравнить с вызовом `jasmine.createSpy()`. Следующий пример демонстрирует основы использования SinonJS-агента:

```
test('should call all subscribers for a message exactly once', function () {
  var message = getUniqueString();
  var spy = this.spy();
  PubSub.subscribe( message, spy );
  PubSub.publishSync( message, 'Hello World' );
  ok( spy.calledOnce, 'the subscriber was called once' );
});
```

Наблюдение за существующими функциями

Воспользуемся методом `this.spy()` и для наблюдения за существующими функциями (такими, как метод `$.ajax` библиотеки `jQuery`), как показано в следующем примере. Наблюдение за существующей функцией никак не сказывается на ее работе, но мы получаем доступ к данным о ее вызовах, что полезно для тестирования.

```
test( 'should inspect the jQuery.getJSON usage of jQuery.ajax', function () {
  this.spy( jQuery, 'ajax' );
  jQuerygetJSON( '/todos/completed' );
  ok( jQuery.ajax.calledOnce );
  equals( jQuery.ajax.getCall(0).args[0].url, '/todos/completed' );
  equals( jQuery.ajax.getCall(0).args[0].dataType, 'json' );
});
```

Тестовый интерфейс

Фреймворк SinonJS содержит богатый тестовый интерфейс, позволяющий проверять, сколько раз и с какими аргументами вызывались агенты, а также значения этих аргументов. Полный перечень возможностей интерфейса SinonJS опубликован на сайте SinonJS.org; мы рассмотрим примеры, демонстрирующие наиболее употребительные возможности SinonJS.

Проверка аргументов: был ли агент вызван с заданным набором аргументов?

```
test( 'Should call a subscriber with standard matching': function () {
  var spy = sinon.spy();
  PubSub.subscribe( 'message', spy );
  PubSub.publishSync( 'message', { id: 45 } );
  assertTrue( spy.calledWith( { id: 45 } ) );
});
```

Более строгая проверка аргументов: был ли агент вызван как минимум один раз с набором аргументов, в точности соответствующим заданному?

```
test( 'Should call a subscriber with strict matching': function () {
  var spy = sinon.spy();
  PubSub.subscribe( 'message', spy );
  PubSub.publishSync( 'message', 'many', 'arguments' );
  PubSub.publishSync( 'message', 12, 34 );
  // Проходит успешно
  assertTrue( spy.calledWith('many') );
  // Проходит неуспешно
  assertTrue( spy.calledWithExactly( 'many' ) );
});
```

Проверка порядка вызовов: был ли агент вызван до или после другого агента?

```
test( 'Should call a subscriber and maintain call order': function () {
  var a = sinon.spy();
  var b = sinon.spy();
  PubSub.subscribe( 'message', a );
  PubSub.subscribe( 'event', b );
  PubSub.publishSync( 'message', { id: 45 } );
  PubSub.publishSync( 'event', [1, 2, 3] );
  assertTrue( a.calledBefore(b) );
  assertTrue( b.calledAfter(a) );
});
```

Проверка счетчика исполнений: был ли агент вызван заданное число раз?

```
test( 'Should call a subscriber and check call counts', function () {
  var message = getUniqueString();
  var spy = this.spy();
  PubSub.subscribe( message, spy );
  PubSub.publishSync( message, 'some payload' );
  // Проходит успешно, если агент вызван ровно один раз.
  ok( spy.calledOnce ); // также поддерживаются calledTwice и calledThrice
  // Количество зарегистрированных вызовов.
  equal( spy.callCount, 1 );
  // Прямая проверка аргументов вызова
  equals( spy.getCall(0).args[0], message );
});
```

Заглушки и мок-объекты

Фреймворк SinonJS также поддерживает две другие фишки — *заглушки* и *мок-объекты*. Они реализуют весь функционал агентского API, однако имеют и собственные дополнительные функции.

Заглушки

Заглушка позволяет заменить существующее поведение определенного метода. Заглушки очень полезны для имитации исключений и чаще всего используются при написании тестовых вариантов, в которых задействованы еще не написанные компоненты кода.

Давайте еще раз бегло рассмотрим Backbone-приложение для управления задачами, в котором есть модель задачи и коллекция TodoList. Допустим, мы хотим изолировать коллекцию TodoList и сымитировать модель задачи, чтобы проверить, как работает добавление новых моделей.

Чтобы понять, как действуют заглушки, представим себе, что модель еще не написана. Коллекция-оболочка, содержащая только ссылку на модель, выглядит следующим образом:

```
var TodoList = Backbone.Collection.extend({  
    model: Todo  
});  
// считаем, что это экземпляр нашей коллекции  
this.todoList;
```

Предположим, что наша коллекция самостоятельно создает экземпляры моделей; тогда для теста нам необходима заглушка конструктора модели. Создадим ее:

```
this.todoStub = sinon.stub( window, 'Todo' );
```

Эта строка создает заглушку метода Todo объекта window. Если мы создаем заглушку для постоянного объекта, то должны вернуть его в исходное состояние. Для этого используем метод `teardown()`, как показано ниже:

```
this.todoStub.restore();
```

Затем изменим значение, возвращаемое конструктором; для этого воспользуемся конструктором `Backbone.Model`. Хотя заглушка не является моделью задачи, она предоставляет нам реальную Backbone-модель.

```
setup: function() {  
    this.model = new Backbone.Model({  
.....        id: 2,  
            title: 'Hello world'  
    });  
    this.todoStub.returns( this.model );  
},
```

Вы могли подумать, что благодаря этому фрагменту кода коллекция `TodoList` всегда создает модель-заглушку задачи, но поскольку ссылка на эту модель уже присутствует в коллекции, необходимо заново задать ее свойство `model`:

```
this.todoList.model = Todo;
```

В результате при создании новых моделей задач коллекция `TodoList` будет возвращать желаемый стандартный экземпляр Backbone-модели. Это позволяет написать тест для добавления новых моделей следующим образом:

```
module( 'Should function when instantiated with model literals', {
  setup:function() {
    this.todoStub = sinon.stub(window, 'Todo');
    this.model = new Backbone.Model({
      id: 2,
      title: 'Hello world'
    });
    this.todoStub.returns(this.model);
    this.todos = new TodoList();
    // изменение ссылки для использования заглушки
    this.todos.model = Todo;
    // добавление модели
    this.todos.add({
      id: 2,
      title: 'Hello world'
    });
    teardown: function() {
      this.todoStub.restore();
    }
  });
  test('should add a model', function() {
    equal( this.todos.length, 1 );
  });
  test('should find a model by id', function() {
    equal( this.todos.get(5).get('id'), 5 );
  });
});
```

Мок-объекты

Мок-объекты фактически являются аналогами заглушек, однако имитируют весь API и поддерживают встроенные ожидания своего использования. Различие между мок-объектом и агентом в том, что ожидания для мок-объектов определены заранее, и если хотя бы одно из них не удовлетворено, то тест завершается неудачей.

Ниже приведен пример использования мок-объекта на базе библиотеки PubSubJS. Метод `clearTodo()` играет роль функции обратного вызова и проверяет собственное поведение с помощью мок-объектов.

```
test('should call all subscribers when exceptions', function () {
  var myAPI = { clearTodo: function () {} };
  var spy = this.spy();
  var mock = this.mock( myAPI );
  mock.expects( 'clearTodo' ).once().throws();
  PubSub.subscribe( 'message', myAPI.clearTodo );
  PubSub.subscribe( 'message', spy );
  PubSub.publishSync( 'message', undefined );
  mock.verify();
  ok( spy.calledOnce );
});
```

Упражнение

Теперь приступим к написанию тестов для приложения, управляющего задачами. Эти тесты приведены и сгруппированы по компонентам (моделям, коллекциям и т. п.). Обратите внимание на название теста, проверяемую логику и, что наиболее важно, на операторы контроля; это создаст у вас представление о том, как изученные нами возможности применяются в полноценном приложении.

Для лучшего ознакомления с материалами, которым посвящена эта глава, я рекомендую изучить «Коаны QUnit» в папке `practicals/qunit-koans` — это портированные в QUnit «Коаны Jasmine».



Если вы еще не изучали «коаны», то поясню: они представляют собой наборы модульных тестов для определенного тестового фреймворка, которые демонстрируют написание тестовых наборов для приложения, а также содержат ряд незавершенных тестов, которые можно дописать в качестве упражнения.

Модели

Тестирование наших моделей должно проверить следующие условия:

- новые экземпляры создаются с ожидаемыми значениями по умолчанию;
- атрибуты устанавливаются ичитываются корректно;
- изменения состояний моделей приводят к генерации всех соответствующих написанных нами событий;
- правила валидации применяются корректно.

```
module( 'About Backbone.Model' );
test('Can be created with default values for its attributes.', function() {
  expect( 3 );
  var todo = new Todo();
  equal( todo.get('text'), '' );
```

продолжение ↗

```

        equal( todo.get('done'), false );
        equal( todo.get('order'), 0 );
    });
test('Will set attributes on the model instance when created.', function() {
    expect( 1 );
    var todo = new Todo( { text: 'Get oil change for car.' } );
    equal( todo.get('text'), 'Get oil change for car.' );
});
test('Will call a custom initialize function on the model instance when
created.', function() {
    expect( 1 );
    var toot = new Todo
        ({ text: 'Stop monkeys from throwing their own crap!' });
    equal( toot.get('text'),
        'Stop monkeys from throwing their own rainbows!' );
});
test('Fires a custom event when the state changes.', function() {
    expect( 1 );
    var spy = this.spy();
    var todo = new Todo();
    todo.on( 'change', spy );
    // изменение состояния модели
    todo.set( { text: 'new text' } );
    ok( spy.calledOnce, 'A change event callback was correctly triggered' );
});
test('Can contain custom validation rules, and will trigger an invalid
event on failed validation.', function() {
    expect( 3 );
    var errorCallback = this.spy();
    var todo = new Todo();
    todo.on('invalid', errorCallback);
    // изменение состояния модели так, чтобы валидация оказалась неуспешной
    todo.set( { done: 'not a boolean' } );
    ok( errorCallback.called, 'A failed validation correctly triggered an
error' );
    notEqual( errorCallback.getCall(0), undefined );
    equal( errorCallback.getCall(0).args[1], 'Todo.done must be a boolean
value.' );
});

```

Коллекции

Для коллекции мы хотим протестировать следующие условия:

- коллекция содержит модель задачи;
- коллекция использует локальное хранилище для синхронизации;
- методы `done()`, `remaining()` и `clear()` работают должным образом;
- порядок задач численно корректен.

```

describe('Test Collection', function() {
    beforeEach(function() {
        // определение новых задач

```

```
this.todoOne = new Todo;
this.todoTwo = new Todo({
  title: "Buy some milk"
});
// создание новой коллекции задач для тестирования
return this.todos = new TodoList([this.todoOne, this.todoTwo]);
});
it('Has the Todo model', function() {
  return expect(this.todos.model).toBe(Todo);
});
it('Uses localStorage', function() {
  return expect(this.todos.localStorage).toEqual(new Store
  ('todos-backbone'));
});
describe('done', function() {
  return it('returns an array of the todos that are done', function() {
    this.todoTwo.done = true;
    return expect(this.todos.done()).toEqual([this.todoTwo]);
  });
});
describe('remaining', function() {
  return it('returns an array of the todos that are not done', function() {
    this.todoTwo.done = true;
    return expect(this.todos.remaining()).toEqual([this.todoOne]);
  });
});
describe('clear', function() {
  return it('destroys the current todo from localStorage', function() {
    expect(this.todos.models).toEqual([this.todoOne, this.todoTwo]);
    this.todos.clear(this.todoOne);
    return expect(this.todos.models).toEqual([this.todoTwo]);
  });
});
return describe('Order sets the order on todos ascending numerically',
  function() {
    it('defaults to one when there arent any items in the collection',
      function() {
        this.emptyTodos = new TodoApp.Collections.TodoList;
        return expect(this.emptyTodos.order()).toEqual(0);
      });
    return it('Increments the order by one each time', function() {
      expect(this.todos.order(this.todoOne)).toEqual(1);
      return expect(this.todos.order(this.todoTwo)).toEqual(2);
    });
  });
});
```

Представления

Для представлений мы хотим проверить следующие условия:

- при создании представления корректно привязываются к DOM-элементу;
- представление способно отображаться, после чего его DOM-элемент становится видимым;
- представления поддерживают привязку своих методов к DOM-элементам.

Расширим задачи тестирования и проверим, что взаимодействие пользователя с представлениями через пользовательский интерфейс приводит к корректному изменению соответствующих моделей:

```
module( 'About Backbone.View', {
    setup: function() {
        $('body').append('<ul id="todoList"></ul>');
        this.todoView = new TodoView({ model: new Todo() });
    },
    teardown: function() {
        this.todoView.remove();
        $('#todoList').remove();
    }
});
test('Should be tied to a DOM element when created, based off the property provided.', function() {
    expect( 1 );
    equal( this.todoView.el.tagName.toLowerCase(), 'li' );
});
test('Is backed by a model instance, which provides the data.', function() {
    expect( 2 );
    notEqual( this.todoView.model, undefined );
    equal( this.todoView.model.get('done'), false );
});
test('Can render, after which the DOM representation of the view will be visible.', function() {
    this.todoView.render();
    // Добавление DOM-элемента представления в ul#todoList
    $('#todoList').append(this.todoView.el);
    // проверка количества li-элементов, отображенных в списке
    equal($('#todoList').find('li').length, 1);
});
asyncTest('Can wire up view methods to DOM elements.', function() {
    expect( 2 );
    var viewElt;
    $('#todoList').append( this.todoView.render().el );
    setTimeout(function() {
        viewElt = $('#todoList li input.check').filter(':first');
        equal(viewElt.length > 0, true);
        // указание QUnit продолжать тестирование
        start();
    }, 1000, 'Expected DOM Elt to exist');
    // указание представлению изменить статус завершения
    // у одной или нескольких задач
    $('#todoList li input.check').click();
    // проверка истинности статуса завершения модели
    equal( this.todoView.model.get('done'), true );
});
```

Приложение

В некоторых случаях полезно написать тесты для всех существующих загрузчиков приложения. Для следующего модуля начальные настройки создаются и добавляются к представлению TodoApp, после чего мы можем проверить всё,

что нас интересует, — от корректности определений локальных экземпляров представлений до правильности изменений, вносимых в экземпляры локальных коллекций в результате взаимодействия приложения с пользователем.

```
module( 'About Backbone Applications' , {
  setup: function() {
    Backbone.localStorageDB = new Store('testTodos');
    $('#qunit-fixture').append('<div id="app"></div>');
    this.App = new TodoApp({ appendTo: $('#app') });
  },
  teardown: function() {
    this.App.todos.reset();
    $('#app').remove();
  }
});
test('Should bootstrap the application by initializing the Collection.', function() {
  expect( 2 );
  // коллекция todos не должна быть неопределенной
  notEqual( this.App.todos, undefined );
  // тем не менее начальная длина списка задач должна быть нулевой
  equal( this.App.todos.length, 0 );
});
test( 'Should bind Collection events to View creation.' , function() {
  // задать описание новой задачи в поле ввода
  $('#new-todo').val( 'Buy some milk' );
  // генерация события нажатия клавиши enter (return) в #new-todo,
  // приводящая к добавлению нового элемента в коллекцию todos
  $('#new-todo').trigger(new $.Event( 'keypress', { keyCode: 13 } ));
  // теперь длина нашей коллекции должна быть равной 1
  equal( this.App.todos.length, 1 );
});
```

Дополнительные источники информации

В этой главе мы рассмотрели тестирование приложений с помощью фреймворков QUnit и SinonJS. Я рекомендую проработать «коаны» для QUnit и Backbone.js и попробовать расширить приведенные в них примеры. В качестве дополнительных источников информации можно воспользоваться следующими материалами:

- Test-Driven JavaScript Development (<http://tddjs.com/>);
- SinonJS/QUnit adapter (<http://sinonjs.org/qunit/>);
- Using SinonJS with QUnit (<http://bit.ly/16tTGWl>);
- Automating JavaScript Testing with QUnit (<http://bit.ly/11YzmrQ>);
- Unit Testing with QUnit (<http://bit.ly/18v2dFc>);
- Another QUnit/Backbone.js demo project (<http://bit.ly/12s5cuK>);
- SinonJS helpers for Backbone (<http://bit.ly/17zHDF3>).

16 Заключение

Я рад, если это введение в библиотеку Backbone.js оказалось для вас полезным. Надеюсь, прочитав книгу, вы поняли, что хотя и можно совершить нечто вроде подвига, разработав сложное JavaScript-приложение с использованием только библиотеки манипулирования DOM (такой, как jQuery), все же трудно создать нетривиальную программу без какой-либо формальной структуры. Нагромождение вложенных обратных вызовов jQuery и DOM-элементов вряд ли удастся легко масштабировать, и поддержка развивающегося приложения окажется очень сложной.

Красота библиотеки Backbone заключается в ее простоте. Библиотека очень компактна, несмотря на ее функциональность и гибкость; это становится очевидным, когда вы начинаете изучать ее исходный код. Джереми Ашкенас (Jeremy Ashkenas) сказал: «Важнейшей идеей, лежащей в основе Backbone, всегда было стремление создать минимальный набор примитивов структурирования данных (модели и коллекции) и пользовательского интерфейса (представления и URL), с помощью которых удобно создавать веб-приложения на языке JavaScript». Все, что делает Backbone, — это помогает улучшить структуру ваших приложений за счет эффективного разделения функций.

Библиотека Backbone поддерживает модели, к которым можно привязывать пары «ключ-значение» и события, коллекции с API из расширенных перечислимых методов, описательные представления с обработкой событий, а также простое соединение существующего API с клиентским приложением через RESTful-интерфейс с использованием формата JSON. Backbone позволяет абстрагировать данные в адекватные модели, а DOM-манипуляции — в представления, и связывать их воедино, пользуясь исключительно механизмом событий.

Почти любой программист, разрабатывающий JavaScript-приложения в течение некоторого времени, в конечном счете создаст похожее решение, если для него

важна архитектура и эксплуатационная технологичность приложений. Вместо того чтобы пользоваться Backbone или другим аналогичным фреймворком, можно написать свой собственный, однако для этого зачастую приходится интегрировать различные библиотеки, не приспособленные к работе друг с другом. Вы можете управлять историей просмотров с помощью BBQ jQuery, работать с шаблонами средствами Handlebars, а создавать абстрактные объекты и тестировать код своими силами.

Сравните такое решение с библиотекой Backbone, которая включает *грамотную* документацию исходного кода, развитое сообщество пользователей и экспертов, а также сайтов вроде Stack Overflow, на которых ежедневно публикуются ответы на тысячи вопросов. Вместо того чтобы заново изобретать колесо, вы можете воспользоваться многочисленными преимуществами решения, основанного на коллективных знаниях и опыте сообщества, для структурирования своих приложений.

Библиотека Backbone не только помогает придать вашим приложениям адекватную структуру, но и обладает богатыми возможностями расширения и позволяет создавать приложения с нестандартной архитектурой, если им требуются возможности, отсутствующие в штатной комплектации Backbone. Об этом наглядно свидетельствуют многочисленные плагины и расширения, выпущенные для Backbone в течение последнего года, в том числе те, которые были рассмотрены нами (такие, как MarionetteJS и Thorax).

На сегодняшний день Backbone.js служит основой многих сложных веб-приложений — от мобильного клиента LinkedIn до популярных RSS-ридеров вроде NewsBlur и виджетов для комментирования наподобие Disqus. Эта небольшая библиотека с простыми, но рациональными абстракциями помогла разработать новое поколение многофункциональных веб-приложений, и я вместе со своими коллегами надеюсь, что когда-нибудь она поможет и вам.

Если вы размышляете над тем, стоит ли использовать Backbone в своем проекте, то задайте себе вопросы: «Обладает ли ваше приложение достаточной сложностью? Находится ли на пределе ваша способность структурировать его код? Будет ли приложение периодически изменять содержимое пользовательского интерфейса, не загружая новые страницы с сервера? Выигрываете ли вы от разделения функций?» Если да, то решение вроде Backbone может помочь вам.

В качестве примера хорошо структурированного одностраничного приложения часто приводят Gmail компании Google. Если вы пользовались им, то знаете, что оно запрашивает исходные данные, содержащие JavaScript-, CSS- и HTML-компоненты, требуемые большинству пользователей, а весь остальной функционал используется позже и работает в фоновом режиме. Gmail легко переключается между папками входящих писем и спама, при этом не отображая

заново всю страницу. Библиотеки вроде Backbone упрощают разработку таких интерфейсов.

Несмотря на это, Backbone не принесет пользы, если вы собираетесь разрабатывать приложение, которое не оправдывает усилий, потраченных на ее освоение. Если сервер возьмет на себя трудоемкую задачу формирования и передачи готовых страниц вашему приложению или сайту, то, скорее всего, для простых эффектов и взаимодействий лучше подойдет JavaScript или jQuery. Потратьте время на то, чтобы разобраться, как Backbone способна помочь вам, и принимайте решение исходя из особенностей конкретного проекта.

Изучение и применение Backbone не отличается сложностью, и время, которое вы потратите на изучение структурирования ваших приложений, окупится с лихвой. Хотя чтение этой книги даст вам базовые знания о библиотеке, лучший способ обучения — создавать собственные реальные приложения. Надеюсь, что в результате код вашего конечного продукта станет более четким, организованным и легким в поддержке.

Я желаю вам всего наилучшего на пути в мир Backbone и хочу закончить свое повествование фразой американского писателя Генри Миллера: «Цель любого путешествия — не место, а новый взгляд на вещи».

A

Темы для дальнейшего изучения

Простой пример реализации MVC-фреймворка для JavaScript

Полное рассмотрение реализации библиотеки Backbone выходит за рамки этой книги. Тем не менее рассмотрим простую MVC-библиотеку (назовем ее Cranium.js), иллюстрирующую реализацию паттерна MVC в фреймворках, аналогичных Backbone.

Как и при работе с Backbone, мы воспользуемся функциями наследования и шаблонизации библиотеки Underscore.

Система событий

В основе реализации MVC-фреймворка для JavaScript-приложений лежит система (объект) Event на базе паттерна публикации/подписки, которая позволяет MVC-компонентам взаимодействовать, будучи изолированными друг от друга. Подписчики прослушивают интересующие их определенные события и реагируют на них, когда публикаторы генерируют эти события.

Класс Event примешан к компонентам представления и модели, чтобы их экземпляры могли публиковать нужные события.

```
// cranium.js - Cranium.Events
var Cranium = Cranium || {};
// Задание утилиты выбора DOM
var $ = document.querySelector.bind(document) || this.jQuery || this.Zepto;
// Примешивание к любому объекту, чтобы позволить ему использовать события.
var Events = Cranium.Events = {
    // список событий и связанных с ними слушателей
    channels: {},
    // счетчик
```

продолжение ↗

```

eventNumber: 0,
// оповещение о событиях и передача данных слушателям;
trigger: function (events, data) {
    for (var topic in Cranium.Events.channels){
        if (Cranium.Events.channels.hasOwnProperty(topic)) {
            if (topic.split("-")[0] == events){
                Cranium.Events.channels[topic](data) !== false ||
                delete Cranium.Events.channels[topic];
            }
        }
    },
    // регистрация типа события и его слушателя
    on: function (events, callback) {
        Cranium.Events.channels[events + --Cranium.Events.eventNumber] = callback;
    },
    // удаление регистрации типа события и его слушателя
    off: function(topic) {
        delete Cranium.Events.channels[topic];
    }
};

```

Система Event позволяет:

- **представлению** — уведомлять своих подписчиков о взаимодействии с пользователем (например, о щелчках мыши или вводе данных в форму), обновлять свой внешний вид, заново отображаться и т. п.;
- **модели, данные которой изменились**, — уведомлять своих подписчиков о том, что им необходимо обновиться (например, представление, которое должно отобразить новые данные) и т. п.

Модели

Модели управляют специфичными для приложения данными. Они не связаны ни с пользовательским интерфейсом, ни с уровнем представления и содержат структурированные данные, которые могут потребоваться приложению. Когда модель изменяется (например, в результате обновления), она уведомляет об этом своих наблюдателей (подписчиков), чтобы они могли отреагировать на изменение.

Рассмотрим простую реализацию модели:

```

// cranium.js - Cranium.Model
// Attributes представляет данные, свойства модели.
// Они передаются в экземпляр модели при его создании.
// Мы также создаем id для каждого экземпляра модели,
// чтобы он мог идентифицировать себя (например, при
// уведомлении об изменениях)
var Model = Cranium.Model = function (attributes) {

```

```
this.id = _.uniqueId('model');
this.attributes = attributes || {};
};

//читывающий метод;
//возвращает именованный элемент данных
Cranium.Model.prototype.get = function(attrName) {
    return this.attributes[attrName];
};

//устанавливающий метод;
//установка/примешивание к данным, привязанным к модели (e.g.{name: "John"})
//и публикация события change
Cranium.Model.prototype.set = function(attrs){
    if (_.isObject(attrs)) {
        _.extend(this.attributes, attrs);
        this.change(this.attributes);
    }
    return this;
};

//возвращает клон объекта данных Models
//(used for view template rendering)
Cranium.Model.prototype.toJSON = function(options) {
    return _.clone(this.attributes);
};

//вспомогательная функция, уведомляющая об изменении Model
//и передающая новые данные
Cranium.Model.prototype.change = function(attrs){
    this.trigger(this.id + 'update', attrs);
};

//Примешивание системы Event
_.extend(Cranium.Model.prototype, Cranium.Events);
```

Представления

Представления являются визуальным образом моделей, содержащим фильтрованное отображение их текущего состояния. Представление, как правило, наблюдает за моделью и получает уведомления при ее изменении, что позволяет ему соответствующим образом обновляться. В литературе о паттернах проектирования представления обычно называют *неинтеллектуальными*, имея в виду, что они обладают ограниченными сведениями о моделях и контроллерах.

Рассмотрим представления на следующем JavaScript-примере:

```
// DOM-представление
var View = Cranium.View = function (options) {
    // примешивание объекта options (например, для расширения функциональности)
    _.extend(this, options);
    this.id = _.uniqueId('view');
};

//Примешивание системы Event
_.extend(Cranium.View.prototype, Cranium.Events);
```

Контроллеры

Контроллеры являются посредниками между моделями и представлениями и решают следующие задачи:

- обновление представления при изменении модели;
- обновление модели при манипуляциях пользователя с представлением.

```
// cranium.js - Cranium.Controller
// контроллер, связывающий модель и представление
var Controller = Cranium.Controller = function(options){
    // Примешивание объекта options (например, для расширения функциональности)
    _.extend(this, options);
    this.id = _.uniqueId('controller');
    var parts, selector, eventType;
    // Разбор объекта Events, переданного при определении
    // контроллера, и его привязка к методу, который будет его обрабатывать;
    if(this.events){
        _.each(this.events, function(method, eventName){
            parts = eventName.split('.');
            selector = parts[0];
            eventType = parts[1];
            $(selector)['on' + eventType] = this[method];
            }.bind(this));
    }
};
```

Практическое использование

Ниже приведен HTML-шаблон и пример, для которого он предназначен:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
</head>
<body>
<div id="todo">
</div>
<script type="text/template" class="todo-template">
    <div>
        <input id="todo_complete" type="checkbox" <%= completed %>>
        <%= title %>
    </div>
</script>
<script src="underscore-min.js"></script>
<script src="cranium.js"></script>
```

```
<script src="example.js"></script>
</body>
</html>
```

Использование Cranium.js:

```
// example.js - использование Cranium MVC
// и экземпляра задачи
var todo1 = new Cranium.Model({
    title: "",
    completed: ""
});
console.log("First todo title - nothing set: " + todo1.get('title'));
todo1.set({title: "Do something"});
console.log("Its changed now: " + todo1.get('title'));
//

// экземпляр представления
var todoView = new Cranium.View({
    // селектор DOM-элемента
    el: '#todo',
    // шаблон задачи; используется Underscore-шаблон
    template: _.template($('.todo-template').innerHTML),
    init: function (model) {
        this.render( model.toJSON() );
        this.on(model.id + 'update', this.render.bind(this));
    },
    render: function (data) {
        console.log("View about to render.");
        $(this.el).innerHTML = this.template( data );
    }
});
var todoController = new Cranium.Controller({
    // задание обновляемой модели
    model: todo1,
    // и представления, наблюдающего за этой моделью
    view: todoView,
    events: {
        "#todo.click" : "toggleComplete"
    },
    // инициализация всего
    initialize: function () {
        this.view.init(this.model);
        return this;
    },
    // изменение значения задачи в модели
    toggleComplete: function () {
        var completed = todoController.model.get('completed');
        console.log("Todo old 'completed' value?", completed);
        todoController.model.set({ completed: (!completed) ? 'checked': '' });
        console.log("Todo new 'completed' value?",
        todoController.model.get('completed'));
        return this;
    }
});
```

продолжение ↗

```
// запуск приложения
todoController.initialize();
todo1.set({ title: "Due to this change Model will notify View and
it will rerender"});
```

Самуэль Клэй (Samuel Clay), один из авторов первой версии библиотеки Backbone.js, отзывается о Cranium.js так: «Неудивительно, что она выглядит очень похоже на начальный вариант Backbone. Представления просты и потому требуют мало подготовительного кода и настройки. Модели отвечают за свои атрибуты и уведомления об их изменениях».

Я надеюсь, что вы нашли эту реализацию полезной в плане объяснения процесса написания «с нуля» собственной библиотеки, аналогичной Backbone, и приобретенный опыт побудит вас и в дальнейшем пользоваться существующими решениями, не пренебрегая при этом изучением их внутренних механизмов.

MVP

MVP (Model-View-Presenter, модель-представление-презентатор) — это производная паттерна проектирования MVC, предназначенная для улучшения презентационной логики. Она создана компанией Taligent в начале 1990-х годов в процессе работы над моделью для C++-среды CommonPoint. Хотя оба паттерна, MVC и MVP, нацелены на разделение функций между множеством компонентов, между ними существует ряд фундаментальных различий.

Далее мы обратим внимание на версию MVP, которая лучше всего подходит для веб-архитектур.

Модели, представления и презентаторы

Буква «P» в аббревиатуре MVP означает «презентатор» (*presenter*). Это компонент, содержащий бизнес-логику пользовательского интерфейса представления. В отличие от паттерна MVC, вызовы из представления делегируются презентатору, при этом они отделены от представления и общаются с ним через определенный интерфейс. Это создает ряд полезных механизмов, в том числе возможность имитировать представления в модульных тестах.

Наиболее распространены реализации MVP, которые используют *пассивное представление*, содержащее минимум логики (вплоть до ее отсутствия). MVP-модели почти идентичны MVC-моделям и обрабатывают данные приложения. Презентатор работает как посредник, взаимодействующий и с представлением, и с моделью; тем не менее представление и модель изолированы друг от друга. Презентаторы эффективно связывают модели с представлениями, выполняя

ту же функцию, что и контроллеры в MVC. Презентаторы составляют основу паттерна MVP и, как можно догадаться, содержат в себе презентационную логику, на которой базируются представления.

Презентаторы получают обращения от представлений, обрабатывают запросы пользователей и возвращают им данные. Они принимают данные, обрабатывают их и определяют, каким образом данные должны быть отображены в представлении. В некоторых реализациях презентатор также взаимодействует с уровнем служб для сохранения данных (моделей). Модели могут генерировать события, однако ответственность за подписку на них и обновление представления лежит на презентаторе. В этой пассивной архитектуре отсутствует концепция прямой привязки данных. Представления публикуют методы установки, с помощью которых презентаторы могут записывать данные.

Преимущество такого отличия от MVC заключается в повышении тестопригодности приложения и более четком разделении представления и модели. Платой за это является отсутствие поддержки привязки данных, которую разработчик вынужден реализовывать самостоятельно.

Хотя пассивное представление всего лишь реализует интерфейс, оно может быть устроено по-разному, в том числе использовать события, дополнительно отделяющие представление от презентатора. Поскольку в Java-Script нет компонента «интерфейс», мы используем его здесь скорее как протокол, нежели как собственно интерфейс. Технически он представляет собой API, поэтому с этой точки зрения мы вполне можем называть его интерфейсом.

Также существует вариант MVP с использованием наблюдаемого контроллера, который ближе к MVC и паттернам MVVM (Model–View–ViewModel, модель–представление–модель представления), поскольку поддерживает привязку к данным модели непосредственно из представления. Плагины наблюдения за ключами и значениями (key/value observing, KVO) (например, Backbone. ModelBinding, написанный Дериком Бейли (Derick Bailey)) реализуют концепцию наблюдаемого контроллера в Backbone.

MVP или MVC?

MVP чаще всего используется в приложениях корпоративного уровня, где необходимо многократно использовать презентационную логику. MVC не очень хорошо подходит для приложений со сложными представлениями и системой взаимодействия с пользователями, поскольку для реализации их функционала может потребоваться множество контроллеров. В MVP всю эту сложную логику можно поместить в презентатор, это существенно упростит поддержку приложения.

Поскольку MVP-представления определяются через интерфейс, а интерфейс технически является единственной точкой соприкосновения системы и представления (за исключением презентатора), этот паттерн также позволяет разработчикам создавать презентационную логику без необходимости ждать, пока дизайнеры разработают макеты и графику приложения.

В некоторых реализациях MVP может оказаться проще MVC с точки зрения автоматизации модульного тестирования. Это часто объясняют тем, что презентатор можно использовать как мок-объект, полностью имитирующий пользовательский интерфейс, и проводить его тестирование независимо от других компонентов приложения. По моему опыту, это в действительности зависит от языков, на которых реализуется MVP (существует большая разница между проектами, в которых MVP реализован на JavaScript и, скажем, на ASP.NET).

В конце концов, внутренние проблемы, с которыми вы можете столкнуться в MVC, скорее всего, будут иметь место и в MVP, поскольку различия между ними в основном носят семантический характер. Четко разделяя задачи между моделями, представлениями и контроллерами (или презентаторами), вы получите максимальные преимущества, независимо от используемого паттерна.

MVC, MVP и Backbone.js

Очень мало JavaScript-фреймворков (а возможно, и ни одного) претендуют на реализацию классической формы паттерна MVC или MVP, поскольку многие JavaScript-разработчики не считают, что MVC и MVP исключают друг друга (на самом деле, строгую реализацию MVP можно увидеть в веб-фреймворках вроде ASP.NET или GWT). Это объясняется тем, что в приложение можно добавлять дополнительную логику презентатора/представления и при этом рассматривать его как разновидность MVC.

Одна из разработчиков библиотеки Backbone Ирэн Рос (Irene Ros) рассуждает подобным образом, поскольку для разделения представления Backbone на отдельные компоненты ей нужен механизм, который связывает их воедино. Это может быть контроллер-маршрутизатор (такой, как `Backbone.Router`) или обратный вызов в ответ на получение данных.

Но некоторые разработчики все же считают, что Backbone.js больше подходит под каноны MVP, нежели MVC. Они аргументируют свою точку зрения следующим образом:

- презентатор в MVP описывает класс `Backbone.View` (слой между шаблонами представлений и связанными с ними данными) лучше, чем контроллер;
- модель соответствует классу `Backbone.Model` (не столь существенно отличается от классической MVC-модели);

- представлениям лучше всего соответствуют шаблоны (например, шаблоны разметки Handlebars/Mustache).

Ответить на эту аргументацию можно следующим образом. Backbone-представление является представлением с точки зрения MVC, поскольку библиотека Backbone обладает достаточной гибкостью, чтобы ее представление можно было использовать в различных целях. Класс `Backbone.View` может играть роль как представления MVC, так и презентатора MVP, поскольку он позволяет решать две задачи: отображать отдельные компоненты и собирать воедино компоненты, отображаемые другими представлениями.

Как вы уже видели, в Backbone функции контроллера выполнялись как классом `Backbone.View`, так и классом `Backbone.Router`, и в следующем примере мы сможем убедиться в истинности данного подхода.

Здесь представление `TodoView` использует паттерн «наблюдатель», чтобы подписаться на изменения своей модели в строке `this.model.on('change',...)`. Оно также обрабатывает шаблон в методе `render()`, но, в отличие от некоторых других реализаций, обработка взаимодействия с пользователем также выполняется в представлении (см. `events`).

```
// DOM-элемент задачи...
app.TodoView = Backbone.View.extend({
    //... представляет собой тег списка.
    tagName: 'li',
    // передача содержимого шаблона задачи через функцию
    // шаблонизации, кэширование шаблона для отдельной задачи
    template: _.template( $('#item-template').html() ),
    // DOM-события, специфичные для элемента.
    events: {
        'click .toggle': 'togglecompleted'
    },
    // представление TodoView прослушивает изменения своей модели
    // и выполняет повторное отображение. Поскольку в этом приложении
    // **Todo** и **TodoView** соотносятся как 1 к 1,
    // для удобства мы устанавливаем прямую ссылку на модель.
    initialize: function() {
        this.model.on( 'change', this.render, this );
        this.model.on( 'destroy', this.remove, this );
    },
    // повторное отображение названия задачи.
    render: function() {
        this.$el.html( this.template( this.model.toJSON() ) );
        return this;
    },
    // переключение состояния завершения модели.
    togglecompleted: function() {
        this.model.toggle();
    },
});
```

Еще одно (совершенно отличное от других) мнение заключается в том, что библиотека Backbone больше всего напоминает реализацию MVC в языке Smalltalk-80, который мы рассмотрели ранее.

Автор библиотеки MarionetteJS Дерик Бейли написал, что лучше всего не подгонять Backbone под рамки конкретных паттернов проектирования. Паттерны проектирования следует рассматривать как гибкие руководства по способам структурирования приложений, и в этом отношении библиотека Backbone не соответствует ни MVC, ни MVP. Она заимствует наиболее удачные концепции из многих архитектурных паттернов и формирует гибкий и эффективный фреймворк. Используйте любые термины, помогающие описать архитектуру приложения: «*в стиле Backbone*», MV* и т. п.

Тем не менее *следует* понимать, где и почему возникли эти концепции, поэтому я надеюсь, что мои пояснения относительно MVC и MVP были полезны. Более структурированные JavaScript-фреймворки случайно или намеренно по-своему реализуют классические паттерны, однако важно то, что с их помощью мы можем разрабатывать хорошо структурированные и легкие в поддержке приложения.

Пространства имен

Изучая Backbone, вы поймете, что *пространства имен* являются важной темой, которой уделяется недостаточно внимания в учебной литературе. Если вы уже имеете опыт работы с пространствами имен в JavaScript, то в следующем разделе найдете рекомендации по специфике применения известных вам концепций в Backbone. Я приведу необходимые пояснения и для новичков, чтобы материал был одинаково понятен всем читателям.

Что такое пространства имен?

Пространства имен представляют собой способ предотвращения конфликтов с другими объектами или переменными в глобальном пространстве имен. Использование пространства имен снижает вероятность сбоев в коде из-за того, что другой сценарий использует переменные с теми же именами, что и вы. Как добросовестный участник глобального пространства имен вы должны сделать все от вас зависящее, чтобы минимизировать вероятность сбоев сценариев других разработчиков из-за вашего кода.

В отличие от других языков, JavaScript на самом деле не имеет встроенной поддержки пространств имен, однако поддерживает замыкания, с помощью которых можно добиться аналогичного эффекта.

В этом разделе мы рассмотрим примеры создания пространств имен для моделей, представлений, маршрутизаторов и других компонентов. Мы изучим следующие паттерны:

- одиночные глобальные переменные;
- объектные константы;
- вложенные пространства имен.

Одиночные глобальные переменные

Популярный паттерн пространств имен в JavaScript использует в качестве основного объекта одиночную глобальную переменную. Ниже приведена эскизная реализация этого паттерна, в которой мы возвращаем объект с помощью функций и свойств:

```
var myApplication = (function(){  
    function(){  
        // ...  
    },  
    return {  
        // ...  
    }  
}());
```

Вероятно, вы уже сталкивались с этой техникой ранее. Пример, специфичный для Backbone, выглядит следующим образом:

```
var myViews = (function(){  
    return {  
        TodoView: Backbone.View.extend({ .. }),  
        TodosView: Backbone.View.extend({ .. }),  
        AboutView: Backbone.View.extend({ .. });  
        //etc.  
    };  
}());
```

Здесь мы можем вернуть набор представлений, однако этим же способом можно вернуть и целую коллекцию моделей, представлений и маршрутизаторов в зависимости от того, как вы решите структурировать свое приложение. Несмотря на то что этот паттерн работает в определенных ситуациях, самая большая проблема, связанная с одиночной глобальной переменной, — гарантировать, чтобы никто не воспользовался глобальной переменной с таким же именем на странице раньше вас.

Согласно Питеру Мишо (Peter Michaux), одно из решений этой проблемы заключается в использовании *префиксных пространств имен*. Идея этой

концепции проста: вам необходимо выбрать общий префикс (в данном примере `myApplication_`), а вслед за ним определить методы, переменные и другие объекты.

```
var myApplication_todoView = Backbone.View.extend({}),
myApplication.todosView = Backbone.View.extend({});
```

Это снижает вероятность существования переменной с таким же именем в глобальной области видимости, однако помните, что с тем же успехом можно использовать и объекты с уникальными именами, но это делает код менее читабельным. Кроме того, самая большая проблема этого паттерна в том, что по мере роста приложения в нем становится много глобальных объектов.

Более подробные соображения Питера Мишо об использовании одиночных глобальных переменных можно прочесть в его замечательной публикации.



Существует несколько вариантов паттерна одиночной глобальной переменной, однако, изучив несколько из них, я посчитал использование префиксного подхода оптимальным для Backbone.

Объектные константы

Достоинство объектных констант в том, что они не засоряют глобальное пространство имен, но при этом помогают логично организовать код и параметры. Их использование помогает создавать легкочитаемые структуры с глубоко вложенными подэлементами. В отличие от простых глобальных переменных, объектные константы проверяют существование переменной с идентичным именем, что помогает снизить вероятность конфликта.

Этот пример демонстрирует два способа проведения проверки существования пространства имен прежде, чем вы определите его. Я предпочитаю вариант 2.

```
/* Не проверяет существование myApplication */
var myApplication = {};
/*
Выполняет проверку существования. Если объект существует, мы используем
его экземпляр.
Вариант 1: if(!myApplication) myApplication = {};
Вариант 2: var myApplication = myApplication || {};
Затем мы можем включить в объектную константу поддержку моделей, представлений,
коллекций и любых других данных:
*/
var myApplication = {
  models : {},
  views : {}}
```

```
    pages : {}
  },
  collections : {}
};
```

Можно также добавлять свойства непосредственно в пространство имен (например, представления, как показано ниже):

```
var myTodosViews = myTodosViews || {};
myTodosViews.todoView = Backbone.View.extend({});  
myTodosViews.todosView = Backbone.View.extend({});
```

Преимущество этого паттерна в том, что вы можете легко инкапсулировать все ваши модели, представления, маршрутизаторы и другие объекты так, что они будут четко отделены друг от друга, создав при этом фундамент для расширения кода вашего приложения.

У этого паттерна есть ряд достоинств. Удачным решением является выделение конфигураций вашего приложения по умолчанию в единственную область, которую можно легко изменять, не выполняя поиск настроек во всей кодовой базе. Ниже приведен пример возможной объектной константы, которая хранит конфигурационные настройки приложения:

```
var myConfig = {
  language: 'english',
  defaults: {
    enableDelegation: true,
    maxTodos: 40
  },
  theme: {
    skin: 'a',
    toolbars: {
      index: 'ui-navigation-toolbar',
      pages: 'ui-custom-toolbar'
    }
  }
}
```

Обратите внимание, что между паттерном объектной константы и набором данных в формате JSON имеются лишь небольшие синтаксические различия. Если вы по какой-либо причине захотите воспользоваться JSON для хранения настроек (например, для удобства их отправки на сервер), то поступите именно так.

Для получения дополнительной информации о паттерне объектной константы я рекомендую прочитать статью Ребекки Мерфи (Rebecca Murphey), посвященную этой теме.

Вложенные пространства имен

Вложенные пространства имен являются расширением паттерна объектных констант. Это еще один распространенный паттерн, снижающий риск возникновения конфликтов за счет того, что даже если пространство имен верхнего уровня уже существует, то маловероятно, что существуют такие же вложенные дочерние пространства имен. Например, в библиотеке YUI компании Yahoo! широко используются вложенные пространства имен объектов:

```
YAHOO.util.Dom.getElementsByClassName('test');
```

Организация DocumentCloud, разработавшая Backbone, также пользуется вложенными пространствами имен в своих ключевых приложениях. Пример реализации вложенных пространств имен в Backbone выглядит следующим образом:

```
var todoApp = todoApp || {};
// выполните аналогичную проверку для вложенных дочерних пространств имен
todoApp.routers = todoApp.routers || {};
todoApp.model = todoApp.model || {};
todoApp.model.special = todoApp.model.special || {};
// маршрутизаторы
todoApp.routers.Workspace = Backbone.Router.extend({}); 
todoApp.routers.TodoSearch = Backbone.Router.extend({}); 
// модели
todoApp.model.Todo = Backbone.Model.extend({}); 
todoApp.model.Notes = Backbone.Model.extend({}); 
// особые модели
todoApp.model.special.Admin = Backbone.Model.extend({});
```

Это читабельный, понятно организованный и относительно безопасный способ использования пространств имен в Backbone-приложении. Обратите внимание, что данный способ требует, чтобы JavaScript-механизм вашего браузера сначала создал объект `todoApp`, а затем «спустился вниз» до функции, которую вы вызываете. Тем не менее некоторые разработчики, например Юрий Зайцев (Jurij Zaytsev, kangax), провели тестирование и выяснили, что различия в производительности при использовании одиночных объектов и вложенных пространств имен несущественны.

Что используется в DocumentCloud?

Ниже приведена оригинальная рабочая область, созданная компанией DocumentCloud (которая разработала библиотеку Backbone, помните?) и надлежащим образом использующая пространства имен. Данный подход оправдан, поскольку документы компании (а также аннотации и списки публикаций) встроены в сторонние новостные сайты.

```
// пространства имен верхнего уровня для javascript-кода.  
(function() {  
    window.dc = {};  
    dc.controllers = {};  
    dc.model = {};  
    dc.app = {};  
    dc.ui = {};  
})();
```

Как видите, DocumentCloud объявляет пространство имен верхнего уровня в окне с названием `dc` (краткая форма имени приложения), за которым следуют вложенные пространства имен для контроллеров, моделей, пользовательского интерфейса и других компонентов.

Рекомендация

Из паттернов пространств имен, которые мы рассмотрели выше, при разработке Backbone-приложений я предпочитаю пользоваться вложенными пространствами имен с объектными константами.

Одиночные глобальные переменные хорошо работают в относительно простых приложениях. Но в крупных кодовых базах, где нужны как пространства имен верхнего уровня, так и вложенные подпространства имен, необходимо лаконичное решение, обладающее читабельностью и масштабируемостью. Я полагаю, что этот паттерн достигает обеих целей и хорошо подходит для большинства Backbone-приложений.

Дополнительно о зависимостях Backbone

В следующих разделах рассматривается использование в Backbone библиотек jQuery/Zepto и Underscore.js.

Манипуляции DOM

Хотя большинство разработчиков не станут использовать такую возможность, Backbone поддерживает задание нестандартной DOM-библиотеки путем замены следующих параметров. Вот фрагмент исходного кода:

```
// For Backbone's purposes, jQuery, Zepto, Ender, or My Library (kidding) owns  
// the `$` variable.  
Backbone.$ = root.jQuery || root.Zepto || root.ender || root.$;
```

С помощью оператора `Backbone.$ = myLibrary`; можно задать любую библиотеку для манипулирования DOM вместо библиотеки jQuery, используемой по умолчанию.

Полезные возможности

Библиотека Underscore.js активно используется «за кулисами» Backbone и выполняет различные действия от расширения объектов до привязки событий. Поскольку библиотека Underscore обычно целиком включена в Backbone, мы получаем свободный доступ к множеству ее полезных возможностей по работе с коллекциями, в том числе к фильтрации (`_.filter()`), сортировке (`_.sortBy()`), привязке (`_.map()`) и т. п.

Фрагмент исходного кода:

```
// Методы Underscore, которые мы хотим реализовать для коллекции.
// 90 % полезных возможностей Backbone-коллекций
// реализовано здесь:
var methods = ['forEach', 'each', 'map', 'collect', 'reduce', 'foldl',
  'inject', 'reduceRight', 'foldr', 'find', 'detect', 'filter', 'select',
  'reject', 'every', 'all', 'some', 'any', 'include', 'contains', 'invoke',
  'max', 'min', 'toArray', 'size', 'first', 'head', 'take', 'initial', 'rest',
  'tail', 'drop', 'last', 'without', 'indexOf', 'shuffle', 'lastIndexOf',
  'isEmpty', 'chain'];
// Примешивание всех Underscore-методов в виде прокси к `Collection#models`.
_.each(methods, function(method) {
  Collection.prototype[method] = function() {
    var args = slice.call(arguments);
    args.unshift(this.models);
    return _[method].apply(_, args);
  };
});
```

Полный связанный список поддерживаемых методов опубликован в официальной документации.

RESTful-сохранение

Backbone позволяет синхронизировать модели и коллекции с сервером с помощью методов `fetch`, `save` и `destroy`. Эти методы передаются функции Backbone.`sync`, которая фактически «обертывает» функцию jQuery/Zepto `$.ajax`, вызывая `GET`, `POST` и `DELETE` для соответствующих методов сохранения Backbone-моделей.

Фрагмент исходного кода Backbone.sync:

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'patch': 'PATCH',
  'delete': 'DELETE',
  'read': 'GET'
};
```

```
Backbone.sync = function(method, model, options) {  
    var type = methodMap[method];  
    // ... далее следуют многочисленные конфигурации Backbone.js. Затем...  
    // формирование запроса, позволяющее пользователю переопределять  
    // любые AJAX-параметры.  
    var xhr = options.xhr = Backbone.ajax(_.extend(params, options));  
    model.trigger('request', model, xhr, options);  
    return xhr;  
}
```

Маршрутизация

Вызовы Backbone.History.start опираются на то, что jQuery/Zepto привязывает слушателей событий popState или hash change к объекту window.

Фрагмент исходного кода из Backbone.history.start:

```
// В зависимости от того, используем ли мы pushState или наборы атрибутов,  
// и от того, поддерживается ли 'onhashchange',  
// мы определяем, как проверить состояние URL.  
if (this._hasPushState) {  
    Backbone.$(window)  
        .on('popstate', this.checkUrl);  
} else if (this._wantsHashChange && ('onhashchange' in window) && !oldIE) {  
    Backbone.$(window)  
        .on('hashchange', this.checkUrl);  
} else if (this._wantsHashChange) {  
    this._checkUrlInterval = setInterval(this.checkUrl, this.interval);  
}  
...  
...
```

Аналогичным образом Backbone.History.stop использует библиотеку манипуляции DOM для отключения этих слушателей событий.

Сравнение Backbone с другими библиотеками и фреймворками

Backbone — лишь одно из множества различных решений для структурирования приложений, и мы никоим образом не пытаемся представить его как универсальное и единственное. Оно сослужило хорошую службу авторам этой книги при создании многих простых и сложных веб-приложений, и надеемся, что оно послужит и вам. Ответ на вопрос «Что лучше: Backbone или X?» зависит в первую очередь от приложения, которое вы собираетесь разрабатывать.

Библиотеки AngularJS и Ember.js являются примерами альтернатив Backbone, однако отличаются от него большей строгостью. Для некоторых проектов это

может оказаться полезным; для некоторых, вероятно, нет. Важно запомнить, что не существует библиотеки или фреймворка, который является универсальным решением для любого случая, поэтому следует изучать инструменты, которыми вы пользуетесь, и выбирать из них наиболее подходящий для конкретного проекта.

Выбирайте соответствующий инструмент под соответствующую задачу. Я рекомендую потратить время на исследования. Принимайте во внимание такие факторы, как продуктивность, простота использования, тестопригодность, мнения сообщества пользователей и документация фреймворка. Если вас интересуют конкретные сравнения фреймворков друг с другом, то я рекомендую ознакомиться со следующими публикациями:

- «Путешествие сквозь джунгли JavaScript MVC» (Journey Through the JavaScript MVC Jungle);
- «Семь фреймворков для создания развитых JavaScript-приложений» (Rich JavaScript Applications – The Seven Frameworks).

Авторы библиотек Backbone.js, AngularJS и Ember обсуждали некоторые сильные и слабые стороны своих решений на сайтах Quora, StackOverflow и др.:

- Джереми Ашкенас (Jeremy Ashkenas) — преимущества Backbone;
- Том Дэйл (Tom Dale) — сравнение Ember.js и AngularJS;
- Брайан Форд (Brian Ford) и Джереми Ашкенас — сравнение Backbone и Angular (дискуссия).

Не исключено, что изучение решения позволит вам разрабатывать нетривиальные функции, и, в конечном счете, с его помощью вы обеспечите многолетнюю поддержку приложения, поэтому обдумайте следующие вопросы:

Каковы реальные возможности библиотеки/фреймворка?

Потратьте время на изучение исходного кода фреймворка и официального списка возможностей и определите, насколько они соответствуют вашим требованиям. В некоторых проектах требуется изменить или расширить существующий исходный код, и если это актуально в вашем случае, исследуйте код фреймворка. Испытан ли фреймворк в реальной разработке? Были ли с его помощью созданы и развернуты крупные приложения, имеющиеся в публичном доступе? У Backbone есть солидное портфолио таких приложений (SoundCloud, LinkedIn, Walmart), однако не все фреймворки могут похвастаться этим. Фреймворк Ember используется в ряде крупных приложений, в том числе в новой версии ZenDesk. Библиотека AngularJS

была использована при создании YouTube-приложения для PS3 и в нескольких других проектах. Важно не только знать, что фреймворк применяется в разработке, но и иметь возможность изучать реальный код приложений и руководствоваться его результатами.

Является ли фреймворк зрелым?

Я не рекомендую разработчикам приступать к работе с первым попавшимся фреймворком. Выпуск новых продуктов часто сопровождается шумихой, однако не теряйте бдительность, выбирая инструмент, с помощью которого собираетесь разрабатывать приложение промышленного уровня. Не подвергайте проект риску свертывания, длительной реорганизации и другим деструктивным изменениям, которых можно избежать при использовании зрелого фреймворка. Солидные проекты обычно сопровождаются подробной документацией, входящей в состав продукта или публикуемой сообществом разработчиков.

Является фреймворк гибким или строгим?

Существует большое число фреймворков обоих видов, поэтому определите, какой из них вам нужен. Строгие фреймворки требуют, чтобы вы работали с ними по их предписаниям. Они ограничивают свободу разработчиков, но в меньшей степени требуют от них самостоятельно придумывать нужные механизмы. Достаточно ли вы экспериментировали с интересующим вас фреймворком?

Сначала напишите небольшое приложение без фреймворков, а затем попробуйте изменить его код, воспользовавшись изучаемым фреймворком, и определите, насколько с ним легко работать. Убеждаться в удобстве фреймворка путем создания реальных программ с его помощью столь же важно, как и исследовать и изучать его исходный код.

Имеется ли у фреймворка исчерпывающий набор документации?

Несомненно, демонстрационные приложения могут быть полезными для общей информации, но я рекомендую всегда обращаться к официальной документации фреймворка, чтобы узнавать о возможностях его API, решении типовых задач, создании компонентов и возможных проблемах. Любой стоящий фреймворк должен сопровождаться подробной документацией, помогающей разработчикам использовать его. В отсутствие документации вы будете вынуждены опираться на информацию из IRC-каналов и групп, а также на собственные эксперименты. Это, несомненно, поможет вам, но займет больше времени по сравнению с использованием готовой документации.

Каков общий размер фреймворка с учетом поддерживаемых им средств минимизации, архивации и модульного программирования?

Какие зависимости имеются у фреймворка? Как правило, в качестве размера фреймворка указывается только общий размер файлов его базовой библиотеки без учета зависимостей. Разница между кажущейся компактностью фреймворка и его реальным размером может оказаться существенной, если он опирается, к примеру, на jQuery и другие библиотеки.

Обращались ли вы к сообществу разработчиков и пользователей фреймворка?

Существует ли активное сообщество разработчиков и пользователей проекта, способных помочь в сложной ситуации? Достаточно ли популярен фреймворк для того, чтобы его можно было изучать с помощью примеров приложений, руководств и, возможно, даже демороликов?

Б

ИСТОЧНИКИ

Книги и курсы

- Prosthetics and Orthotics (<https://leanpub.com/building-backbone-plugins>).
- PeepCode: Backbone.js Basics (<https://peepcode.com/products/backbone-js>).
- Prosthetics and Orthotics (<https://leanpub.com/building-backbone-plugins>).
- CodeSchool: Anatomy of Backbone (<http://www.codeschool.com/courses/anatomy-of-backbonejs>).
- Recipes with Backbone (<http://recipeswithbackbone.com/>).
- Backbone Patterns (<http://ricostacruz.com/backbone-patterns/>).
- Backbone on Rails (<https://learn.thoughtbot.com/products/1-backbone-js-on-rails>).
- Derick Bailey's Resources for Learning Backbone (<http://lostechies.com/derickbailey>).
- Learn Backbone.js Completely (<http://javascriptissexy.com/learn-backbone-js-completely>).
- Backbone.js on Rails (<https://learn.thoughtbot.com/products/1-backbone-js-on-rails>).

Расширения и библиотеки

- MarionetteJS (<http://marionettejs.com/>).
- Backbone Layout Manager (<http://bit.ly/14MJw3n>).

- AuraJS (<http://bit.ly/13Wzr3k>).
- Thorax (<http://thoraxjs.org/>).
- Lumbar (<http://bit.ly/13sFBUA>).
- Backbone Boilerplate (<http://bit.ly/YCoQs5>).
- Backbone Forms (<https://github.com/powmedia/backbone-forms>).
- Backbone-Nested (<http://afeld.github.com/backbone-nested/>).
- Backbone.Validation (<http://github.com/thedersen/backbone.validation>).
- Backbone.Offline (<https://github.com/Ask11/backbone.offline>).
- Backbone-relational (<https://github.com/PaulUithol/Backbone-relational>).
- Backgrid (<https://github.com/wyuenho/backgrid>).
- Backbone.ModelBinder (<http://bit.ly/14MJp7S>).
- Backbone Relational—for model relationships (<http://bit.ly/15FQOVY>).
- Backbone CouchDB (<http://bit.ly/YLix04>).
- Backbone.Validation—HTML5-inspired validations (<http://bit.ly/12P0JCu>).