

Майкл С. Миковски
Джош К. Пауэлл

Разработка одностраничных веб-приложений

Предисловие
Грегори Д. Бенсона



ДМК
ИЗДАТЕЛЬСТВО



MANNING

Майкл С. Миковски, Джош К. Пауэлл

Разработка одностраничных веб-приложений

Michael S. Mikowski, Josh C. Powell

Single Page Web Applications

Foreword by Gregory D. Benson



MANNING
SHELTER ISLAND

Майкл С. Миковски, Джош К. Пауэлл

Разработка одностраничных веб-приложений

С предисловием Грегори Д. Бенсона



Москва, 2014

УДК 004.738.5:004.45JavaScript
ББК 32.973.202-018.2
M59

Майкл С. Миковски, Джош К. Пауэлл
M59 Разработка одностраничных веб-приложений / пер. с англ.
Слинкина А. А. – М.: ДМК Пресс, 2014. – 512 с.: ил.

ISBN 978-5-97060-072-6

Если ваш сайт представляет собой набор дергающихся страниц, связанных ссылками, то вы отстали от жизни. Следующей ступенью вашей карьеры должны стать одностраничные приложения (SPA). В таком приложении отрисовка пользовательского интерфейса и бизнес-логика перенесены в браузер, а взаимодействие с сервером сводится к синхронизации данных. Пользователь работает с таким сайтом, как с персональным приложением на рабочем столе, что гораздо удобнее и приятнее. Однако разрабатывать, сопровождать и тестировать SPA нелегко.

В этой книге показано как организуется командная разработка передовых SPA —проектирование, тестирование, сопровождение и развитие — с применением JavaScript на всех уровнях и без привязки к какому-то конкретному каркасу.

Попутно вы отточите навыки работы с HTML5, CSS3 и JavaScript и узнаете об использовании JavaScript не только в браузере, но также на сервере и в базе данных.

УДК 004.738.5:004.45JavaScript
ББК 32.973.202-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-61729-075-6 (анг.)
ISBN 978-5-97060-072-6 (рус.)

© 2014 by Manning Publications Co.
© Оформление, перевод,
ДМК Пресс, 2014

*Моим родителям, жене и детям.
Вы так многому меня научили, я всех вас люблю.*

– М. С. М.

*Моей жене Марианне. Спасибо тебе за то, что ты так покорно
терпела, пока я писал эту книгу. Я люблю тебя.*

– Дж. К. П.

Содержание

Предисловие	13
Вступление	15
Благодарности	16
Об этой книге	19
Об иллюстрации на обложке	25
Часть I. Введение в SPA	26
Глава 1. Наше первое одностраничное приложение	28
1.1. Определение, немного истории и несколько слов о предмете книги.....	29
1.1.1. Немного истории	29
1.1.2. Почему SPA на JavaScript появились так поздно?.....	30
1.1.3. Предмет книги	34
1.2. Создаем наше первое SPA.....	36
1.2.1. Постановка задачи.....	36
1.2.2. Общая структура файла	37
1.2.3. Настройка инструментов разработчика в Chrome.....	38
1.2.4. Разработка HTML и CSS	39
1.2.5. Добавление JavaScript-кода	40
1.2.6. Изучение приложения с помощью инструментов разработчика в Chrome	46
1.3. Чем хорошо написанное SPA удобно пользователям	49
1.4. Резюме.....	51
Глава 2. Новое знакомство с JavaScript	53
2.1. Область видимости переменной	55
2.2. Поднятие переменных	58
2.3. Еще о поднятии переменных и объекте контекста выполнения.....	60
2.3.1. Поднятие.....	60
2.3.2. Контекст выполнения и объект контекста выполнения.....	62
2.4. Цепочка областей видимости.....	66
2.5. Объекты в JavaScript и цепочка прототипов	69
2.5.1. Цепочка прототипов.....	73

2.6. Функции – более пристальный взгляд.....	78
2.6.1. Функции и анонимные функции.....	78
2.6.2. Самовыполняющиеся анонимные функции.....	79
2.6.3. Паттерн модуля – привнесение в JavaScript закрытых переменных.....	82
2.6.4. Замыкания	88
2.7. Резюме.....	92

Часть II. Клиентская часть одностраничного

приложения.....	94
------------------------	-----------

Глава 3. Разработка оболочки	96
---	-----------

3.1. Знакомимся с Shell	96
3.2. Организация файлов и пространств имен.....	98
3.2.1. Создание дерева файлов	98
3.2.2. HTML-файл приложения.....	100
3.2.3. Создание корневого пространства имен CSS.....	101
3.2.4. Создание корневого пространства имен JavaScript.....	103
3.3. Создание функциональных контейнеров	104
3.3.1. Стратегия.....	105
3.3.2. HTML-код модуля Shell.....	105
3.3.3. CSS-стили модуля Shell	106
3.4. Отрисовка функциональных контейнеров	109
3.4.1. Преобразование HTML в JavaScript-код	110
3.4.2. Добавление HTML-шаблона в JavaScript-код	111
3.4.3. Создание таблицы стилей для Shell.....	113
3.4.4. Настройка приложения для использования Shell	115
3.5. Управление функциональными контейнерами	116
3.5.1. Метод сворачивания и раскрытия окна чата	117
3.5.2. Добавление обработчика события щелчка мышью по окну чата	119
3.6. Управление состоянием приложения	124
3.6.1. Какого поведения ожидает пользователь браузера?.....	124
3.6.2. Стратегия работы с элементами управления историей.....	125
3.6.3. Изменение якоря при возникновении события истории.....	126
3.6.4. Использование якоря для управления состоянием приложения	128
3.7. Резюме.....	135

Глава 4. Добавление функциональных модулей	137
4.1. Стратегия функциональных модулей	138
4.1.1. Сравнение со сторонними модулями	139
4.1.2. Функциональные модули и паттерн «фрактальный MVC»	141
4.2. Подготовка файлов функционального модуля	144
4.2.1. Планируем структуру каталогов и файлов	145
4.2.2. Создание файлов	146
4.2.3. Что мы соорудили	152
4.3. Проектирование API модуля	153
4.3.1. Паттерн якорного интерфейса	153
4.3.2. API конфигурирования модуля Chat	155
4.3.3. API инициализации модуля Chat	156
4.3.4. Метод <code>setSliderPosition</code> из API модуля Chat	157
4.3.5. Каскадное конфигурирование и инициализация	158
4.4. Реализация API функционального модуля	160
4.4.1. Таблицы стилей	160
4.4.2. Модификация модуля Chat	166
4.4.3. Модификация модуля Shell	172
4.4.4. Проследивание выполнения	179
4.5. Добавление часто используемых методов	181
4.5.1. Метод <code>removeSlider</code>	181
4.5.2. Метод <code>handleResize</code>	183
4.6. Резюме	188
Глава 5. Построение модели	189
5.1. Что такое модель	189
5.1.1. Что мы собираемся сделать	190
5.1.2. Что делает модель	192
5.1.3. Чего модель не делает	193
5.2. Подготовка файлов модели, и не только	194
5.2.1. Планируем структуру каталогов и файлов	194
5.2.2. Создание файлов	196
5.2.3. Использование унифицированной библиотеки ввода	202
5.3. Проектирование объекта <code>people</code>	202
5.3.1. Проектирование объекта <code>person</code>	203
5.3.2. Проектирование API объекта <code>people</code>	205
5.3.3. Документирование API объекта <code>people</code>	209
5.4. Реализация объекта <code>people</code>	210
5.4.1. Создание подставного списка людей	212

5.4.2. Начало реализации объекта people.....	213
5.4.3. Завершение работы над объектом people.....	218
5.4.4. Тестирование API объекта people	225
5.5. Реализация аутентификации и завершения сеанса в Shell.....	228
5.5.1. Проектирование пользовательского интерфейса аутентификации.....	229
5.5.2. Модификация JavaScript-кода модуля Shell	229
5.5.4. Тестирование аутентификации и завершения сеанса в пользовательском интерфейсе.....	233
5.6. Резюме.....	234
Глава 6. Завершение модулей Model и Data	235
6.1. Проектирование объекта chat	235
6.1.1. Проектирование методов и событий	236
6.1.2. Документирование API объекта chat	239
6.2. Реализация объекта chat.....	240
6.2.1. Начинаем с метода join	240
6.2.2. Модификация модуля Fake для поддержки метода chat.join.....	243
6.2.3. Тестирование метода chat.join.....	246
6.2.4. Добавление средств работы с сообщениями в объект chat	247
6.2.5. Модификация модуля Fake для имитации работы с сообщениями.....	252
6.2.6. Тестирование работы с сообщениями в чате.....	254
6.3. Добавление поддержки аватаров в модель.....	256
6.3.1. Добавление поддержки аватаров в объект chat	256
6.3.2. Модификация модуля Fake для имитации аватаров	258
6.3.3. Тестирование поддержки аватаров.....	259
6.3.4. Разработка через тестирование	260
6.4. Завершение функционального модуля Chat	262
6.4.1. Модификация JavaScript-кода модуля Chat.....	263
6.4.2. Модификация таблиц стилей.....	271
6.4.3. Тестирование пользовательского интерфейса чата.....	276
6.5. Разработка функционального модуля Avatar.....	277
6.5.1. JavaScript-код модуля Avatar	278
6.5.2. Создание таблицы стилей для модуля Avatar	284
6.5.3. Модификация модуля Shell и головного HTML-документа.....	285
6.5.4. Тестирование функционального модуля Avatar	286

6.6. Привязка к данным и jQuery.....	287
6.7. Разработка модуля Data.....	288
6.8. Резюме.....	291

Часть III. Сервер SPA..... 292

Глава 7. Веб-сервер..... 294

7.1. Роль сервера.....	294
7.1.1. Аутентификация и авторизация.....	294
7.1.2. Валидация.....	295
7.1.3. Сохранение и синхронизация данных.....	296
7.2. Node.js.....	297
7.2.1. Почему именно Node.js?.....	297
7.2.2. Приложение «Hello World» для Node.js.....	298
7.2.3. Установка и использование Connect.....	302
7.2.4. Добавление промежуточного уровня Connect.....	304
7.2.5. Установка и использование Express.....	305
7.2.6. Добавление промежуточного уровня в Express-приложение.....	308
7.2.7. Окружения в Express.....	309
7.2.8. Обслуживание статических файлов с помощью Express.....	310
7.3. Более сложная маршрутизация.....	312
7.3.1. CRUD-маршруты для управления пользователями.....	312
7.3.2. Обобщенная маршрутизация для операций CRUD.....	319
7.3.3. Перенос маршрутизации в отдельный модуль Node.js.....	322
7.4. Добавление аутентификации и авторизации.....	327
7.4.1. Базовая аутентификация.....	327
7.5. Веб-сокеты и Socket.IO.....	329
7.5.1. Простой пример применения Socket.IO.....	329
7.5.2. Socket.IO и сервер обмена сообщениями.....	333
7.5.3. Обновление JavaScript-кода с помощью Socket.IO.....	334
7.6. Резюме.....	338

Глава 8. Серверная база данных..... 339

8.1. Роль базы данных.....	339
8.1.1. Выбор хранилища данных.....	340
8.1.2. Исключение преобразования данных.....	340
8.1.3. Помещайте логику туда, где она нужнее.....	341
8.2. Введение в MongoDB.....	342
8.2.1. Документоориентированное хранилище.....	343

8.2.2. Динамическая структура документа.....	343
8.2.3. Начало работы с MongoDB.....	345
8.3. Драйвер MongoDB.....	347
8.3.1. Подготовка файлов проекта.....	347
8.3.2. Установка и подключение MongoDB	348
8.3.3. Использование методов CRUD в MongoDB	350
8.3.4. Добавление операций CRUD в серверное приложение.....	353
8.4. Валидация данных, поступивших от клиента	357
8.4.1. Проверка типа объекта	357
8.4.2. Проверка объекта.....	360
8.5. Создание отдельного модуля CRUD.....	368
8.5.1. Подготовка структуры файлов	369
8.5.2. Перенос операций CRUD в отдельный модуль.....	372
8.6. Реализация модуля chat.....	378
8.6.1. Начало модуля Chat	379
8.6.2. Создание обработчика сообщения adduser.....	382
8.6.3. Создание обработчика сообщения updatechat	386
8.6.4. Создание обработчиков отключения.....	388
8.6.5. Создание обработчика сообщения updateavatar.....	390
8.7. Резюме.....	393

Глава 9. Подготовка SPA к промышленной эксплуатации.....

9.1. Поисковая оптимизация SPA.....	396
9.1.1. Как Google индексирует SPA.....	396
9.2. Облачные и сторонние службы.....	400
9.2.1. Анализ работы сайта.....	400
9.2.2. Протоколирование ошибок на стороне клиента.....	403
9.2.3. Сети доставки содержимого	406
9.3. Кэширование и отключение кэширования	406
9.3.1. Варианты кэширования	407
9.3.2. Веб-хранилище	408
9.3.3. HTTP-кэширование	410
9.3.4. Кэширование на сервере	414
9.3.5. Кэширование запросов к базе данных.....	420
9.4. Резюме.....	421

Приложение А. Стандарт кодирования на JavaScript.....

А.1. Зачем нам стандарт кодирования?	424
--	-----

A.2. Форматирование кода и комментарии	425
A.2.1. Форматирование кода с учетом удобства чтения.....	426
A.2.2. Комментарии как средство пояснения и документирования.....	434
A.3. Именованние переменных	437
A.3.1. Сокращение и повышение качества комментариев за счет соглашений об именовании	437
A.3.2. Рекомендации по именованию.....	439
A.3.3. Практическое применение рекомендаций	447
A.4. Объявление и присваивание переменным.....	448
A.5. Функции	450
A.6. Пространства имен.....	453
A.7. Имена и структура дерева файлов.....	454
A.8. Синтаксис.....	455
A.8.1. Метки	456
A.8.2. Предложения.....	456
A.8.3. Прочие замечания о синтаксисе	459
A.9. Валидация кода	460
A.9.1. Установка JSLint	460
A.9.2. Настройка JSLint	461
A.9.3. Использование JSLint	462
A.10. Шаблон модуля.....	463
A.11. Резюме.....	465
Приложение Б. Тестирование SPA	467
Б.1. Режимы тестирования	468
Б.2. Выбор каркаса тестирования.....	472
Б.3. Настройка nodeunit.....	473
Б.4. Создание комплекта тестов.....	474
Б.4.1. Инструктируем Node.js загрузить наши модули	475
Б.4.2. Подготовка одного теста в nodeunit	478
Б.4.3. Создание первого настоящего теста.....	479
Б.4.4. План событий и тестов.....	480
Б.4.5. Создание комплекта тестов	483
Б.5. Адаптация модулей SPA для тестирования.....	496
Б.6. Резюме	499
Предметный указатель	501

Предисловие

Свое первое одностраничное веб-приложение (single page web application – SPA) я написал в 2006 году, хотя тогда оно еще так не называлось. Для меня это стало заметным событием. До того я занимался низкоуровневым программированием ядра Linux, а также параллельными и распределенными вычислениями, а пользовательский интерфейс сводился к простой командной строке. Получив в 2006 году постоянную преподавательскую работу в Университете Сан-Франциско, я затеял амбициозный проект в области распределенных вычислений, получивший название River (<http://river.cs.usfca.edu>). Там был нужен интерактивный графический интерфейс для управления распределенными машинами и отладки системы.

Алекс Рассел (Alex Russell) как раз тогда придумал термин «комета» (comet)¹, и мы с воодушевлением решили использовать эту технологию для реализации интерфейса в веб-браузере. Возникли сложности при попытке организовать взаимодействие в реальном времени с помощью JavaScript. Худо-бедно все работало, но не так эффективно, как мы надеялись. Проблема была в том, что нам почти все приходилось разрабатывать самостоятельно, так как имеющихся сегодня библиотек и технических приемов тогда еще не было и в помине. Например, первая версия jQuery вышла как раз в тот год, но позже.

В июле 2011 года я был директором по разработкам в компании SnapLogic, Inc. (<http://snaplogic.com>), а Майк Миковски был принят на должность архитектора пользовательского интерфейса. Мы работали в команде, которая проектировала продукт для интеграции данных следующего поколения. Вместе с Майком мы провели бесчисленные часы за обсуждением принципиальных вопросов программной инженерии и проектирования языков. Мы многому научились друг у друга. Майк также показал мне черновики той книги, которую вы сейчас читаете, и именно тогда я узнал о придуманном им и Джошем методе построения SPA. Было понятно, что они разработали несколько поколений коммерческих SPA и воспользовались этим опытом, чтобы отточить технику и архитектурные принципы, сделав их полными, ясными и сравнительно простыми.

¹ Comet – любая модель работы веб-приложения, при которой постоянное HTTP-соединение позволяет веб-серверу отправлять данные браузеру без дополнительного запроса со стороны браузера. См. [http://ru.wikipedia.org/wiki/Comet_\(программирование\)](http://ru.wikipedia.org/wiki/Comet_(программирование)). – *Прим. перев.*

Со времен проекта River 2006 года ингредиенты одностраничных приложений, работающих в браузере, достигли высокой степени зрелости и теперь, вообще говоря, превосходят сторонние подключаемые модули типа Java или Flash. Существует немало прекрасных книг об этих ингредиентах по отдельности: HTML, CSS, JavaScript, jQuery, NodeJS и HTTP. Но, к сожалению, почти нет книг, описывающих, как объединить их в единое целое.

Эта книга является исключением из правила. В ней подробно излагаются тщательно проверенные рецепты построения неотразимых SPA с помощью JavaScript, применяемого на всех уровнях стека. Авторы делятся опытом, приобретенным и отточенным в ходе разработки многих поколений SPA. Можно сказать, что Майк и Джош совершили немало ошибок, которые вам повторять вовсе не обязательно. Прочитав эту книгу, вы сможете сосредоточиться на цели приложения, а не на его реализации.

Решения, описанные в этой книге, основаны на современных веб-стандартах, поэтому они останутся актуальны еще долго и должны работать в самых разных браузерах и устройствах. Как бы мне хотелось, чтобы современные технологии и эта книга существовали в далеком 2006 году, когда мы работали над проектом River. Уж мы бы нашли им применение!

Грегори Д. Бенсон,
профессор факультета информатики
Университета Сан-Франциско

Вступление

С Джошем мы встретились летом 2011 года, когда я искал работу, и он предложил мне место архитектора. Хотя в конечном итоге я решил принять другое предложение, мы подружились и с интересом обсуждали тему одностраничных веб-приложений и будущего Интернета. Как-то раз Джош по наивности предложил вместе написать книгу. Я сдуру согласился, и это решило нашу судьбу на последующие сотни недель. Мы думали, что книга получится тоненькой, не больше 300 страниц. Идея была в том, чтобы постоять за спиной опытного разработчика, создающего SPA промышленного качества с помощью одного лишь JavaScript. Мы собирались использовать только лучшие в своем классе инструменты и приемы, чтобы предложить пользовательский интерфейс мирового уровня. Все наши идеи должны были быть применимы к разработке любого SPA на JavaScript – так, как делается в этой книге, или с помощью иных имеющихся библиотек и каркасов.

После того как черновик книги был опубликован на сайте издательства Manning в рамках программы раннего ознакомления, в первый же месяц поступило более тысячи заказов на книгу. Мы прислушивались к мнениям читателей и беседовали с тысячами разработчиков и авторитетов на разных встречах, в университетах, на отраслевых конференциях – чтобы разобраться, в чем заключается очарование SPA. Из услышанного мы сделали вывод, что существует неутоленная жажда знаний по этому вопросу. Мы поняли, что разработчики мечтают узнать о более совершенных способах создания веб-приложений. Поэтому мы включили дополнительные темы. Например, мы добавили приложение Б, по объему сопоставимое с главой, в котором детально описываем, как настроить автоматизированное тестирование SPA, поскольку многим читателям показалось, что тестирование рассматривается в рукописи недостаточно подробно.

В итоге мы сохранили подход к изложению, основанный на идее подглядывания за опытным разработчиком, и добавили ряд тем, о которых просили читатели. В результате «маленькая книжка» разрослась и почти вдвое превысила первоначально планировавшийся объем. Надеемся, что вам она понравится.

Майкл С. Миковски

Благодарности

Авторы выражают благодарность следующим лицам.

- Джою Бруксу (Joey Brooks), специалисту по подбору персонала, который познакомил нас. Это ты во всем виноват, Джой.
- Джону Резигу (John Resig) и всем разработчикам jQuery за создание фантастической библиотеки – лаконичной, расширяемой и функционально богатой. Благодаря jQuery SPA оказываются быстрее и надежнее, а их разработка доставляет удовольствие.
- Яну Смиту (Ian Smith) за создание и сопровождение TaffyDB, мощного инструмента для манипуляции данными в браузере.
- Нильсу Джонсону (Niels Johnson), известному также как «Spikels», за предложение вычитать наш материал в обмен на раннее ознакомление. Думаю, что от этой сделки выиграли мы, потому что его рецензии оказались поразительно подробными и весьма полезными при окончательном редактировании.
- Майклу Стивенсу (Michael Stephens) из издательства Manning, который помог нам собрать первый черновой вариант и определиться со структурой книги.
- Берту Бейтсу (Bert Bates), который лучше большинства живущих на этой планете знает, как писать технические книги. Он научил нас всегда помнить о том, для какой аудитории мы пишем.
- Карен Миллер (Karen Miller), нашему редактору-консультанту, которая работала вместе с нами почти все время, понукала и нас, и всех остальных участников процесса, не позволяя отставать на полпути.
- Бенджамину Бергу (Benjamin Berg), нашему редактору; Джанет Вайль (Janet Vail), редактору по производству, фантастически коммуникабельной женщине, знающей все о том, как подготовить издание к печати; и всем сотрудникам издательства Manning, помогавшим нам в работе над книгой.
- Эрнесту Фридман-Хиллу (Ernest Friedman-Hill), консультанту по техническим иллюстрациям, который подал нам идеи, положенные в основу самых удачных рисунков в этой книге.
- Джону Дж. Райану (John J. Ryan) за скрупулезное техническое редактирование окончательного варианта рукописи непосредственно перед сдачей в печать.
- Всем рецензентам, детально проанализировавшим наш текст и код и предложившим упрощения и улучшения: Анне Эпштейн (Anne Epstein), Чарльзу Энгельке (Charles Engelke), Кэрти-

су Миллеру (Curtis Miller), Даниэлю Бретуа (Daniel Bretoi), Джеймсу Хэтуэю (James Hatheway), Джейсону Качору (Jason Kaszor), Кену Муру (Ken Moore), Кену Римплу (Ken Rimple), Кэвину Мартину (Kevin Martin), Лео Половцу (Leo Polovets), Марку Райалу (Mark Ryall), Марку Торрэнсу (Mark Torrance), Майку Гринхалгу (Mike Greenhalgh), Стэну Байсу (Stan Bice) и Вайятту Барнетту (Wyatt Barnett).

- Тысячам людей, купивших ранний вариант книги, участникам конференций и коллегам, которые побуждали нас оптимизировать представленные в книге решения.

Майк также благодарит:

- Джоша Пауэлла, который предложил мне написать эту книгу. Это были замечательная идея и великолепный опыт, многому научивший меня. Только вот нельзя ли вернуть потраченные годы жизни?
- Грега Бенсона за написание предисловия и за напоминание о том, что это слово пишется «foreword», а не «forward».
- Гаурава Дхиллона (Gaurav Dhillon), Джона Шустера (John Schuster), Стива Гудвина (Steve Goodwin), Джойса Лэма (Joyce Lam), Тима Лайкариша (Tim Likarish) и других сотрудников компании SnapLogic, которые понимают ценность экономности и элегантности решения.
- Анис Икбал (Anees Iqbal), Майкла Лортон (Michael Lorton), Дэвида Гуда (David Good) и других сотрудников компании GameCrush. Разработка продукта в GameCrush поставлена не идеально, но ничего более близкого к идеалу я не встречал.
- Моим родителям за то, что они купили мне компьютер, но отказались покупать к нему какой-нибудь софт. Это стало отличным поводом научиться программировать.
- Всем, кого я забыл упомянуть. Согласно закону Мэрфи, подраздел 8, я наверняка забыл кого-то очень важного, а вспомню только после выхода книги из печати. За это я искренне прошу прощения и надеюсь, что оно будет даровано.

Джош также благодарит:

- Майка Миковски за согласие совместно написать книгу. Я так рад, что мне не пришлось писать всю книгу самому. Тунеядец! Я имел в виду – спасибо большое.
- Луку Пауэлла, моего брата, который имеет мужество не отказываться от своей мечты, создать бизнес и быть самим собой. Он меня вдохновляет.

- Других членов моей семьи и друзей, без которых я не стал бы тем, кто я есть.
- Джона Келли (John Kelly) за то, что он позволил мне закончить книгу, понимая, что для такого дела требуется время. Да еще какое!
- Марку Торренсу (Mark Torrance) за то, что он наставлял меня, когда я создавал квалифицированную команду инженеров, и за свободу, предоставленную, когда я начал писать книгу.
- Уилсону Янгу (Wilson Yeung) и Дэйву Киферу (Dave Keefer), которые подталкивали меня к более глубокому изучению веб-технологий. Вы оказали огромное влияние на мою карьеру и знания в области программной инженерии.

Об этой книге

Задумывая эту книгу, мы собирались посвятить примерно две трети разработке клиентской части SPA. А в оставшейся трети мы планировали рассказать о веб-сервере и службах, необходимых для работы SPA. Но никак не могли выбрать конкретный веб-сервер. Нам пришлось писать десятки серверных веб-приложений – традиционных и одностраничных – с применением Ruby/Rails, Java/Tomcat, mod_perl и других платформ, но у всех были какие-то недостатки, особенно в плане поддержки SPA, заставлявшие нас желать большего.

Недавно мы перешли на «чистую» JavaScript-платформу: Node.js в качестве веб-сервера и MongoDB в качестве базы данных. Хотя кое-какие трудности имеются, мы считаем такое сочетание весьма привлекательным и раскрепощающим. Преимущества единого языка и общего формата данных обычно намного перевешивают потерю некоторых языковозависимых возможностей, присутствующих в «полиглотическом» стеке.

Мы решили, что знакомство с «чистым» JavaScript'овым стеком принесет нашим читателям наибольшую пользу, поскольку нам неизвестна никакая другая книга, в которой показано, как все эти компоненты собрать воедино. И мы полагаем, что эта платформа будет набирать популярность и станет одной из самых распространенных платформ для создания одностраничных приложений.

Структура книги

Глава 1 содержит введение в одностраничные приложения. Определяется, что такое JavaScript SPA, и проводится сравнение с другими видами SPA. SPA сравниваются с традиционными веб-сайтами, обсуждаются достоинства, возможности и проблемы, возникающие при использовании SPA. По ходу главы описывается постепенная разработка работоспособного SPA.

В главе 2 рассматриваются средства JavaScript, необходимые для создания SPA. Поскольку JavaScript изначально используется для написания почти всего кода SPA, а не добавляется в уже готовое приложение для реализации второстепенной интерактивности, чрезвычайно важно понимать, как устроен этот язык. Обсуждаются переменные, синтаксис и функции, а также более сложные темы: контекст выполнения, замыкания и прототипы объектов.

В главе 3 дается введение в архитектуру SPA, используемую в этой книге. Здесь же читатель знакомится с основным модулем пользо-

вательского интерфейса – Shell. Модуль Shell координирует работу специализированных модулей и событий браузера, а также содержит средства для работы с данными, в частности URL-адресами и куками. Демонстрируются реализация обработчика событий и использование паттерна якорного интерфейса для управления состоянием страницы.

В главе 4 детально рассматриваются функциональные модули, которые предоставляют SPA четко определенные возможности, ограниченные областью видимости. Проводится сравнение правильно написанных функциональных модулей со сторонним JavaScript-кодом. Подчеркивается важность изоляции для обеспечения качества и модульности кода.

В главе 5 описывается построение модуля Model, который консолидирует всю бизнес-логику в едином пространстве имен. Этот модуль изолирует своих клиентов от управления данными и взаимодействия с сервером. Здесь же проектируется и разрабатывается People API. Производится тестирование модели с помощью модуля подставных данных Fake и консоли JavaScript.

В главе 6 завершается работа над модулем Model. Проектируется и разрабатывается Chat API, который также тестируется с помощью модуля Fake и консоли JavaScript. Появляется модуль Data, и приложение модифицируется для работы с «живыми» данными от веб-сервера.

В главе 7 читатель знакомится с веб-сервером Node.js. Поскольку большая часть кода SPA находится на стороне клиента, серверная часть может быть написана на любом языке, достаточно производительном, чтобы справиться с запросами приложения. Написание серверной части на JavaScript обеспечивает единообразие сред программирования и упрощает разработку всей системы. Для тех, кто незнаком с Node.js, эта глава послужит отличным введением, а опытные пользователи Node.js узнают из нее о роли сервера в SPA.

В главе 8 мы продолжаем спускаться по стеку компонентов и переходим к базе данных. Мы используем MongoDB, потому что это проверенная на практике промышленная база данных, в которой данные хранятся в виде документов в формате JSON – том самом, который применяется для передачи данных клиентам. Прежде чем переходить к рассмотрению роли базы данных в SPA, мы приводим краткое введение в MongoDB для непосвященных.

В главе 9 обсуждаются некоторые концептуальные детали, отличающие SPA от традиционного веб-приложения в стиле MVC: оптимизация SPA для поисковых систем, сбор аналитических данных и

протоколирование ошибок. Мы также рассматриваем некоторые вопросы, интересные и для традиционных веб-приложений, но особенно важные при разработке SPA: быстрая доставка статического содержимого с помощью CDN-сетей и кэширование на всех уровнях стека.

Приложение А посвящено подробному рассмотрению применяемых нами стандартов кодирования на JavaScript; пригодятся они вам или нет, вам решать, но мы находим их чрезвычайно полезными для структурирования JavaScript-кода в одностраничном приложении, потому что они обеспечивают тестопригодность, простоту сопровождения и удобочитаемость. Мы объясняем, почему стандарты кодирования так важны, описываем способы организации и документирования кода, подход к именованию переменных и методов, защиту пространств имен, организацию файлов и использование программы JSLint для проверки JavaScript-кода.

Приложение Б посвящено тестированию SPA. На эту тему можно было бы написать отдельную книгу, но она настолько важна, что мы просто не смогли оставить ее без внимания. Мы рассматриваем настройку режима тестирования, выбор каркаса тестирования, создание комплекта тестов и учет тестирования при написании модулей SPA.

Предполагаемая аудитория

Эта книга рассчитана на веб-разработчиков, архитекторов и менеджеров продуктов, имеющих хотя бы поверхностные знания о JavaScript, HTML и CSS. Если вы никогда не занимались веб-разработкой, то эта книга не для вас, хотя мы будем рады, если вы ее все равно купите (давайте, давайте, папочке нужна новая машина). Есть немало хороших книг, в которых повествуется об азах проектирования и разработки сайтов, но эта не из их числа.

Назначение этой книги – стать полезным руководством по проектированию и построению крупномасштабных одностраничных веб-приложений (SPA), в которых на всех уровнях стека применяется JavaScript. Мы используем язык JavaScript и в базе данных, и на веб-сервере, и в браузерном приложении. Примерно две трети книги посвящено разработке клиентской части. А в оставшейся трети показано, как на JavaScript написать серверную часть, воспользовавшись Node.js и MongoDB. Если вы вынуждены работать на другой серверной платформе, то большую часть логики можно будет легко перенести, но для реализации обмена сообщениями все-таки необходим событийно-управляемый веб-сервер.

Графические выделения и загрузка исходного кода

Весь исходный код в листингах и в тексте выделяется моноширинным шрифтом. Листинги сопровождаются аннотациями, иллюстрирующими основные идеи.

Весь исходный код, представленный в этой книге, можно скачать с сайта издательства по адресу www.manning.com/SinglePageWebApplications.

Требования к программному обеспечению и оборудованию

У тех, кто работает с последними версиями Mac OS X или Linux, не должно возникнуть проблем с приведенными в книге примерами, при условии что будет установлено все упоминаемое нами программное обеспечение.

Если вы работаете с Windows, то при выполнении упражнений из частей I и II вряд ли возникнут сложности. Но в части III используются программы, которые на платформе Windows либо отсутствуют, либо имеют ограничения. Мы рекомендуем воспользоваться какой-нибудь бесплатной виртуальной машиной (см. <http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html>) и дистрибутивом Linux (мы рекомендуем Ubuntu Server 13.04, см. <http://www.ubuntu.com/download/server>).

Автор в сети

Приобретение книги «Одностраничные веб-приложения» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/SinglePageWebApplications. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не

означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору какие-нибудь хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

Об авторах

Майкл С. Миковски – удостоенный наград промышленный дизайнер и архитектор SPA с 13-летним опытом работы в качестве архитектора и разработчика всех компонентов веб-приложений. Четыре года он работал руководителем разработки на высокопроизводительной и высокодоступной платформе, обслуживающей сотни миллионов запросов в день и состоящей из большого кластера серверов приложений, написанных на `mod_perl`.

Он начал работать над коммерческими одностраничными веб-приложениями в 2007 году, когда разрабатывал сайт «Где купить» для компании AMD, а ограничения по хостингу исключали почти все другие решения. Затем, воодушевленный возможностями, которые открывали SPA, он спроектировал и разработал еще много подобных решений. Он твердо верит, что проектирование с учетом качества, «творческое разрушение»¹, минимализм и целенаправленное тестирование могут устранить сложности и непонимание, связанные с разработкой SPA.

Майк участвует во многих проектах с открытым исходным кодом и опубликовал ряд подключаемых модулей для jQuery. Он выступал с докладами на конференциях разработчиков по HTML5 в 2012 и 2013 годах, на конференции Developer Week 2013 года в Университете Сан-Франциско и на семинарах в различных компаниях. В последнее время занимается архитектурой пользовательских интерфейсов, консультированием и инженерией пользовательского восприятия.

Джош К. Пауэлл занимается вебом еще с тех пор, когда IE 6 считался хорошим браузером. Имея 13-летний опыт работы в области программной инженерии и веб-архитектуры, он обожает разрабатывать веб-приложения и организовывать команды для этого дела. В настоящее время увлечен экспериментами с различными технологиями

¹ См. http://ru.wikipedia.org/wiki/Креативное_разрушение. – *Прим. перев.*

построения одностраничных веб-приложений и отдает этому все свое время.

По странной прихоти природы, он получает заряд энергии от выступлений на публике. Он проводил презентации, посвященные одностраничным приложениям и JavaScript на различных конференциях, в том числе HTML 5 Developers Conference и NoSQL Now!, в университетах и в таких компаниях из Кремниевой долины, как Engine Yard, RocketFuel, и многих других. Он также пишет статьи для сайта www.learningjquery.com и различных сетевых журналов.

Об иллюстрации на обложке

Рисунок на обложке книги называется «Gobenador de la Abisinia», или Губернатор Абиссинии, нынешней Эфиопии. Он взят из испанского собрания местных костюмов, впервые изданного в Мадриде в 1799 году. На титульном листе книги имеется следующая надпись:

Coleccion general de los Trages que usan actualmente todas las Naciones del Mundo desubierto, dibujados y grabados con la mayor exactitud por R.M.V.A.R. Obra muy util y en special para los que tienen la del viajero universal.

В переводе это означает:

Всеобщая коллекция костюмов, ныне используемых народами известного мира, подготовленная и напечатанная со всей возможной точностью R. M. V. A. R. Этот труд особенно полезен тем, кто считает себя путешественником по миру.

Хотя ничего не известно о художниках, граверах и рабочих, которые вручную раскрашивали этот рисунок, бросается в глаза, что он действительно выполнен «со всей возможной точностью». «Gobenador de la Abisinia» – лишь один из множества рисунков в этой цветной коллекции. Их разнообразие – красноречивое свидетельство неповторимости и самобытности костюмов, которые носили народы разных стран 200 лет назад.

Издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из этого собрания.

Введение в SPA

За то время, что вам требуется для чтения этой страницы, будет потрачено 35 млн человеко-минут на ожидание загрузки традиционных веб-страниц. Этого достаточно, чтобы 96 раз доставить марсоход Curiosity на Марс и обратно. Отношение издержек к производительности, характерное для традиционных сайтов, изумляет, и для бизнеса это может иметь самые печальные последствия. Наткнувшись на медленный сайт, пользователь уйдет – прямо к гостеприимно распахнутым кошелькам конкурентов.

Одна из причин медленной работы традиционных сайтов заключается в том, что популярные серверные каркасы для создания MVC-приложений «заточены» под передачу статических страниц «тупому» по сути своей клиенту. Например, когда мы щелкаем по ссылке на традиционном сайте, демонстрирующем слайд-шоу, экран мигает, и в течение нескольких секунд происходит перезагрузка всей страницы целиком: элементов навигации, рекламных баннеров, верхнего и нижнего колонтитулов и текста. Хотя изменилась только текущая фотография и, быть может, пояснительный текст. Хуже того, нет никакого индикатора, показывающего, что некий элемент на странице снова готов к взаимодействию с пользователем. Например, иногда по ссылке можно щелкнуть сразу после ее появления на странице, а иногда приходится ждать, пока вся страница перерисовется, а потом еще пяток секунд. Такое медленное, непоследовательное и неэффективное функционирование становится неприемлемым для все более требовательных пользователей веб.

Приготовьтесь к знакомству с другим – и, смеем сказать, лучшим – подходом к разработке веб-приложений – одностраничными приложениями (SPA). SPA переносят в браузер приемы работы, привычные для персональных приложений. В результате мы получаем быструю реакцию, которая не раздражает, а, наоборот, удивляет и восхищает пользователей. В первой части книги мы рассмотрим следующие вопросы.

- Что такое SPA и какие у него преимущества, по сравнению с традиционными сайтами?

- Как подход на основе SPA позволяет резко сократить время реакции приложения и сделать его притягательным?
- Как отточить навыки работы с JavaScript, необходимые для разработки SPA?
- Как создать демонстрационное SPA?

Дизайн продукта все чаще становится решающим фактором, определяющим успех или провал коммерческого, да и корпоративного веб-приложения. Одностраничные приложения нередко обеспечивают оптимальный, с точки зрения пользователя, способ работы. Поэтому мы ожидаем, что спрос на дизайн с упором на пользователя станет причиной более широкого распространения и большей изощренности SPA.

Глава 1

Наше первое одностраничное приложение

В этой главе:

- ✧ Определение одностраничного приложения.
- ✧ Сравнение популярных платформ для создания одностраничных приложений – Java, Flash, JavaScript.
- ✧ Создание первого одностраничного приложения на JavaScript.
- ✧ Исследование приложения с помощью инструментов разработчика в Chrome.
- ✧ Преимущества одностраничных приложений с точки зрения пользователя.

Эта книга рассчитана на веб-разработчиков, архитекторов и менеджеров продуктов, имеющих хотя бы поверхностные знания о JavaScript, HTML и CSS. Если вы никогда не занимались веб-разработкой, то эта книга *не* для вас, хотя мы будем рады, если вы ее все равно купите (давайте, давайте, папочке нужна новая машина). Есть немало хороших книг, в которых повествуется об азах проектирования и разработки сайтов, но эта не из их числа.

Назначение данной книги – стать полезным руководством по проектированию и построению крупномасштабных одностраничных веб-приложений (SPA), в которых на всех уровнях стека применяется JavaScript. Как видно по рис. 1.1, мы используем JavaScript в качестве языка базы данных, веб-сервера и браузерного приложения.

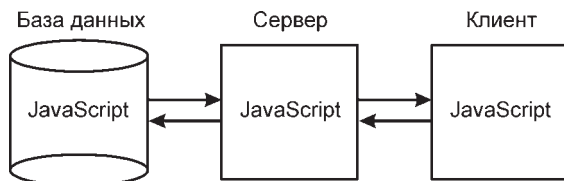


Рис. 1.1 ✧ JavaScript на всех уровнях стека

Последние шесть лет мы руководили разработкой многочисленных крупномасштабных коммерческих и корпоративных SPA. И все это время мы не прекращали оттачивать свои навыки, чтобы во всеоружии встречать новые задачи. В этой книге мы поделимся накопленным опытом того, как разрабатывать приложения быстрее, делать работу пользователя удобнее, повышать качество и улучшать взаимоотношения в коллективе.

1.1. Определение, немного истории и несколько слов о предмете книги

SPA – это приложение, которое работает в браузере и не перезагружает страницу во время работы. Как и любое другое приложение, оно предназначено для того, чтобы помочь пользователю в решении стоящей перед ним задачи, например: «подготовить документ» или «администрировать веб-сервер». Мы можем рассматривать SPA как толстый клиент, загружаемый с веб-сервера.

1.1.1. Немного истории

SPA существуют уже довольно давно. Рассмотрим несколько примеров.

- *Крестики-нолики* – <http://rintintin.colorado.edu/~epperson/Java/TicTacToe.html>. Ну мы же не говорили, что приложение должно быть красивым. Конкретно это SPA предлагает нам обыграть великий, ужасный и ничего не прощающий компьютер в крестики-нолики. Для его работы необходим подключаемый модуль Java – см. <http://www.java.com/en/download/index.jsp>. Возможно, придется разрешить браузеру запуск этого апплета.
- *Посади корабль* – <http://games.whomwah.com/spacelander.html>. Это одна из первых Flash-игр, написанная Дунканом Робертсоном (Duncan Robertson) примерно в 2001 году. Для его работы необходим подключаемый модуль Flash – см. <http://get.adobe.com/flashplayer/>.
- *Ипотечный калькулятор на JavaScript* – <http://www.mcfedries.com/creatingawebpage/mortgage.htm>. Этому калькулятору, наверное, столько же лет, сколько самому JavaScript, но к его работе нет никаких нареканий. *Никаких подключаемых модулей не требуется.*

Внимательные – да даже и рассеянные¹ – читатели, наверное, заметили, что в качестве примеров мы взяли три самые популярные платформы разработки SPA: Java-апплеты, Flash/Flex и JavaScript. И те же самые читатели, вероятно, обратили внимание, что *только SPA на JavaScript работают без издержек и сложностей, относящихся к безопасности, которые характерны для сторонних подключаемых модулей*.

На сегодняшний день написанные на JavaScript SPA – лучший вариант из трех. Но чтобы стать конкурентоспособным средством создания SPA, да и вообще претендовать на эту роль, JavaScript прошел долгий путь. Давайте посмотрим, почему.

1.1.2. Почему SPA на JavaScript появились так поздно?

Flash и Java-апплеты пришли к 2000 году уже достаточно развитыми. Язык Java использовался для доставки через браузер сложных приложений и даже целых офисных пакетов². Технология Flash стала наиболее популярной платформой для создания браузерных игр, а позднее видеопроигрывателей. С другой стороны, JavaScript использовался в основном для таких простых задач, как ипотечные калькуляторы, контроль данных в формах, эффекты при наведении мыши и всплывающие окна. Проблема заключалась в том, что мы не могли полагаться на JavaScript (и используемые в нем методы отрисовки) как на средство единообразной реализации критических для приложения функций во всех популярных браузерах. Но даже и при таком положении вещей написание SPA на JavaScript обещало ряд заманчивых преимуществ над Flash и Java.

- *Отсутствие необходимости в подключаемых модулях* – для доступа к приложению пользователям не нужно думать об установке и сопровождении подключаемых модулей и о совместимости с ОС. А программистам нет нужды забивать голову еще одной моделью безопасности со всеми вытекающими из нее сложностями в плане разработки и сопровождения³.

¹ Если, читая эту главу, вы одновременно жуete чипсы, просыпая крошки себе на грудь, то вас можно назвать рассеянным.

² Applix (VistaSource) Anywhere Office.

³ Вам знакомы слова «правило ограничения домена»? Если вы когда-нибудь разрабатывали приложения на Flash или Java, то почти наверняка знакомы с этой проблемой.

- *Меньше расход памяти* – для SPA, написанного с помощью JavaScript и HTML, наверняка нужно куда меньше ресурсов, чем для подключаемого модуля, нуждающегося в отдельной исполняющей среде.
- *Один клиентский язык* – веб-архитекторам и большинству разработчиков приходится изучать много языков и форматов данных: HTML, CSS, JSON, XML, JavaScript, SQL, PHP/Java/Ruby/Perl и т. д. Так зачем писать апплеты на Java или Flash-приложения на ActionScript, если мы уже используем на страницах JavaScript? Применение только одного языка программирования на стороне клиента – замечательный способ снизить сложность.

- *Более подвижная и интерактивная страница* – все мы видели, как выглядят Flash- или Java-приложения на веб-странице. Часто приложение размещается в какой-то прямоугольной области, где детали существенно отличаются от окружающих HTML-элементов: другие графические виджеты, другое поведение при нажатии правой кнопкой мыши, другие звуки, ограничения на взаимодействие с прочими частями страницы. В случае SPA, написанного на JavaScript, интерфейсом приложения является все окно браузера.

По мере становления JavaScript его недостатки были в основном исправлены или смягчены, а достоинства выступили на передний план.

- *Веб-браузер – самое распространенное в мире приложение.* У многих пользователей окно браузера открыто на протяжении всего рабочего дня. Для доступа к JavaScript-приложению достаточно одного щелчка по закладке.
- *JavaScript в браузере – одна из самых распространенных в мире сред выполнения.* По данным на декабрь 2011 года, ежедневно активировалось примерно миллион мобильных устройств на платформах Android и iOS. На каждом таком устройстве имеется стабильная среда выполнения JavaScript, встроенная в ОС. За последние три года в мире было продано более миллиарда стабильных реализаций JavaScript в телефонах, планшетах, ноутбуках и настольных ПК.
- *Развертывание JavaScript-приложения – тривиальная задача.* Чтобы сделать JavaScript-приложение доступным миллиарду пользователей, достаточно разместить его на HTTP-сервере.

- *JavaScript полезен для кросс-платформенной разработки.* Теперь у нас есть возможность создавать SPA в своей любимой операционной системе, будь то Windows, Mac OS X или Linux, и развертывать его не только на любом настольном ПК, но и на смартфонах и планшетах. Мы можем полагаться на совместимые реализации стандартов во всех браузерах и на достигшие зрелости библиотеки типа jQuery и PhoneGap, скрывающие оставшиеся различия.
- *JavaScript стал работать на удивление быстро и в некоторых случаях может соперничать даже с компилируемыми языками.* Такое ускорение – следствие непрекращающейся ожесточенной конкуренции между Mozilla Firefox, Google Chrome, Opera и Microsoft IE. В современных реализациях JavaScript применяются такие передовые методы оптимизации, как JIT-компиляция в машинный код, предсказание ветвлений, выведение типа и многопоточность¹.
- *В язык JavaScript включены дополнительные возможности.* К ним относятся встроенный объект JSON, встроенные селекторы по образцу jQuery и более последовательно реализованные средства AJAX. Такие библиотеки, как Strophe и Socket.IO, существенно упростили «проталкивание» сообщений от сервера клиенту.
- *Стандарты HTML5, SVG и CSS3 и их поддержка в браузерах продолжают развиваться.* Благодаря такому развитию отрисовка графики с пиксельной достоверностью может по скорости и качеству соперничать с Java и Flash.
- *JavaScript можно использовать на всех уровнях проекта веб-приложения.* Теперь у нас есть великолепный веб-сервер Node.js и хранилища данных типа CouchDB или MongoDB, умеющие передавать и хранить данные в формате JSON, «родном» для JavaScript. Мы даже можем использовать на стороне браузера и сервера одни и те же библиотеки.
- *Настольные ПК, ноутбуки и даже мобильные устройства становятся все более мощными.* Повсеместное наличие многоядерных процессоров и гигабайтов памяти означает, что обработку, которая раньше производилась на сервере, можно перенести на сторону клиента в браузере.

¹ О сравнении с Flash ActionScript 3 см. <http://iq12.com/blog/as3-benchmark/> и <http://jacksondunstan.com/articles/1636>.

Благодаря указанным преимуществам написанные на JavaScript SPA становятся все популярнее, а следовательно, растет спрос на опытных разработчиков и архитекторов, уверенно владеющих JavaScript. Приложения, которые раньше разрабатывались для нескольких операционных систем (или на Java, или Flash), теперь принимают вид единого JavaScript SPA. Новые компании выбирают Node.js в качестве веб-сервера, а разработчики мобильных приложений используют JavaScript и PhoneGap для создания приложений, которые выглядят как «родные» на разных платформах, хотя код единый.

JavaScript неидеален, и не надо далеко ходить в поисках упущений, несогласованностей и прочих минусов. Но то же можно сказать о любом языке. Освоив базовые принципы, применяя рекомендованные приемы и поняв, чего следует избегать, вы начнете получать от разработки на JavaScript удовольствие и сможете повысить свою продуктивность.

Сгенерированный JavaScript-код: одна цель, два пути

На наш взгляд, проще писать JavaScript-код SPA вручную. Такие SPA мы называем *естественными* (native). Другой, на удивление популярный, подход – *генерация* JavaScript-кода, когда разработчик пишет код на другом языке, а затем транслирует его на JavaScript. Трансляция производится либо во время выполнения, либо в ходе отдельного этапа генерации. К числу широко известных генераторов JavaScript относятся:

- *Google Web Toolkit (GWT)* – см. <http://code.google.com/webtoolkit/>. GWT генерирует JavaScript-код из кода на Java;
- *Cappuccino* – см. <http://cappuccino.org/>. Входным языком для Cappuccino является Objective-J – клон языка Objective-C, используемого в Mac OS X. Сам Cappuccino – это перенесенная система разработки приложений Cocoa, применяемая все в той же OS X;
- *CoffeeScript* – см. <http://coffeescript.org/>. CoffeeScript – это специальный язык, созданный на основе JavaScript, но с некоторыми синтаксическими дополнениями.

Вспомнив, что Google использует GWT в проектах Blogger, Google Groups и на многих других сайтах, мы можем с уверенностью сказать, что SPA со сгенерированным JavaScript-кодом широко распространены. В этой связи возникает вопрос: *зачем писать код на одном языке высокого уровня, чтобы затем транслировать его на другой?* Есть ряд причин, по которым генерация JavaScript остается популярным решением, хотя сейчас они выглядят не так убедительно, как когда-то.

- *Знакомство* – разработчики стремятся использовать более знакомый или более простой язык. Генератор и инфраструктура позволяют им писать код, не думая о причудах JavaScript. Беда в том, что при трансляции иногда что-то теряется. Когда такое случается, разработчик вынужден изучать сгенерированный JavaScript-код, чтобы понять, как исправить ситуацию. Мы работаем более эффективно, когда пишем непосредственно на JavaScript, минуя уровень его абстрагирования.

- *Инфраструктура* – разработчики ценят хорошо продуманную систему согласованных библиотек для сервера и клиента, предоставляемую GWT. Это убедительный аргумент, особенно если команда уже наработала большой опыт и набор эксплуатируемых продуктов.
- *Несколько целевых платформ* – генератор может порождать разные файлы, например один для Internet Explorer, а другой – для всех остальных браузеров. И хотя генерация кода для разных платформ – на первый взгляд, привлекательная идея, мы полагаем, что гораздо эффективнее развернуть единый исходный JavaScript-код, работающий во всех браузерах. Благодаря постепенному устранению различий между браузерами и наличию зрелых кросс-браузерных библиотек типа jQuery теперь стало значительно проще писать сложные SPA, работающие без модификации во всех основных браузерах.
- *Зрелость* – разработчики считают, что JavaScript недостаточно структурирован для создания крупномасштабных приложений. Однако со временем язык JavaScript значительно улучшился, его сильные стороны вызывают уважение, а слабые вполне контролируемы. Программисты, привыкшие писать на строго типизированных языках, например Java, иногда не могут простить отсутствие типобезопасности. А некоторые разработчики, знакомые с такими всеобъемлющими каркасами, как Ruby on Rails, жалуются на кажущееся отсутствие структуры. К счастью, эти проблемы можно сгладить, применяя инструменты проверки кода, стандарты кодирования и зрелые библиотеки.

Мы полагаем, что в настоящее время естественные JavaScript SPA обычно более предпочтительны. И именно такого подхода мы будем придерживаться в этой книге.

1.1.3. Предмет книги

В этой книге речь пойдет о том, как разрабатывать привлекательные, надежные, масштабируемые и удобные в сопровождении SPA, используя JavaScript на всех уровнях стека¹. Если явно не оговорено противное, то, начиная с этого места, мы под SPA будем понимать естественное JavaScript SPA, в котором бизнес-логика и презентационная логика написаны непосредственно на JavaScript и которое исполняется в браузере. JavaScript-код отрисовывает пользовательский интерфейс с помощью таких технологий, как HTML5, CSS3, Canvas и SVG.

В SPA могут использоваться любые серверные технологии. Поскольку значительная часть веб-приложения перемещается в браузер, требования к серверу можно существенно ослабить. На рис. 1.2 показано, как бизнес-логика и генерация HTML-кода переносятся с сервера на клиент.

¹ Эту книгу можно было бы также назвать «Передовые методы создания одностраничных веб-приложений», но это слишком длинно.

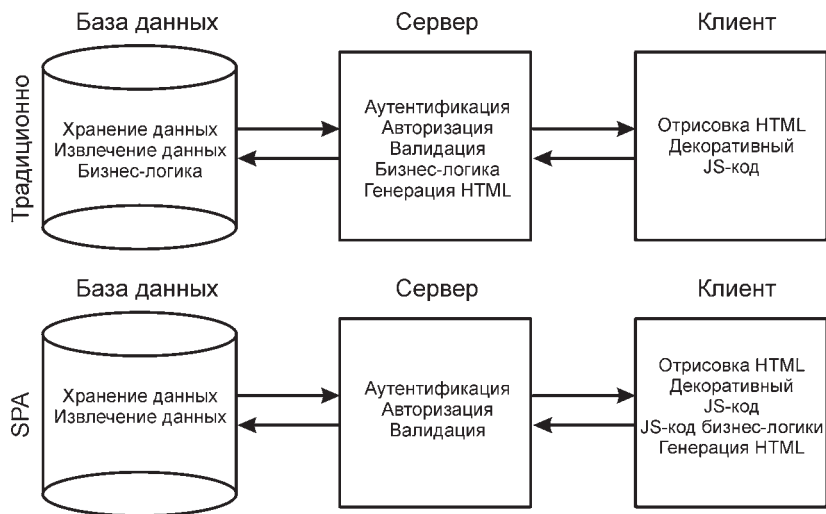


Рис. 1.2 ❖ Сферы ответственности базы данных, сервера и клиента

Серверная часть рассматривается в главах 7 и 8, где мы используем веб-сервер и базу данных, программируемые на JavaScript. Возможно, на вашей машине нет этих программ или вы предпочитаете другое решение. Ничего страшного – большинство принципов и приемов построения SPA, рассматриваемых в этой книге, применимы безотносительно к используемым серверным технологиям. Но если вы хотите использовать JavaScript на всех уровнях стека, то у нас есть как раз то, что вам нужно.

Что касается клиентских библиотек, то мы используем jQuery для манипуляций объектной моделью документа (DOM), а также подключаемые модули для управления историей и обработки событий. Для реализации высокопроизводительных, ориентированных на обработку данных моделей мы применяем библиотеку TaffyDB. Библиотека Socket.IO обеспечивает естественный механизм обмена сообщениями между клиентом и сервером, работающий почти в реальном масштабе времени. В качестве веб-сервера мы используем событийно-управляемый сервер Node.js. Он построен на базе движка JavaScript Google V8 и способен обрабатывать десятки тысяч одновременных соединений. Библиотека Socket.IO используется также на сервере. В роли базы данных у нас выступает MongoDB, принадлежащая к классу NoSQL, в которой данные хранятся в формате JSON, «родном» для JavaScript, а в API и в интерфейсе командной строки

также используется JavaScript. Все это популярные проверенные на практике решения.

При разработке SPA приходится писать как минимум на порядок больше кода на JavaScript, чем в случае традиционного сайта, поскольку значительная часть логики приложения перемещается с сервера в браузер. Над созданием одного SPA могут одновременно работать много программистов, а объем кода вполне может превысить 100 000 строк. При таком масштабе становятся абсолютно необходимыми следование соглашениям и соблюдение дисциплины программирования, ранее считавшиеся обязательными только для разработки серверной части. С другой стороны, серверное ПО упрощается, на его долю остаются только аутентификация, валидация и службы данных. Помните об этом, прорабатывая примеры.

1.2. Создаем наше первое SPA

Настало время заняться разработкой SPA. Мы будем применять передовые методы и объяснять, что делаем на каждом шаге.

1.2.1. Постановка задачи

У нашего первого SPA будет скромная задача: отобразить в правом нижнем углу окна браузера всплывающее окно чата по аналогии с тем, что вы, возможно, видели в Gmail или Facebook. Сразу после загрузки приложения это окно свернуто, а при щелчке по нему – выплывает, как показано на рис. 1.3. При повторном щелчке окно чата снова сворачивается.

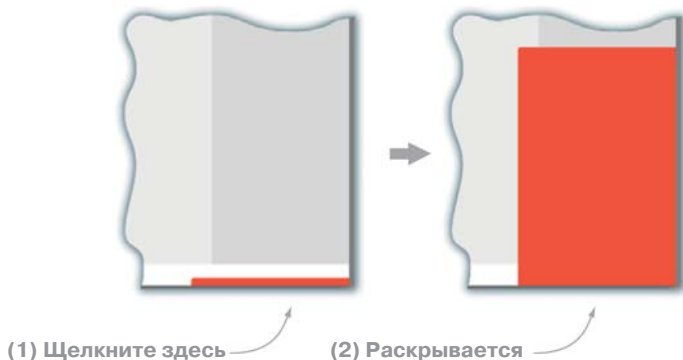


Рис. 1.3 ❖ Окно чата в свернутом и раскрытом состояниях

Обычно SPA делает куда больше, чем открытие и закрытие окна чата, – например, посылает в чат и принимает из него сообщения. Но эти досадные детали мы опустим, поскольку введение должно быть простым и кратким. Перефразируя известную поговорку, можно сказать, что SPA не один день строилось. Но не расстраивайтесь, мы еще вернемся к отправке и получению сообщений в главах 6 и 8.

В следующих разделах мы подготовим файл для разработки SPA, познакомимся с некоторыми из наших любимых инструментов, напишем код для отображения всплывающего чата и остановимся на некоторых рекомендуемых приемах. Вам предстоит усвоить довольно много материала, и мы не ждем, что вы с первого раза все поймете – особенно некоторые хитроумные трюки JavaScript. Каждому из этих вопросов будет уделено внимание в последующих главах, а пока расслабьтесь, не берите в голову и познакомьтесь с местностью.

1.2.2. Общая структура файла

Все наше приложение будет находиться в одном файле `spa.html`, а единственной внешней библиотекой будет jQuery. Обычно рекомендуется помещать код на CSS и на JavaScript в разные файлы, но в начале разработки и в примерах удобнее работать с одним файлом. Прежде всего отведем место для стилей и JavaScript-кода. Еще нам понадобится контейнер `<div>`, в который приложение будет помещать объекты HTML. Таким образом, получается структура, показанная в листинге 1.1:

Листинг 1.1 ❖ Заготовка – `spa.html`

```
<!doctype html>
<html>
<head>
  <title>SPA Chapter 1 section 1.2.2</title>
  <style type="text/css"></style>
  <script type="text/javascript"></script>
</head>
<body>
  <div id="spa"></div>
</body>
</html>
```

Добавляем тег `style`, внутри которого будут находиться CSS-селекторы. Рекомендуется загружать CSS раньше JavaScript, потому что обычно страница при этом отрисовывается быстрее.

Добавляем тег `script`, внутри которого будет находиться наш JavaScript-код.

Создаем `div` с идентификатором `spa`. Содержимым этого контейнера будем управлять из JavaScript.

Подготовив файл, настроим инструменты разработчика в Chrome, чтобы можно было изучать текущее состояние приложения.

1.2.3. Настройка инструментов разработчика в Chrome

Откроем файл `spa.html` в браузере Google Chrome. Мы должны увидеть пустое окно, так как никакого содержимого еще нет. Однако под капотом кое-что происходит. Воспользуемся инструментами разработчика в Chrome, чтобы понять, что именно.

Чтобы открыть инструменты разработчика, нужно щелкнуть по значку с изображением трех горизонтальных полос в правом верхнем углу Chrome, выбрать из меню пункт **Инструменты**, а затем **Инструменты разработчика**. Появится панель, изображенная на рис. 1.4. Если консоль JavaScript отсутствует, ее можно показать, щелкнув по кнопке **Show console** в левом нижнем углу. Консоль должна быть пустой, это означает, что никаких ошибок или предупреждений JavaScript нет. Это естественно, так как мы еще не добавили свой JavaScript-код. В секции **Elements** над консолью JavaScript показаны HTML-код и структура нашей страницы.

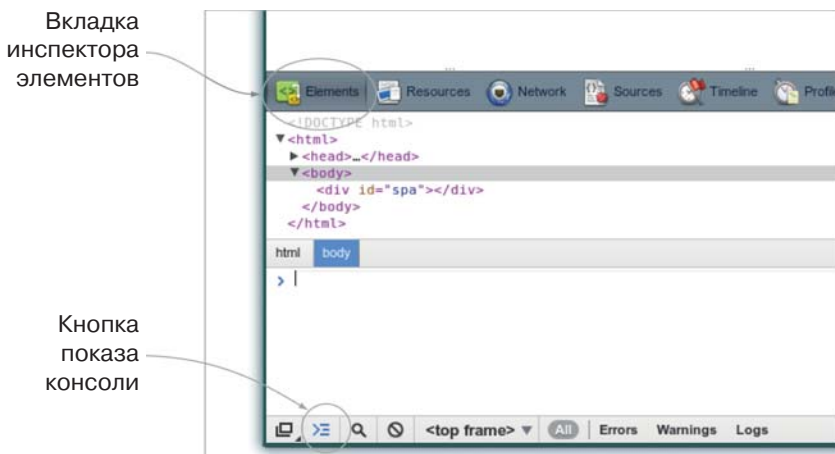


Рис. 1.4 ❖ Инструменты разработчика в Google Chrome

В этой книге мы пользуемся инструментами разработчика, встроенными в Chrome, но аналогичные возможности имеются и в других браузерах. Например, в Firefox есть Firebug, а в IE и Safari – собственные версии инструментов разработчика.

Теперь напомним HTML и CSS для нашей страницы.

1.2.4. Разработка HTML и CSS

Мы должны добавить в HTML-разметку единственный контейнер, содержащий окно чата. Начнем с определения стилей контейнеров в разделе `<style>` в файле `spa.html`. Стили показаны в следующем листинге.

Листинг 1.2 ❖ HTML и CSS – spa.html

```
<!doctype html>
<html>
<head>
  <title>SPA Chapter 1 section 1.2.4</title>
  <style type="text/css">
    body {
      width    : 100%;
      height   : 100%;
      overflow : hidden;
      background-color : #777;
    }
    #spa {
      position : absolute;
      top      : 8px;
      left     : 8px;
      bottom   : 8px;
      right    : 8px;
      border-radius : 8px 8px 0 8px;
      background-color : #fff;
    }
    .spa-slider {
      position : absolute;
      bottom   : 0;
      right    : 2px;
      width    : 300px;
      height   : 16px;
      cursor   : pointer;
      border-radius : 8px 0 0 0;
      background-color : #f00;
    }
  </style>
  <script type="text/javascript"></script>
</head>
<body>
  <div id="spa">
    <div class="spa-slider"></div>
  </div>
</body>
</html>
```

Говорим, что тег `<body>` должен заполнять все окно браузера и скрывать то, что не поместилось. Задаем также серый цвет фона.

Определяем контейнер, в котором будет находиться все содержимое нашего SPA.

Определяем класс `spa-slider`, так чтобы контейнер, содержащий окно чата, был прикреплен к правому нижнему углу объемлющего контейнера. Устанавливаем красный цвет фона и скругляем левый верхний угол.

Открыв файл `spa.html` в браузере, мы увидим, что окно чата свернуто, как показано на рис. 1.5. Мы используем эластичную верстку, при

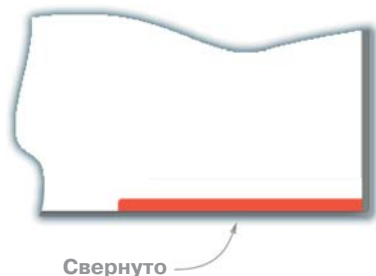


Рис. 1.5 ❖ Окно чата свернуто – spa.html

которой интерфейс адаптируется к размеру окна, а чат всегда находится в правом нижнем углу. Мы не стали обрамлять контейнер, потому что рамка увеличивает ширину контейнера и может осложнить разработку, так как пришлось бы изменить размеры контейнеров, чтобы учесть рамки. Рамки удобно добавлять после того, как основной макет уже сверстан и проверен; мы будем так делать в следующих главах.

Определившись с визуальными элементами, мы можем перейти к написанию JavaScript-кода, который сделает страницу интерактивной.

1.2.5. Добавление JavaScript-кода

Мы хотим применять в своем JavaScript-коде самые передовые методы. В этом нам поможет, в частности, инструмент JSLint, написанный Дугласом Крокфордом. JSLint проверяет программу на предмет соблюдения разнообразных рекомендаций по кодированию на JavaScript. Кроме того, мы собираемся использовать созданную Джоном Резигом библиотеку jQuery для манипулирования объектной моделью документа (DOM). jQuery предоставляет простые кросс-браузерные средства, позволяющие анимировать выплывание окна чата.

Прежде чем приступить к написанию JavaScript-кода, решим, что мы хотим сделать. Первый тег `script` загружает библиотеку jQuery. Второй тег `script` будет содержать *наш* JavaScript-код, который мы разобьем на три части.

1. Заголовок, в котором объявляются параметры JSLint.
2. Функция `spa`, которая создает окно чата и управляет им.
3. Строка, в которой функция `spa` вызывается, как только объектная модель документа будет готова к работе.

Рассмотрим внимательнее, что должна делать функция `spa`. По опыту мы знаем, что потребуется секция, в которой мы будем объявлять переменные модуля и конфигурационные константы. Нам понадобится функция, переключающая состояние выплывающего чата. А также функция, которая принимает событие щелчка мышью и вызывает функцию переключения. Наконец, необходима еще функция, которая инициализирует состояние приложения. Оформим этот план в виде заготовки.

Листинг 1.3 ❖ Разработка JavaScript-кода, первый проход – `spa.html`

```
/* Параметры jshint */

// Модуль /spa/
// Обеспечивает функциональность выплывающего чата
// Переменные в области видимости модуля
// Задать константы
// Объявить все прочие переменные в области видимости модуля

// Метод DOM /toggleSlider/
// изменяет высоту окна чата

// Обработчик события /onClickSlider/
// получает событие щелчка и вызывает toggleSlider

// Открытый метод /initModule/
// устанавливает начальное состояние и предоставляет функциональность
// отрисовать HTML
// инициализировать высоту и описание окна чата
// привязать обработчик к событию щелчка мышью

// запустить spa, когда модель DOM будет готова
```

Начало положено! Оставим все комментарии и добавим сам код. Для наглядности мы выделили комментарии полужирным шрифтом.

Листинг 1.4 ❖ Разработка JavaScript-кода, второй проход – `spa.html`

```
/* Параметры jshint */

// Модуль /spa/
// Обеспечивает функциональность выплывающего чата
//
var spa = (function ( $ ) {
    // Переменные в области видимости модуля
    var
        // Задать константы
        configMap = { },
        // Объявить все прочие переменные в области видимости модуля
```

```
$chatSlider,
toggleSlider, onClickSlider, initModule;

// Метод DOM /toggleSlider/
// изменяет высоту окна чата
//
toggleSlider = function () {};

// Обработчик события /onClickSlider/
// получает событие щелчка и вызывает toggleSlider
onClickSlider = function ( event ) {};

// Открытый метод /initModule/
// устанавливает начальное состояние и предоставляет функциональность
//
initModule = function ( $container ) {
    // отрисовать HTML
    // инициализировать высоту и описание окна чата
    // привязать обработчик к событию щелчка мышью
};
})();

// запустить spa, когда модель DOM будет готова
```

Последний проход по файлу `spa.html` показан в листинге 1.5. Мы загружаем библиотеку jQuery, затем включаем собственный JavaScript-код, который содержит заданные нами параметры JSLint, наш модуль `spa` и строку, запускающую этот модуль по готовности модели DOM. Модуль `spa` теперь полностью работоспособен. Не расстраивайтесь, если вы не все поняли, — здесь много чего происходит, и все это мы подробно разберем в последующих главах. А это лишь пример, показывающий, что можно сделать.

Листинг 1.5 ❖ Разработка JavaScript-кода, третий проход – `spa.html`

```
<!doctype html>
<html>
<head>
  <title>SPA Chapter 1 section 1.2.5</title>
  <style type="text/css">
...
  </style>

  <script type="text/javascript" src=
    "http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">
  </script>

  <script type="text/javascript">
```

Загружаем библиотеку jQuery из сети доставки содержимого Google (CDN), что снижает нагрузку на наши серверы и часто оказывается быстрее. Поскольку много других сайтов тоже загружают jQuery из Google CDN, то высока вероятность, что браузер пользователя уже кэшировал ее и сможет использовать, не отправляя HTTP-запроса.

```

/*jslint      browser : true, continue : true,
  devel : true, indent : 2,      maxerr : 50,
  newcap : true, nomen : true, plusplus : true,
  regexp : true, sloppy : true, vars : true,
  white : true
*/
/*global jQuery */

// Модуль /spa/
// Обеспечивает функциональность всплывающего чата
//
var spa = (function ( $ ) {
  // Переменные в области видимости модуля
  var
    // Задать константы
    configMap = {
      extended_height : 434,
      extended_title : 'Click to retract',
      retracted_height : 16,
      retracted_title : 'Click to extend',
      template_html : '<div class="spa-slider"><\div>'
    },

    // Объявить все прочие переменные в области видимости модуля
    $chatSlider,
    toggleSlider, onClickSlider, initModule;

  // Метод DOM /toggleSlider/
  // изменяет высоту окна чата
  //
  toggleSlider = function () {
    var
      slider_height = $chatSlider.height();

    // раскрыть окно чата, если оно свернуто
    if ( slider_height === configMap.retracted_height ) {
      $chatSlider
        .animate({ height : configMap.extended_height })
        .attr( 'title', configMap.extended_title );
      return true;
    }

    // свернуть окно чата, если оно раскрыто
    else if ( slider_height === configMap.extended_height ) {
      $chatSlider
        .animate({ height : configMap.retracted_height })
        .attr( 'title', configMap.retracted_title );
      return true;
    }

    // ничего не делать, если окно чата в процессе перехода
    return false;
  }
}

```

Включаем параметры JSLint. Мы пользуемся JSLint для проверки отсутствия в нашем коде типичных ошибок. Сейчас не так важно, что эти параметры означают. Более подробно JSLint рассмотрена в приложении А.

Помещаем весь наш код в пространство имен spa. Дополнительные сведения об этом приеме приведены в главе 2.

Объявляем все переменные до использования. Сохраняем конфигурационные параметры модуля в configMap, а переменные состояния — в stateMap.

Помещаем все методы манипуляции DOM в одну секцию.

Этот код раскрывает окно чата. Он сравнивает текущую высоту окна с минимальной, чтобы понять, свернуто ли оно полностью. Если это так, то с помощью имеющегося в jQuery механизма анимации раскрывает его.

Этот код сворачивает окно чата. Он сравнивает текущую высоту окна с максимальной, чтобы понять, раскрыто ли оно полностью. Если это так, то с помощью имеющегося в jQuery механизма анимации сворачивает его.

```

// Обработчик события /onClickSlider/
// получает событие щелчка и вызывает toggleSlider
onClickSlider = function ( event ) {
    toggleSlider();
    return false;
};

// Открытый метод /initModule/
// устанавливает начальное состояние и предоставляет функциональность
//
initModule = function ( $container ) {

    // отрисовать HTML
    $container.html( configMap.template_html );

    $chatSlider = $container.find( '.spa-slider' );
    // инициализировать высоту и описание окна чата
    // привязать обработчик к событию щелчка мышью
    $chatSlider
        .attr( 'title', configMap.retracted_title )
        .click( onClickSlider );

    return true;
};

return { initModule : initModule };
}( jQuery ));

// запустить spa, когда модель DOM будет готова
//
jQuery(document).ready(
    function () { spa.initModule( jQuery( '#spa' ) ); }
);
</script>
</head>

<body>
    <div id="spa"></div>
</body>
</html>

```

Помещаем все методы обработки событий в одну секцию. Рекомендуется делать обработчики небольшими и целенаправленными. Для обновления экрана и реализации бизнес-логики они должны вызывать другие методы.

Помещаем все открытые методы в одну секцию.

Добавляем код, который помещает в \$container шаблон HTML-разметки окна чата.

Находим div выплывающего чата и сохраняем его в переменной \$chatSlider в области видимости модуля. Переменные в области видимости модуля доступны только функциям из пространства имен spa.

Устанавливаем заголовок выплывающего окна и привязываем обработчик onClickSlider к событию щелчка по окну чата.

Экспортируем открытые методы, возвращая объект из пространства имен spa. Мы экспортируем только один метод — initModule.

Запускаем SPA только после того, как модель DOM будет готова к работе, пользуясь для этого методом ready из библиотеки jQuery.

Очищаем HTML. Ответственность за отрисовку окна чата берет на себя наш JavaScript-код, поэтому в статической разметке его быть не должно.

Не переживайте по поводу проверки с помощью JSLint, мы вернемся к этому вопросу в последующих главах. Но есть несколько вещей, о которых уместно сказать уже сейчас. Во-первых, в комментариях в начале скрипта задаются параметры проверки. Во-вторых, этот скрипт проходит указанные проверки без ошибок и предупреждений. Наконец, JSLint настаивает, чтобы функции были объявлены до использования, поэтому скрипт написан «снизу вверх», то есть функции самого верхнего уровня находятся в конце.

Мы пользуемся библиотекой jQuery, потому что она предоставляет оптимизированные кросс-браузерные средства решения таких фундаментальных для JavaScript задач, как выборка элементов, обход и манипулирование моделью DOM, методы AJAX и обработка событий. Например, метод jQuery `$(selector).animate(...)` дает простой способ выполнить совсем нетривиальное действие: анимировать изменение высота окна чата с минимальной на максимальную (и наоборот) в течение заданного промежутка времени. Анимация начинается медленно, затем ускоряется, а потом снова замедляется до полной остановки. Чтобы реализовать подобный вид анимации – вдоль *переходной кривой*, необходимо знать о вычислении частоты кадров, о тригонометрических функциях и об особенностях различных браузеров. Если бы вы стали писать ее самостоятельно, то потребовалось бы несколько десятков дополнительных строк.

Функция `jQuery(document).ready(function)` также избавляет нас от трудоемкой работы. Она позволяет вызвать указанную функцию, после того как модель DOM будет готова к работе. Традиционно для решения этой задачи использовалось событие `window.onload`. По разным причинам это событие не очень эффективно для более требовательных SPA, хотя в данном случае разница невелика. Однако написать корректный код, который работал бы во всех браузерах, исключительно трудно, и получается он довольно длинным¹.

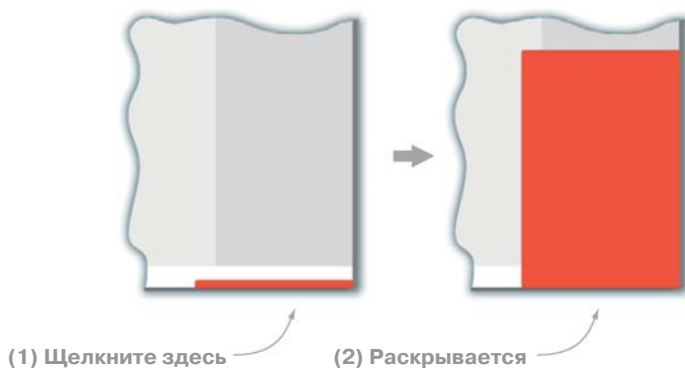


Рис. 1.6 ❖ Готовый всплывающий чат в действии – spa.html

¹ Чтобы оценить возникающие трудности, прочитайте статью по адресу www.javascriptkit.com/dhtmltutors/domready.shtml.

Как видно из рассмотренного примера, преимущества jQuery обычно намного перевешивают накладные расходы. В данном случае нам удалось сократить время разработки, уменьшить длину скрипта и обеспечить надежную кросс-браузерную совместимость. Плата за использование jQuery варьируется от низкой до пренебрежимо малой, потому что в минимизированном виде библиотека совсем небольшая, да к тому же, скорее всего, уже находится в кэше браузера. На рис. 1.6 показан готовый выплывающий чат. Закончив первую реализацию, посмотрим, как в действительности работает это приложение, воспользовавшись инструментами разработчика.

1.2.6. Изучение приложения с помощью инструментов разработчика в Chrome

Если вы хорошо знакомы с инструментами разработчика в Chrome, то можете этот раздел пропустить. В противном случае мы *настоятельно* рекомендуем поэкспериментировать с ними дома.

Откройте файл `spa.html` в Chrome. Как только он загрузится, откройте инструменты разработчика (**Меню** ⇒ **Инструменты** ⇒ **Инструменты разработчика**).

Первое, что бросается в глаза, – изменение DOM в результате включения нашим модулем элемента `<div class="spa-slider" ... >` (см. рис. 1.7). По мере развития нашего приложения появится еще *очень много* таких динамических элементов.



Рис. 1.7 ❖ Инспектирование элементов – `spa.html`

Мы можем исследовать, как выполняется JavaScript-код. Для этого нажмите кнопку **Sources** в верхнем меню на панели инструментов разработчика, а затем выберите содержащий JavaScript-код файл, как показано на рис. 1.8.

В последующих главах мы будем помещать JavaScript-код в отдельный файл. Но в этом примере код находится в HTML-файле, как

Выбор
исходного
файла



Рис. 1.8 ❖ Выбор исходного файла – spa.html

показано на рис. 1.9. Чтобы найти интересующий нас код, файл придется прокрутить вниз.

Исходный файл
в том виде,
в каком он
загружен
с сервера

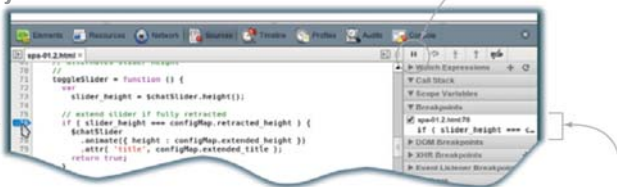


Рис. 1.9 ❖ Просмотр исходного файла – spa.html

Дойдя до строки 76, вы увидите предложение `if` (рис. 1.10). Нам хотелось бы посмотреть на код до выполнения этого предложения, поэтому щелкнем в левом поле, чтобы поставить точку прерывания. Как только интерпретатор JavaScript дойдет до этой строки скрипта, он приостановит выполнение программы, дав нам возможность проинспектировать элементы и переменные и разобраться, что происходит.

Приостановка/возобновление

Щелкаем по номеру
строки здесь...



... и выбранная строка появляется
в списке точек прерывания

Рис. 1.10 ❖ Установка точки прерывания – spa.html

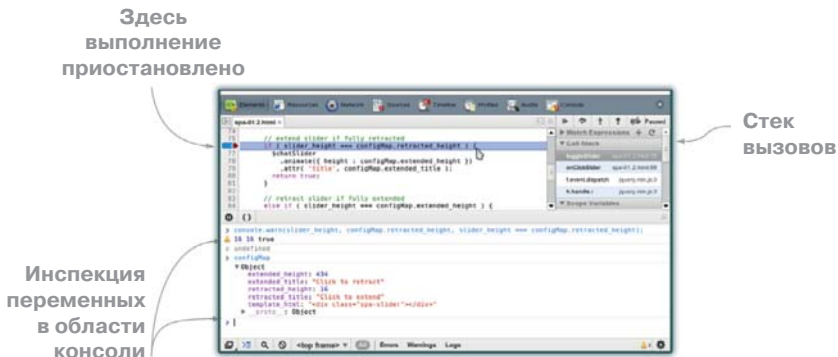


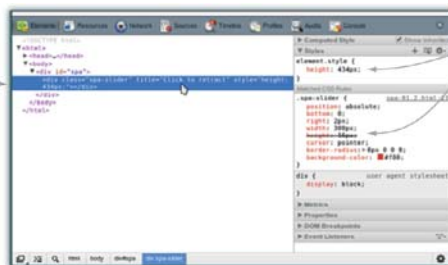
Рис. 1.11 ❖ Инспекция значений после прерывания – spa.html

Теперь вернемся в браузер и щелкнем по окну чата. Мы увидим, что скрипт остановился в строке 76, помеченной красной стрелкой (рис. 1.11). Пока приложение приостановлено, мы можем инспектировать переменные и элементы. Откройте консоль, введите имя какой-нибудь переменной и нажмите клавишу **Enter**, чтобы увидеть ее значение. Мы видим, что условие в предложении `if` истинно (`slider_height` равно 16 и `configMap.retracted_height` равно 16), и можем даже проинспектировать составные переменные, например объект `configMap`, который показан на рисунке в нижней части консоли. Закончив инспектирование, мы можем удалить точку прерывания, щелкнув в левом поле строки 76, а затем продолжить выполнение, нажав кнопку **Resume** в правом верхнем углу (над областью контрольных выражений `Watch Expressions`).

После нажатия **Resume** выполнение скрипта продолжится со строки 76, и смена состояния чата завершится. Вернемся на вкладку **Elements** и посмотрим, как изменилась модель DOM (рис. 1.12). По этому рисунку видно, что CSS-свойство `height`, которое раньше бралось из класса `spa-slider` (см. область подходящих правил `Matched CSS Rules` справа внизу), теперь замещено стилем элемента (стили, заданные непосредственно для элемента, приоритетнее, чем стили, определенные для класса или идентификатора). Снова щелкнув по окну чата, мы сможем в реальном времени наблюдать, как изменяется высота окна при сворачивании.

В этом кратком введении в инструменты разработчика в Chrome мы смогли показать лишь малую толику средств, помогающих понять, что происходит «под капотом» приложения, и внести изменения. По

Следите,
как изменяется
это значение
после щелчка
по окну чата



Стиль
элемента
всегда
замещает
стиль класса
или иденти-
фикатора

Рис. 1.12 ❖ Просмотр изменений в DOM – spa.html

ходу разработки приложения мы будем неоднократно пользоваться этими инструментами, поэтому рекомендуем потратить время на изучение руководства, размещенного по адресу <http://mng.bz/PzIJ>. Это время с лихвой окупится.

1.3. Чем хорошо написанное SPA удобно пользователям

Итак, мы построили свое первое одностраничное приложение, теперь обсудим основное преимущество SPA над традиционным веб-сайтом: существенные удобства, предоставляемые пользователям. SPA воплощает в себе лучшее из обоих миров: немедленную реакцию персонального приложения и переносимость и доступность сайта.

- *SPA отрисовывается как персональное приложение.* SPA перерисовывает лишь те части интерфейса, которые изменились, и лишь тогда, когда это необходимо. Напротив, традиционный сайт перерисовывает всю страницу в ответ на различные действия пользователя, что приводит к задержкам и «миганию», поскольку браузер должен получить страницу от сервера и нарисовать ее на экране. Если страница велика, сервер занят или соединение с Интернетом медленное, то задержка может составить несколько секунд, а пользователю остается лишь гадать, когда с ней снова можно будет работать. Это кошмар, по сравнению с быстрой отрисовкой и мгновенной обратной связью SPA.
- *SPA может реагировать как персональное приложение.* SPA минимизирует время реакции за счет того, что переносит рабочие (временные) данные и часть обработки с сервера в браузер.

В распоряжении SPA имеются данные и бизнес-логика, необходимые для принятия большинства решений локально, а значит, быстро. Лишь аутентификация пользователя, валидация и постоянное хранение данных должны остаться на сервере – по причинам, которые мы будем обсуждать в главах 6–8. В случае традиционного сайта большая часть логики приложения находится на сервере, поэтому, чтобы получить ответ на свои действия, пользователь должен дожидаться завершения цикла запрос–ответ–перерисовка. Это может занять несколько секунд, тогда как реакция SPA почти мгновенна.

- *SPA может уведомлять пользователя о своем состоянии, как и персональное приложение.* Если SPA все-таки должно ждать ответа сервера, то оно может динамически обновлять индикатор хода выполнения или занятости, чтобы пользователь не пугался задержки. При работе же с традиционным сайтом пользователь вынужден гадать, когда загрузка страницы закончится и с ней можно будет взаимодействовать.
- *SPA, как и сайт, доступно почти всегда.* В отличие от большинства персональных приложений, пользователь может обратиться к SPA, имея лишь соединение с Интернетом и пристойный браузер. В настоящее время все это есть на смартфонах, планшетах, телевизорах, ноутбуках и настольных ПК.
- *SPA, как и сайт, обновляется и распространяется мгновенно.* Пользователю вообще не нужно ничего делать, чтобы получить выгоду от этой возможности, – стоит перезагрузить браузер, и все работает. Никакой мороки с поддержанием актуальности многочисленных программ¹. Авторы работали со SPA, которые пересобирались и обновлялись несколько раз в день. Персональное приложение обычно необходимо скачать, а потом установить с правами администратора, а интервал между выпуском версий может составлять несколько месяцев или даже лет.
- *SPA, как и сайт, работает на разных платформах.* В отличие от большинства персональных приложений, хорошо написанное SPA может работать в любой операционной системе, где есть современный браузер с поддержкой HTML5. Обычно эта осо-

¹ Это не совсем верно: что делать, если изменяется формат обмена данными между клиентом и сервером, а у многих пользователей в браузере кэширована старая версия приложения? Но эту проблему можно решить, позаботившись заранее.

бенность считается преимуществом для разработчика, но она не менее важна многочисленным пользователям, работающим с несколькими устройствами, скажем, с Windows на работе, с Mac'ом дома, с сервером под управлением Linux, с телефоном Android и с планшетом Amazon.

Все сказанное означает, что следующее свое приложение вы, возможно, захотите сделать одностраничным. Все более требовательные пользователи постепенно начинают испытывать неприязнь к неуклюжим сайтам, которые перерисовывают страницу целиком после каждого щелчка. Хорошо написанное SPA с быстрым и информативным интерфейсом, доступное из любого места, где есть Интернет, поможет заинтересовать и удержать пользователей.

1.4. Резюме

Одностраничные приложения появились не вчера. До недавнего времени в качестве клиентской платформы для SPA чаще всего использовались Flash и Java, поскольку с точки зрения богатства функциональности, быстродействия и концептуальной целостности они превосходили JavaScript и механизмы отрисовки, встроенные в браузер. Но теперь в развитии JavaScript и браузеров наступил переломный момент – большинство досадных недостатков преодолено, а значительные преимущества, по сравнению с другими клиентскими платформами, остались.

Нас будет интересовать создание SPA с применением стандартного JavaScript и браузерных механизмов отрисовки. Говоря *SPA*, мы имеем в виду естественное JavaScript SPA, если не оговорено противное. Наш инструментарий для создания SPA включает jQuery, TaffyDB, Node.js, Socket.IO и MongoDB. Все это популярные, проверенные на практике решения. Впрочем, никто не мешает вам применять альтернативные технологии – принципиальная структура SPA не зависит от конкретных технических решений.

Разработанное нами простое приложение для показа выплывающего чата демонстрирует многие особенности написанного на JavaScript SPA. Оно мгновенно реагирует на действия пользователя и для принятия решений пользуется данными, которые хранятся на стороне клиента, а не сервера. Мы пользуемся программой JSLint, которая удостоверяет отсутствие типичных ошибок в JavaScript-коде приложения. А для выбора и анимации элемента DOM, равно как и для обработки щелчка мышью по окну чата, мы воспользовались библио-

текой jQuery. Мы изучили, как инструменты разработчика в Chrome позволяют разобраться в работе приложения.

SPA может предложить лучшее из обоих миров – мгновенную реакцию персонального приложения наряду с переносимостью и доступностью веб-сайта. JavaScript SPA доступно более чем миллиарду устройств, которые поддерживают современные браузеры и не нуждаются в сторонних подключаемых модулях. Приложив немного усилий, можно сделать так, что оно будет работать на настольных ПК, планшетах и смартфонах с разными операционными системами. SPA легко обновлять и распространять, обычно это не требует никаких действий со стороны пользователя. Все это объясняет, почему может возникнуть желание сделать следующее приложение одностраничным.

В следующей главе мы рассмотрим некоторые ключевые концепции JavaScript, которые необходимы для разработки SPA, но часто остаются недостаточно понятыми или невостребованными. На этом фундаменте мы затем улучшим и дополним одностраничное приложение, разработанное в данной главе.

Глава 2

Новое знакомство с JavaScript

В этой главе:

- ✧ Область видимости переменных, поднятие переменных и объект контекста выполнения.
- ✧ Что такое цепочка областей видимости переменной и как ей можно воспользоваться.
- ✧ Создание объектов в JavaScript с помощью прототипов.
- ✧ Самовыполняющиеся анонимные функции.
- ✧ Модули и закрытые переменные.
- ✧ Замыкания — сочетание приятного с полезным.

В этой главе мы дадим обзор уникальных особенностей JavaScript, которые необходимо понимать для написания сколько-нибудь серьезного естественного одностраничного приложения на JavaScript. В листинге 2.1 приведен фрагмент кода, разработанного в главе 1, иллюстрирующий рассматриваемые концепции. Если вы ясно понимаете, *как* и *почему* они используются, то можете вообще пропустить эту главу или, просмотрев ее по диагонали, перейти к главе 3, где мы начнем работать над SPA.

Прорабатывая материал дома, вы можете копировать код из листингов, приведенных в этой главе, на консоль, входящую в состав инструментов разработчика в Chrome, и, нажав **Enter**, смотреть, как он выполняется. Мы настоятельно рекомендуем не пренебрегать подобной возможностью.

Листинг 2.1 ✧ JavaScript-код приложения

```
...
var spa = (function ( $ ) { ← Самовыполняющиеся анонимные
    // Переменные в области видимости модуля      функции, паттерн модуля.
    var
        configMap = { ← Прототипическое наследование,
            extended_height : 434,                  понятие переменных, область
            extended_title  : 'Click to retract',    видимости переменной.
            retracted_height : 16,
            retracted_title  : 'Click to extend',
            template_html    : '<div class="spa-slider"></div>'
        }
```

```
    },
    $chatSlider,
    toggleSlider, onClickSlider, initModule;
...

// Открытый метод
initModule = function ( $container ) { ← Анонимные функции, паттерн модуля,
                                        замыкания.

    $container.html( configMap.template_html );
    $chatSlider = $container.find( '.spa-slider' );

    $chatSlider
        .attr( 'title', configMap.retracted_title )
        .click( onClickSlider );

    return true;
};

return { initModule : initModule }; ← Паттерн модуля, цепочка контекстов.

}( jQuery )); ← Самовыполняющиеся анонимные функции.

...
```

Стандарты кодирования и синтаксис JavaScript

Новичкам синтаксис JavaScript может показаться странным. Прежде чем идти дальше, важно понять, что такое блоки объявления переменных и литеральные объекты. Если вы уже с ними знакомы, можете пропустить эту врезку. Полный перечень важных, с нашей точки зрения, особенностей синтаксиса JavaScript и стандартов хорошего кодирования см. в приложении А.

Блоки объявления переменных

```
var spa = "Hello world!";
```

Переменные в JavaScript объявляются после ключевого слова `var`. Переменная может содержать данные любого типа: массивы, целые, с плавающей точкой и т. д. Тип переменной не указывается, поэтому JavaScript считается *слаботипизированным* языком. Даже после присваивания переменной значения ее тип может быть изменен в результате последующего присваивания значения другого типа, поэтому язык также считается *динамическим*.

В JavaScript объявления переменных и операторы присваивания можно сцеплять, поместив их после ключевого слова `var` и разделив запятыми:

```
var book, shopping_cart,
    spa = "Hello world!",
    purchase_book = true,
    tell_friends = true,
    give_5_star_rating_on_amazon = true,
    leave_mean_comment = false;
```

Существует несколько мнений о том, какой формат блока объявления переменных считать лучшим. Мы предпочитаем размещать в начале объявления неинициализированных переменных, а вслед за ними – объявления вместе с определениями. Мы также предпочитаем помещать запятые в конце строк, как показано выше, но не настаиваем на этом с пеной у рта, тем более что интерпретатору JavaScript это безразлично.

Литеральные объекты

Литеральный объект – это объект, определенный в виде списка атрибутов, перечисленных через запятую и заключенных в фигурные скобки. Значение атрибута отделяется от имени двоеточием, а не знаком равенства. Литеральные объекты могут также содержать массивы, представленные в виде списка элементов, перечисленных через запятую и заключенных в квадратные скобки. Для определения методов в качестве значения одного из атрибутов задается функция.

```
var spa = {
  title: "Single Page Web Applications", //атрибут
  authors: [ "Mike Mikowski", "Josh Powell" ], //массив
  buy_now: function () { //функция
    console.log( "Book is purchased" );
  }
}
```

Литеральные объекты и блоки объявления переменных используются в этой книге сплошь и рядом.

2.1. Область видимости переменной

Начать обсуждение удобно с вопроса о поведении переменных и о том, когда переменная находится или не находится в области видимости.

Областью видимости переменной в JavaScript является функция. Переменные могут быть глобальными или локальными. *Глобальная* переменная видна в любой точке программы, *локальная* – только там, где объявлена. Повторим еще раз: единственным блоком, определяющим область видимости переменной, в JavaScript является функция. Глобальные переменные определены вне любой функции, локальные – внутри функции. Просто, не правда ли?

По-другому эту ситуацию можно описать, уподобив функцию тюрьме, а определенные в ней переменные – заключенным. Как тюрьма является местом содержания заключенных и не позволяет им выходить за территорию, так и функция содержит переменные и не выпускает их наружу. Это иллюстрируется в следующем фрагменте кода.

```
var regular_joe = 'Я глобальная!';

function prison() {
```



```
var prisoner = 'Я локальная!';  
}  
  
prison();  
console.log( regular_joe );  
console.log( prisoner );
```

← Выводится «Я глобальная!».

← Выводится «Error: prisoner is not defined».

JavaScript 1.7, 1.8, 1.9+ и область видимости блока

В версии JavaScript 1.7 появился новый конструктор переменной с областью видимости блока – предложение `let`. К сожалению, хотя для версий JavaScript 1.7, 1.8, 1.9 существуют стандарты, даже версия 1.7 поддерживается в браузерах несогласованно. И до тех пор, пока браузеры не станут совместимыми в части обновления JavaScript, мы будем считать, что версии JavaScript, начиная с 1.7, не существуют. Тем не менее посмотрим, как это должно работать:

```
let (prisoner = 'Я в тюрьме!') {  
  console.log( prisoner );  
}  
console.log( prisoner );
```

← Выводится «Я в тюрьме!».

← Выводится «Error: prisoner isn't defined».

Чтобы использовать JavaScript 1.7, укажите номер версии в атрибуте `type` тега `script`:

```
<script type="application/javascript;version=1.7">
```

Это лишь мимолетное знакомство с JavaScript 1.7+; существует еще много изменений и дополнительных возможностей.

Ах, если бы все было так просто. Первое, на что натыкаешься, изучая правила видимости в JavaScript, – возможность объявить глобальную переменную внутри функции, просто опустив объявление `var`, как показано на рис. 2.1. А, как в любом языке программирования, глобальные переменные – почти всегда Плохая Идея.

```
function prison () {  
  prisoner_1 = 'Я сбежала!';  
  var prisoner_2 = 'Я заперта!';  
}  
  
prison();  
console.log( prisoner_1 );  
console.log( prisoner_2 );
```

← Выводится «Я сбежала!».

← Выводится «Error: prisoner_2 is not defined».

Ничего хорошего в этом нет – заключенные не должны сбегать. Еще одно место, где часто встречается этот подводный камень, – объявление счетчика цикла `for` без ключевого слова `var`. Рассмотрим следующие два определения функции `prison`:

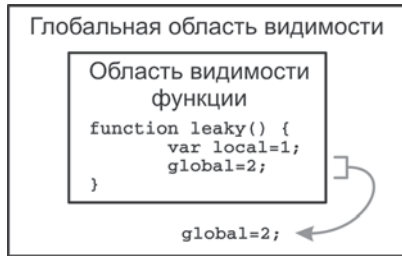


Рис. 2.1 ❖ Забыв поставить ключевое слово `var` при объявлении локальной переменной, вы создадите глобальную переменную

```
// неправильно
function prison () {
    for( i = 0; i < 10; i++ ) {
        //...
    }
}
prison();
console.log( i ); // i равно 10
delete window.i;

// допустимо
function prison () {
    for( var i = 0; i < 10; i++ ) {
        //...
    }
}
prison();
console.log( i ); // i не определена

// наилучшее решение
function prison () {
    var i;
    for ( i = 0; i < 10; i++ ) {
        // ...
    }
}
prison();
console.log( i ); // i не определена
```

Последняя версия нравится нам больше, потому что, объявляя переменную в начале функции, мы сразу понимаем, какова ее область видимости. Видя объявление переменной в инициализаторе цикла `for`, человек может ошибочно подумать, что областью ее видимости

является только сам цикл, как в некоторых других языках программирования.

Следуя этой логике, мы рекомендуем помещать все объявления и большую часть присваиваний в начало той функции, где они находятся, чтобы область видимости переменной не вызывала сомнений:

```
function prison() {
  var prisoner = 'Я локальная!',
      warden   = 'Я тоже локальная!',
      guards   = 'И я локальная!';
};
```

Перечисляя объявления локальных переменных через запятую в одном предложении `var`, мы делаем их хорошо видными и, что важнее, уменьшаем шанс по ошибке создать глобальную переменную вместо локальной. Также обратите внимание на аккуратное выравнивание строк и на то, что точка с запятой в конце воспринимается как естественное завершение блока объявления переменных. Об этом и других способах форматирования JavaScript-кода с целью повысить удобочитаемость и понятность мы будем говорить в приложении А «Стандарты кодирования на JavaScript». С методом объявления локальных переменных связана еще одна особенность JavaScript – поднятие переменных (variable hoisting). Рассмотрим ее.

```
function hoisted() {
  console.log(v);
  var v=1;
}

function hoisted() {
  var v;
  console.log(v);
  v=1;
}
```

Рис. 2.2 ❖ Объявления переменных в JavaScript «поднимаются» в начало объемлющей функции, но инициализация производится там, где переменная встретилась. Интерпретатор JavaScript на самом деле не переписывает код, объявление перемещается при каждом вызове функции

2.2. Поднятие переменных

Любое объявление переменной в JavaScript *поднимается* в начало области видимости объемлющей функции, при этом переменной присваивается значение `undefined`. Получается, что переменная, объявленная в любом месте функции, существует во всем коде этой функции, но остается неопределенной, пока ей не будет присвоено значение (рис. 2.2).

```
function prison () {
  console.log(prisoner); ← Выводится «prisoner is undefined».
  var prisoner = 'Теперь я определена!';

  console.log(prisoner); ← Выводится «Теперь я определена!».
}
prison();
```

Сравните код на этом рисунке с попыткой обратиться к переменной, не объявленной ни локально, ни глобально. Это приведет к ошибке времени выполнения, и интерпретатор JavaScript остановится на соответствующем предложении:

```
function prison () {
  console.log(prisoner); ← Выводится «Error: prisoner is not defined», и интерпретатор JavaScript прекращает выполнение программы.
}
prison();
```

Поскольку объявления переменных все равно поднимаются в начало области видимости функции, мы рекомендуем объявлять все переменные в начале функции, предпочтительно в одном предложении `var`. Это согласуется с поведением JavaScript и помогает избежать путаницы, проиллюстрированной на рисунке выше.

```
function prison () {
  console.log(prisoner); ← Выводится «undefined».
  var prisoner, warden, guards;

  console.log(prisoner); ← Выводится «undefined».
  prisoner = 'prisoner присвоено значение';

  console.log(prisoner); ← Выводится «prisoner присвоено значение».
}
prison();
```

В сочетании области видимости и механизм поднятия переменных иногда приводят к неожиданному поведению. Взгляните на следующий код:

```
var regular_joe = 'Regular Joe';
function prison () {
  console.log(regular_joe);
}
prison();
```

regular_joe определена в глобальной области видимости.

Глобальная переменная regular_joe печатается внутри функции prison, выводится 'Regular Joe'.

Когда внутри функции `prison` значение переменной `regular_joe` передается функции `console.log()`, интерпретатор JavaScript сначала

проверяет, объявлена ли `regular_joe` в локальной области видимости. Поскольку это не так, интерпретатор далее просматривает глобальную область видимости, обнаруживает, что там переменная определена, и возвращает ее значение. Это называется *проходом по цепочке областей видимости*. Но что, если переменная объявлена также в локальной области видимости?

```
var regular_joe = 'regular_joe присвоено значение';
function prison () {
  console.log(regular_joe);
  var regular_joe;
}
prison();
```

Выводится «undefined». Объявление `regular_joe` поднимается в начало функции, и это поднятое объявление проверяется еще до поиска `regular_joe` в глобальной области видимости.

Противоречит интуиции? Странно? Что ж, пройдем вслед за JavaScript по всему пути поднятия объявлений.

2.3. Еще о поднятии переменных и объекте контекста выполнения

Считается, что любой разработчик на JavaScript, если хочет быть успешным, обязан понимать рассмотренные до сих пор концепции. Но сделаем еще один шаг и заглянем под капот: вы станете одним из немногих, кто понимает, как на самом деле работает JavaScript. Начнем с одной из самых «магических» особенностей JavaScript: поднятия переменных и функций.

2.3.1. Поднятие

Как всегда бывает с магией, фокус вызывает чуть ли не разочарование, когда его секрет раскрыт. А секрет в том, что интерпретатор JavaScript, входя в область видимости, делает два прохода по коду. На первом проходе инициализируются переменные, а на втором выполняется код. Просто, я знаю – и не понимаю, почему это обычно объясняют другими словами. Но посмотрим внимательнее, что интерпретатор делает на первом проходе, потому что это имеет некоторые любопытные последствия.

На первом проходе интерпретатор JavaScript просматривает код и делает три вещи:

1. Объявляет и инициализирует аргументы функций.
2. Объявляет локальные переменные, в том числе анонимные функции, присвоенные локальной переменной, но не инициализирует их.

3. Объявляет и инициализирует функции.

Листинг 2.2 ❖ Первый проход

```
function myFunction( arg1, arg2 ) {
  var local_var = 'foo';
  a_function = function () {
    console.log( 'a function' );
  };

  function inner () {
    console.log('inner');
  }
}
myFunction( 1,2 );
```

1 Объявляет и инициализирует аргументы функций.

2 Объявляет локальные переменные, в том числе анонимные функции, присвоенные локальной переменной, но не инициализирует их.

3 Объявляет и инициализирует функции.

На первом проходе локальным переменным *не* присваиваются значения, потому что значение может вычисляться в коде, а код на первом проходе не исполняется. Аргументам же значения присваиваются, потому что код, в котором они вычисляются, уже выполнен до передачи аргумента функции.

Убедиться в том, что значения аргументов присваиваются на первом проходе, можно, проведя сравнение с кодом из предыдущего раздела, где демонстрировалось поднятие переменных.

Листинг 2.3 ❖ Переменные не определены до момента объявления

```
var regular_joe = 'regular_joe присвоено значение';
function prison () {
  console.log(regular_joe);
  var regular_joe;
}
prison();
```

Выводится «undefined». Объявление `regular_joe` поднимается в начало функции, и это поднятое объявление проверяется еще до поиска `regular_joe` в глобальной области видимости.

Переменная `regular_joe` равна `undefined`, потому что она объявлена в функции `prison`, но если `regular_joe` также передается в качестве аргумента, то еще до объявления она имеет значение.

Листинг 2.4 ❖ Переменные имеют значение до момента объявления

```
var regular_joe = 'regular_joe присвоено значение';
function prison ( regular_joe ) {
  console.log(regular_joe);
  var regular_joe;

  console.log(regular_joe);
}
prison( 'аргумент regular_joe' );
```

Выводится «аргумент regular_joe». Аргументам присваиваются значения на первом проходе. Если не знать о двух проходах интерпретатора JavaScript, то кажется, что аргумент `regular_joe` должен быть перезаписан в результате поднятия объявления локальной переменной `regular_joe`.

Выводится «аргумент regular_joe». Сюрприз! Поскольку `regular_joe` было присвоено значение как аргументу, то при объявлении локальной переменной с тем же именем оно не перезаписывается. Это объявление излишне.

Если у вас от всего этого голова идет кругом, то это нормально. Мы объяснили, что интерпретатор JavaScript делает два прохода по

исполняемой функции и что на первом проходе он сохраняет переменные, но мы еще не видели, *как* переменные хранятся. Знание о том, как интерпретатор хранит переменные, наверное, устранил все оставшиеся недоразумения. Переменные хранятся как атрибуты так называемого *объекта контекста выполнения*.

2.3.2. Контекст выполнения и объект контекста выполнения

С каждым вызовом функции связывается новый контекст выполнения. Контекст выполнения – это концепция, а не объект, точнее концепция выполняемой функции. Можно провести аналогию с атлетом в беговом или прыжковом контексте. Мы могли бы сказать «бегущий атлет», а не «атлет в беговом контексте», и точно так же можно было бы говорить о выполняемой функции, но технический жаргон устроен иначе, и мы говорим *контекст выполнения*.

Контекст выполнения знает обо всем, что происходит во время выполнения функции. Он отличается от объявления функции, потому что в объявлении говорится о том, что *будет* происходить во время выполнения функции, а контекст – это и *есть* ее выполнение.

Все переменные и функции, определенные внутри функции, считаются частью контекста ее выполнения. Контекст выполнения – это часть того, что имеют в виду программисты, говоря об *области видимости* функции (scope). Говорят, что переменная «находится в области видимости», если она доступна в текущем контексте выполнения. По-другому то же самое можно выразить, сказав, что переменная доступна, когда функция выполняется.

Переменные и функции, являющиеся частью контекста выполнения, хранятся в *объекте контекста выполнения* – реализации контекста выполнения в стандарте ЕСМА. Объект контекста выполнения – это объект внутри интерпретатора JavaScript, а не переменная, доступная непосредственно написанной на JavaScript программе. Но к ней несложно получить косвенный доступ, потому что всякий раз, используя какую-то переменную, вы обращаетесь к атрибуту объекта контекста выполнения.

Выше мы сказали, что интерпретатор JavaScript делает два прохода по контексту выполнения – для объявления и инициализации переменных. Но где эти переменные хранятся? А вот как раз в объекте контекста выполнения – в виде его атрибутов. Пример хранения переменных приведен в табл. 2.1.

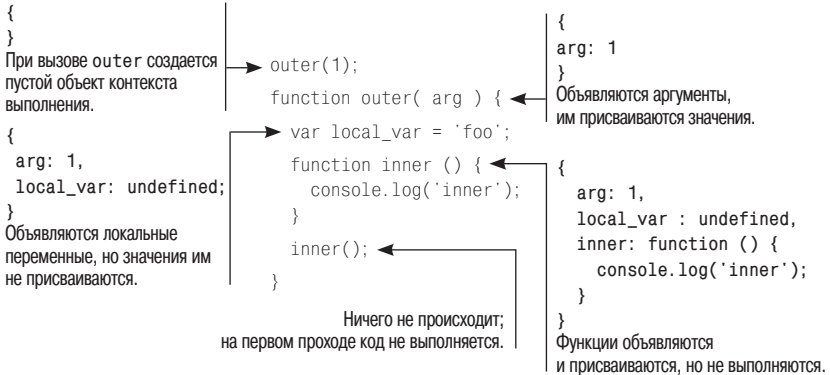
Таблица 2.1. Объект контекста выполнения

Код	Объект контекста выполнения
<pre>var example_variable = "example", another_example = "another";</pre>	<pre>{ example_variable: "example", another_example: "another" };</pre>

Возможно, вы никогда не слышали об объекте контекста выполнения. Эту тему нечасто обсуждают в сообществе веб-разработчиков, наверное, потому что объект контекста выполнения скрыт глубоко в недрах реализации JavaScript и напрямую программисту недоступен.

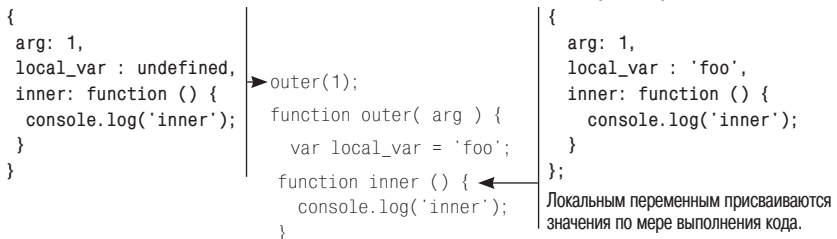
Без понимания того, что такое объект контекста выполнения, не понять материала этой главы, поэтому пройдем по всему жизненному циклу этого объекта и создающего его JavaScript-кода.

Листинг 2.5 ❖ Объект контекста выполнения – первый проход



Теперь, когда аргументы и функции объявлены и инициализированы, а локальные переменные только объявлены, начинается второй проход, на котором выполняется JavaScript-код, а локальным переменным присваиваются значения.

Листинг 2.6 ❖ Объект контекста выполнения – второй проход



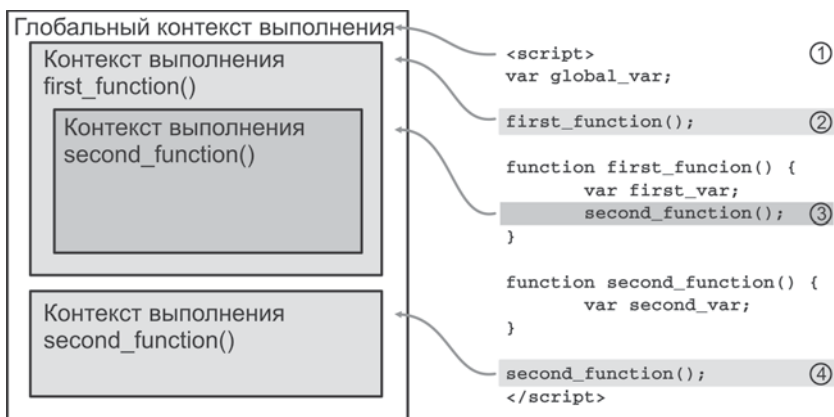


Рис. 2.3 ❖ Вызов функции создает контекст выполнения

```

inner(); ← {
}
    {
    arg: 1,
    local_var : 'foo',
    inner: function () {
        console.log('inner');
    }
    }
}

```

Атрибуты, представляющие переменные в этом объекте контекста выполнения, остаются такими же, но при вызове функции `inner` внутри него создается новый объект контекста выполнения.

Количество уровней вложенности может быть гораздо больше, так как каждый вызов функции внутри контекста выполнения создает новый контекст выполнения, вложенный в текущий. Что, опять голова кругом? Тогда пора нарисовать картинку. См. рис. 2.3.

1. Все, что находится внутри тега `<script>`, образует глобальный контекст выполнения.
2. Вызов функции `first_function` создает новый контекст выполнения внутри глобального. Во время выполнения `first_function` имеет доступ к переменным в том контексте выполнения, в котором была вызвана. В данном случае `first_function` имеет доступ к переменным, определенным в глобальном контексте выполнения, и к локальным переменным, определенным внутри `first_function`. Говорят, что эти переменные находятся *в области видимости*.
3. При вызове функции `second_function` создается новый контекст выполнения внутри контекста выполнения `first_function`.

Функция `second_function` имеет доступ к переменным из контекста выполнения `first_function`, потому что она в нем и вызвана. Но `second_function` также имеет доступ к переменным из глобального контекста выполнения и к локальным переменным, определенным внутри `second_function`. Говорят, что эти переменные находятся *в области видимости*.

4. Функция `second_function` вызывается снова, на этот раз в глобальном контексте выполнения. Теперь она не имеет доступа к переменным в контексте выполнения `first_function`, потому что вызывалась не в этом контексте. Проще говоря, при втором вызове `second_function` не имеет доступа к переменным, определенным внутри `first_function`, потому что не вызывалась из `first_function`.

Этот контекст выполнения `second_function` также не имеет доступа к переменным, созданным при предыдущем вызове `second_function`, потому что эти вызовы произведены в разных контекстах выполнения. Иначе говоря, при вызове функции мы не имеем доступа к локальным переменным, созданным во время ее предыдущего вызова. Говорят, что эти недоступные переменные находятся *вне области видимости*.

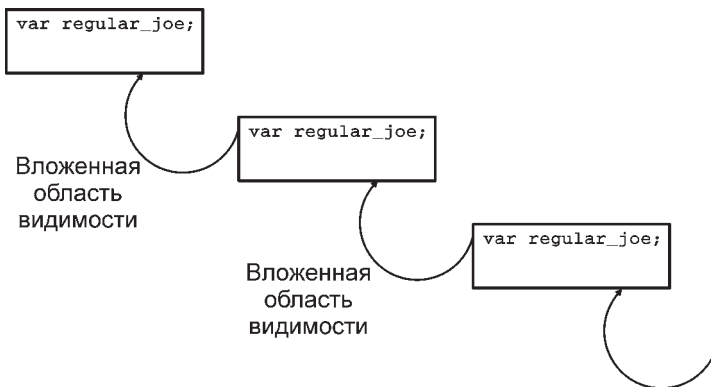


Рис. 2.4 ❖ Во время выполнения интерпретатор JavaScript просматривает иерархию областей видимости, пытаясь разрешить имена переменных

Порядок, в котором интерпретатор JavaScript просматривает объекты контекста выполнения при доступе к переменным, находящимся «в области видимости», называется *цепочкой областей видимости*.

Вместе с *цепочкой прототипов* эта цепочка определяет порядок, в котором JavaScript находит переменные и их атрибуты. Эти концепции мы обсудим в следующих разделах.

2.4. Цепочка областей видимости

До сих пор при обсуждении областей видимости переменных мы ограничивались *глобальными* и *локальными* переменными. Для начала это неплохо, но понятие области видимости не такое плоское, как следует из обсуждения вложенных контекстов выполнения в предыдущем разделе. Область видимости переменной правильнее представлять себе как цепочку, показанную на рис. 2.4. При поиске определения переменной интерпретатор JavaScript сначала просматривает локальный объект контекста выполнения. Если в нем определение не найдено, то интерпретатор переходит вверх по цепочке областей видимости к контексту выполнения, в котором был создан текущий контекст, и ищет определение переменной там. Так происходит до тех пор, пока определение не будет найдено или не будет достигнута глобальная область видимости.

Модифицируем предыдущий пример, чтобы проиллюстрировать концепцию цепочки областей видимости. Код в листинге 2.7 напечатает такие сообщения:

```
Я здесь, чтобы спасти положение!
regular_joe присвоено значение
undefined
```

Листинг 2.7 ❖ Пример цепочки областей видимости – переменная `regular_joe` определена в каждой области видимости

```
var regular_joe = 'Я здесь, чтобы спасти положение!';
// печатается 'Я здесь, чтобы спасти положение!'
console.log(regular_joe);
function supermax(){
    var regular_joe = 'regular_joe присвоено значение';
    // печатается 'regular_joe присвоено значение'
    console.log(regular_joe);
    function prison () {
        var regular_joe;
        console.log(regular_joe);
    }
    // печатается 'undefined'
```

regular_joe присвоено значение
в глобальном контексте.

Вызывающая область видимости: глобальная. Ближайшее соответствие в цепочке областей видимости: глобальная переменная regular_joe.

Вызывающая область видимости: глобальная -> supermax(). Ближайшее соответствие в цепочке областей видимости: переменная regular_joe, определенная внутри supermax().

Вызывающая область видимости: глобальная -> supermax() -> prison(). Ближайшее соответствие в цепочке областей видимости: переменная regular_joe, определенная внутри prison().

```
    prison();  
}  
supermax();
```

Во время выполнения интерпретатор JavaScript просматривает цепочку областей видимости, пытаясь разрешить имена переменных. Просмотр начинается с текущей области видимости, а затем продолжается вверх по цепочке до области видимости верхнего уровня, каковой является объект `window` (в браузерах) или `global` (в Node.js). Как только будет найдено первое соответствие, поиск прекращается. Это означает, что переменные в более глубоко вложенных областях видимости могут скрывать те, что определены в областях, более близких к глобальной, так как обнаруживаются раньше. Хорошо это или плохо, зависит от того, ожидаете вы такого поведения или нет. В реальной программе следует стремиться к тому, чтобы имена переменных были по возможности уникальны; рассматриваемый код, в котором одно и то же имя вводится в трех вложенных областях видимости, вряд ли можно считать удачным, он приведен лишь для иллюстрации идеи.

В листинге выше значение переменной `regular_joe` запрашивается в трех областях видимости.

1. В последней строке главной программы вызов `console.log(regular_joe)` производится в глобальной области видимости. Интерпретатор JavaScript начинает с поиска свойства `regular_joe` в объекте глобального контекста выполнения. Такое свойство там есть, оно имеет значение `Я здесь, чтобы спасти положение!`, которое и используется.
2. В последней строке функции `supermax` также находится вызов `console.log(regular_joe)`. Он производится в контексте выполнения `supermax`. Интерпретатор JavaScript начинает с поиска свойства `regular_joe` в объекте контекста выполнения `supermax`. Такое свойство там есть, оно имеет значение `regular_joe` присвоено значение, которое и используется.
3. И в последней строке функции `prison` мы также видим вызов `console.log(regular_joe)`. Он производится в контексте выполнения `prison`. Интерпретатор JavaScript начинает с поиска свойства `regular_joe` в объекте контекста выполнения `prison`. Такое свойство там есть, оно имеет значение `undefined`, которое и используется.

В данном примере переменная `regular_joe` определена во всех трех областях видимости. В следующей версии (листинг 2.8) мы определим

эту переменную только в глобальной области видимости. Теперь программа три раза печатает строку «Я здесь, чтобы спасти положение!».

Листинг 2.8 ❖ Пример цепочки областей видимости – переменная `regular_joe` определена только в одной области видимости

```
var regular_joe = 'I am here to save the day!';
// печатается 'Я здесь, чтобы спасти положение!'
console.log(regular_joe);
function supermax(){

    // печатается 'Я здесь, чтобы спасти положение!'
    console.log(regular_joe);

    function prison () {
        console.log(regular_joe);
    }

    // печатается 'Я здесь, чтобы спасти положение!'
    prison();
}
// печатается 'Я здесь, чтобы спасти положение!'. Дважды.
supermax();
```

regular_joe присвоено значение в глобальном контексте.

Вызывающая область видимости: глобальная. В ней переменная и будет найдена.

Вызывающая область видимости: глобальная -> supermax(). Ближайшее соответствие в цепочке областей видимости: глобальная переменная regular_joe.

Вызывающая область видимости: глобальная -> supermax() -> prison(). Ближайшее соответствие в цепочке областей видимости: глобальная переменная regular_joe.

Важно помнить, что запрошенная переменная может быть найдена в любом месте цепочки областей видимости. Контроль над тем, откуда берутся значения, лежит на нас, и если мы не будем этого понимать, то в коде воцарится мучительный хаос. Стандарты кодирования на JavaScript, приведенные в приложении А, рекомендуют ряд способов справиться с этой проблемой, и мы будем ими пользоваться по ходу дела.

Глобальные переменные и объект window

То, что мы привыкли называть *глобальными* переменными, – на самом деле свойства объекта верхнего уровня, представляющего среду выполнения. В браузере таковым является объект `window`; в Node.js – объект `global`, и область видимости переменных работает по-разному.

Объект `window` содержит много свойств, которые сами по себе содержат объекты, методы (`onload`, `onresize`, `alert`, `close`...), элементы DOM (`document`, `frames`...), а также другие переменные. Доступ ко всем этим свойствам осуществляется с помощью синтаксиса `window.property`.

```
window.onload = function(){
    window.alert('window loaded');
}
```

Объект верхнего уровня в Node.js называется `global`. Поскольку Node.js – веб-сервер, а не браузер, множество доступных функций и свойств существенно отличается.

Когда интерпретатор JavaScript, работающий в браузере, проверяет существование *глобальной* переменной, он просматривает объект `window`.

```
var regular_joe = 'Глобальная переменная';
console.log( regular_joe );           // 'Глобальная переменная'
console.log( window.regular_joe );    // 'Глобальная переменная'
console.log( regular_joe === window.regular_joe ); // true
```

В JavaScript имеется параллельная цепочке областей видимости концепция *цепочки прототипов*, смысл которой – определить, где объект ищет определения своих атрибутов. Поговорим теперь о прототипах и цепочке прототипов.

2.5. Объекты в JavaScript и цепочка прототипов

Понятие объекта в JavaScript основано на прототипах, тогда как в большинстве распространенных в настоящее время языков используется понятие класса объекта. В системе на основе классов мы определяем объект, указывая класс, по образцу которого он устроен. В системе на основе прототипов создается объект, который, по нашей задумке, должен служить образцом для всех объектов подобного типа, после чего мы сообщаем интерпретатору JavaScript, что хотели бы получить другие похожие объекты.

Не желая заходить в метафору слишком далеко, все же отметим, что если бы архитектура была системой, основанной на классах, то архитектор представил бы чертежи одного дома, и все дома строились бы по этим чертежам. А если бы она была основана на прототипах, то архитектор построил бы один дом и сказал, что все дома нужно строить так, как этот.

Возьмем предыдущий пример заключенного и посмотрим, что нужно сделать в обеих системах, чтобы создать одного заключенного со свойствами имя, номер заключенного, срок заключения в годах и количество лет до условно-досрочного освобождения.

Таблица 2.2. Создание простого объекта: сравнение класса и прототипа

На основе классов	На основе прототипов
<pre>public class Prisoner { public int sentence = 4; public int probation = 2; public string name = "Joe"; public int id = 1234; }</pre>	<pre>var prisoner = { sentence : 4, probation : 2, name : 'Joe', id : 1234 };</pre>
<pre>Prisoner prisoner = new Prisoner();</pre>	

Создать объект на основе прототипа проще и быстрее в случае, когда имеется всего один экземпляр объекта. В системе на основе классов необходимо сначала определить класс и конструктор класса, а только потом создать объект этого класса. Объект на основе прототипа определяется прямо «на месте».

Плюсы системы на основе прототипов проявляются особенно ярко, когда имеется один простой объект, но она способна поддерживать и более сложный случай, когда есть несколько объектов с похожими характеристиками. Возьмем пример с заключенными и разрешим программе изменять имя и номер заключенного, запретив трогать срок заключения и количество лет до УДО.

Как видно из табл. 2.3, в обоих случаях последовательность программирования похожа, так что если вы привыкли к классам, то для перехода на прототипы вроде бы не придется прикладывать особых усилий. Однако дьявол кроется в деталях, поэтому программист, который раньше писал на языке с классами и приступил к работе на JavaScript, не изучив предварительно подхода на основе прототипов, рискует споткнуться на, казалось бы, ровном месте, где не видно никаких сложностей. Пройдем показанную последовательность по шагам и посмотрим, чему на ней можно научиться.

Таблица 2.3. Несколько объектов: сравнение класса и прототипа

На основе классов	На основе прототипов
<pre> /* var 1 */ public class Prisoner { public int sentence = 4; public int probation = 2; public string name; public string id; /* var 2 */ public Prisoner(string name, string id) { this.name = name; this.id = id; } } /* var 3 */ Prisoner firstPrisoner = new Prisoner("Joe", "12A"); Prisoner secondPrisoner = new Prisoner("Sam", "2BC"); </pre>	<pre> // * var 1 * var proto = { sentence : 4, probation : 2 }; // * var 2 * var Prisoner = function(name, id){ this.name = name; this.id = id; }; // * var 3 * Prisoner.prototype = proto; // * var 4 * var firstPrisoner = new Prisoner('Joe', '12A'); var secondPrisoner = new Prisoner('Sam', '2BC'); </pre>
<ol style="list-style-type: none"> 1. Определяем класс. 2. Определяем конструктор класса. 3. Создаем объекты. 	<ol style="list-style-type: none"> 1. Определяем объект-прототип. 2. Определяем конструктор объекта. 3. Связываем конструктор с прототипом. 4. Создаем объекты.

В каждом случае сначала создается шаблон объектов. В языках на основе классов шаблон называется *классом*, а в языках на основе прототипов – *прототипом*, но в обоих случаях они играют роль образца, по которому создаются объекты.

Затем создается конструктор. В языках на основе классов конструктор определяется внутри класса, поэтому при создании объекта совершенно ясно, где находится соответствующий конструктор. В JavaScript конструктор объектов определяется вне прототипа, поэтому нужен дополнительный шаг – связывание одного с другим.

Наконец, создается объект.

Использование оператора `new` в JavaScript означает отход от принципов прототипирования, быть может, в попытке сделать язык более понятным разработчикам, знакомым с наследованием на основе классов. Увы, нам кажется, что это только наводит тень на плетень и делает то, что должно быть незнакомым (и, стало быть, требует изучения), якобы знакомым, в результате чего разработчик очертя голову кидается программировать, а потом часами ищет ошибку, вызванную тем, что он по незнанию принял JavaScript за систему, основанную на классах.

В качестве альтернативы оператору `new` был разработан метод `Object.create`, который придает созданию объектов в JavaScript характер, более свойственный прототипическому наследованию. В этой книге мы только этим методом и будем пользоваться. Продемонстрированное в табл. 2.3 создание заключенных можно реализовать с помощью метода `Object.create` следующим образом.

Листинг 2.9 ❖ Использование `Object.create` для создания объектов

```
var proto = {
  sentence : 4,
  probation : 2
};

var firstPrisoner = Object.create( proto );
firstPrisoner.name = 'Joe';
firstPrisoner.id = '12A';

var secondPrisoner = Object.create( proto );
secondPrisoner.name = 'Sam';
secondPrisoner.id = '2BC';
```

Метод `Object.create` принимает прототип в качестве аргумента и возвращает объект; таким образом, можно определить объект-прототип с общими атрибутами и методами и использовать его для создания других объектов с такими же свойствами. Существует и другой,

тоже широко распространенный, подход к использованию `Object.create` – написать с его помощью фабричную функцию, которая создает и возвращает объект. Все фабричные функции мы будем называть по образцу `make<object_name>`.

Листинг 2.10 ❖ Использование `Object.create` с фабричной функцией

```
var proto = {
  sentence : 4,
  probation : 2
};

var makePrisoner = function( name, id ) {
  var prisoner = Object.create( proto );
  prisoner.name = name;
  prisoner.id = id;

  return prisoner;
};

var firstPrisoner = makePrisoner( 'Joe', '12A' );
var secondPrisoner = makePrisoner( 'Sam', '2BC' );
```

makePrisoner – фабричная функция, она создает объекты prisoner.

Само создание объекта производится так же, как в листинге выше, просто оно обернуто фабричной функцией.

Теперь для создания новых заключенных достаточно вызвать функцию makePrisoner, передав ей имя и номер ЗК.

Хотя в JavaScript есть несколько способов создания объектов (это еще одна тема, часто обсуждаемая в среде разработчиков), обычно рекомендуется использовать метод `Object.create`. Мы предпочитаем его, потому что он ясно показывает, как устанавливается прототип. К сожалению, чаще всего для создания объектов все-таки применяется оператор `new`. Мы говорим «к сожалению», потому что он наводит разработчиков на совершенно ошибочную мысль о том, будто язык основан на классах, и затушевывает тонкости системы на основе прототипов.

Метод `Object.create` для старых браузеров

Метод `Object.create` работает в IE 9+, Firefox 4+, Safari 5+ и Chrome 5+. Для совместимости с более старыми браузерами (это о вас – IE 6, 7 и 8!) необходимо определить этот метод, если он не существует, и оставить встроенный вариант для браузеров, где он и так реализован.

// Кросс-браузерный метод для поддержки `Object.create()`

```
var objectCreate = function ( arg ){
  if ( ! arg ) { return {}; }
  function obj() {};
  obj.prototype = arg;
  return new obj;
};

Object.create = Object.create || objectCreate;
```

Итак, мы теперь знаем, как в JavaScript используются прототипы для создания объектов с одинаковыми свойствами. Далее рассмотрим цепочку прототипов и поговорим о том, как интерпретатор JavaScript ищет значения атрибутов объекта.

2.5.1. Цепочка прототипов

Атрибуты объекта в основанном на прототипах языке JavaScript реализованы и работают иначе, чем в языках на основе классов. Сходство достаточно велико, так что по большей части можно жить, не понимая всех деталей, но когда различия все же проявляются, за непонимание приходится расплачиваться горьким разочарованием и потерянным временем. Поэтому с цепочкой прототипов лучше познакомиться заранее, как и с различиями между прототипами и классами.

В JavaScript *цепочка прототипов* применяется для поиска значений свойств. Когда запрашивается свойство объекта, интерпретатор сначала ищет его непосредственно в самом объекте. Если там его нет, то интерпретатор ищет его в прототипе (который хранится в свойстве `__proto__` объекта).

Если свойства нет и в прототипе объекта, то интерпретатор проверяет прототип прототипа (ведь прототип – это обычный объект, поэтому у него тоже есть прототип). И так далее. Цепочка прототипов обрывается на универсальном объекте `Object`. Если, просмотрев всю цепочку, интерпретатор так и не смог найти запрошенное свойство, он возвращает `undefined`. Детали просмотра цепочки прототипов в JavaScript могут показаться запутанными, но в этой книге нам до-

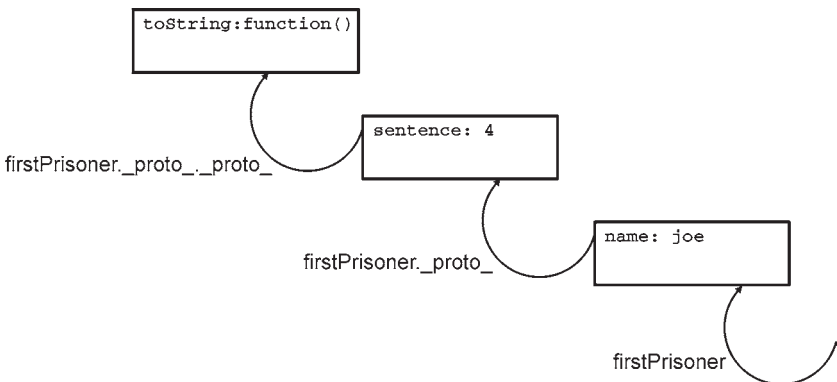


Рис. 2.5 ❖ Во время выполнения интерпретатор JavaScript просматривает цепочку прототипов в поисках значения свойства

статочно просто запомнить, что если свойство отсутствует у самого объекта, то проверяется его наличие в прототипе.

Описанный подъем по цепочке прототипов аналогичен подъему по цепочке областей видимости в поисках определения переменной. Легко видеть, что рис. 2.4 и 2.5, иллюстрирующие ту и другую концепцию, почти идентичны.

Обойти цепочку приоритетов можно и самостоятельно, пользуясь свойством `__proto__`.

```
var proto = {
  sentence : 4,
  probation : 2
};

var makePrisoner = function( name, id ) {

  var prisoner = Object.create( proto );
  prisoner.name = name;
  prisoner.id = id;

  return prisoner;
};

var firstPrisoner = makePrisoner( 'Joe', '12A' );

// Весь объект, включая свойства прототипа
// {"id": "12A", "name": "Joe", "probation": 2, "sentence": 4}
console.log( firstPrisoner );

// Только свойства прототипа
// {"probation": 2, "sentence": 4}
console.log( firstPrisoner.__proto__ );

// Прототип – это объект, у которого есть свой прототип. Если
// прототип явно не был задан, то таковым будет универсальный
// объект, представленный пустыми фигурными скобками
// {}
console.log( firstPrisoner.__proto__.__proto__ );

// Но у универсального объекта прототипа нет
// null
console.log( firstPrisoner.__proto__.__proto__.__proto__ );

// а попытка получить прототип null приводит к ошибке
// "firstPrisoner.__proto__.__proto__.__proto__ is null"
console.log( firstPrisoner.__proto__.__proto__.__proto__.__proto__ );
```

При запросе `firstPrisoner.name` JavaScript находит имя заключенного непосредственно в объекте и возвращает `Joe`. Свойство же

`firstPrisoner.sentence` у объекта отсутствует, поэтому JavaScript ищет его в прототипе и возвращает значение 4. А если мы вызовем метод `firstPrisoner.toString()`, то получим строку `[object Object]`, потому что такой метод имеется в базовом объекте-прототипе `Object`. Наконец, запросив свойство `firstPrisoner.hopeless`, мы получим `undefined`, потому что оно отсутствует во всей цепочке прототипов. Результаты сведены в табл. 2.4.

Таблица 2.4. Цепочка прототипов

Запрошенное свойство	Цепочка прототипов
<code>firstPrisoner</code>	<div><pre>{ id: '12A', name: 'Joe', __proto__: { probation: 2, sentence: 4, __proto__: { toString: function () {} } } }</pre></div> <div>← Объект <code>firstPrisoner</code>, созданный выше, его прототип и прототип прототипа, базовый объект JavaScript.</div>
<code>firstPrisoner.name</code>	<div><pre>{ id: '12A', name: 'Joe', __proto__: { probation: 2, sentence: 4, __proto__: { toString: function () {} } } }</pre></div> <div>← Свойство <code>name</code> читается непосредственно из объекта <code>firstPrisoner</code>.</div>
<code>firstPrisoner.sentence</code>	<div><pre>{ id: '12A', name: 'Joe', __proto__: { probation: 2, sentence: 4, __proto__: { toString: function () {} } } }</pre></div> <div>← Атрибут <code>sentence</code> отсутствует в самом объекте <code>firstPrisoner</code>, поэтому ищем в прототипе, где он имеется.</div>

Таблица 2.4 (окончание)

Запрошенное свойство	Цепочка прототипов
firstPrisoner.toString	<pre> { id: '12A', name: 'Joe', __proto__: { probation: 2, sentence: 4, __proto__: { toString: function () {} } } } </pre> <p>Метода toString() нет ни в объекте, ни в его прототипе, поэтому анализируем прототип прототипа, который оказывается базовым объектом JavaScript.</p> <p>← [машинный код]</p>
firstPrisoner.hopeless	<pre> { id: '12A', name: 'Joe', __proto__: { probation: 2, sentence: 4, __proto__: { toString: function () {} } } } </pre> <p>Атрибут hopeless не определен ни в объекте...</p> <p>← ... ни в прототипе...</p> <p>← ... ни в прототипе прототипа, поэтому имеет значение undefined.</p> <p>← [машинный код]</p>

Продемонстрировать цепочку прототипов можно и по-другому – посмотрев, что происходит, когда мы изменяем значение атрибута, заданное в прототипе.

Листинг 2.11 ❖ Замещение прототипа

```

var proto = {
  sentence : 4,
  probation : 2
};

var makePrisoner = function( name, id ) {

  var prisoner = Object.create( proto );
  prisoner.name = name;
  prisoner.id = id;

  return prisoner;
}

```

```

};

var firstPrisoner = makePrisoner( 'Joe', '12A' );

// В обоих случаях выводится 4
console.log( firstPrisoner.sentence );
console.log( firstPrisoner.__proto__.sentence );
firstPrisoner.sentence = 10;

// Выводится 10
console.log( firstPrisoner.sentence );

// Выводится 4
console.log( firstPrisoner.__proto__.sentence );
delete firstPrisoner.sentence;

// В обоих случаях выводится 4
console.log( firstPrisoner.sentence );
console.log( firstPrisoner.__proto__.sentence );

```

Атрибут `sentence` не найден в объекте `firstPrisoner`, поэтому при анализе `firstPrisoner.sentence` интерпретатор заглядывает в прототип объекта и там находит искомое свойство.

В свойство `sentence` объекта записывается значение 10.

Проверяем, что свойство действительно стало равно 10...

... но в прототипе этого объекта ничего не изменилось, там свойство по-прежнему равно 4.

Чтобы значение атрибута бралось из прототипа, как и раньше, удаляем атрибут из объекта.

Теперь интерпретатор JavaScript не находит атрибута в объекте, поэтому должен начать просмотр цепочки прототипов; искомый атрибут обнаруживается в объекте-прототипе.

Так что же все-таки происходит, когда мы изменяем значение атрибута объекта-прототипа? Интересно, да?

Мутация прототипа

У прототипического наследования есть одно мощное – и потенциально опасное – свойство: возможность изменить сразу *все* объекты, основанные на одном и тот же прототипе. Те из вас, кто знаком со статическими переменными, могут провести аналогию между заданными в прототипе атрибутами и статическими членами класса. Рассмотрим приведенный выше код еще раз.

```

var proto = {
  sentence : 4,
  probation : 2
};

var makePrisoner = function( name, id ) {
  var prisoner = Object.create( proto );
  prisoner.name = name;
  prisoner.id = id;

  return prisoner;
};

var firstPrisoner = makePrisoner( 'Joe', '12A' );

var secondPrisoner = makePrisoner( 'Sam', '2BC' );

```

Если сейчас проинспектировать объекты `firstPrisoner` и `secondPrisoner`, то мы увидим, что унаследованное свойство `sentence` равно 4.

...

```
// В обоих случаях выводится 4
console.log( firstPrisoner.sentence );
console.log( secondPrisoner.sentence );
```

Если изменить объект-прототип, например выполнив команду `proto.sentence = 5`, то во всех объектах, созданных по этому прототипу как позже, *так и раньше*, значение `sentence` изменится. То есть `firstPrisoner.sentence` и `secondPrisoner.sentence` станут равны 5.

...

```
proto.sentence = 5;
```

```
// В обоих случаях выводится 5
console.log( firstPrisoner.sentence );
console.log( secondPrisoner.sentence );
```

У такого поведения есть плюсы и минусы. Но важно, что оно имеет место во всех реализациях JavaScript, а зная об этом, мы можем соответственно писать код.

Теперь, зная, как объекты наследуют свойства других объектов с помощью прототипов, посмотрим, как работают функции, потому что их поведение тоже может отличаться от того, что вы ожидаете. Мы исследуем также некоторые проистекающие из этих отличий полезные возможности, которые будем применять в данной книге.

2.6. Функции – более пристальный взгляд

Функции в JavaScript – полноправные объекты. Их можно хранить в переменных, присваивать атрибутам и даже передавать другим функциям в качестве аргументов. Они применяются для управления областью видимости переменных и для реализации закрытых переменных и методов. Понимание функций – один из ключей к пониманию языка JavaScript и фундамент для построения профессиональных одностраничных приложений.

2.6.1. Функции и анонимные функции

Важная особенность функции в JavaScript – тот факт, что она является объектом. Кто из нас не встречал таких объявлений функций JavaScript:

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

Но функцию можно также сохранить в переменной:

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

Мы можем уменьшить избыточность (и шансы на рассогласование имен), создав *анонимную функцию*, которая является просто меткой, присвоенной функции, объявленной без имени. Вот пример анонимной функции, сохраненной в локальной переменной:

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

Функция, сохраненная в локальной переменной, вызывается точно так же, как обычная функция:

```
var prison = function () {
    console.log('вызвана prison');
};
prison(); ← Выводится «вызвана prison».
```

2.6.2. Самовыполняющиеся анонимные функции

Одна из проблем, с которой мы постоянно сталкиваемся в JavaScript, – тот факт, что все определенное в глобальной области видимости доступно из любого места программы. Иногда мы не хотим делиться со всеми и уж точно не хотим, чтобы сторонние библиотеки разделяли свои внутренние переменные с нами, потому что это прямой путь к вмешательству в чужие библиотеки и трудным для диагностики ошибкам. Используя свои знания о функциях, мы могли бы обернуть всю программу в функцию, а затем эту функцию вызвать, и тогда наши переменные были бы недоступны внешнему коду.

```
var myApplication = function () {
    var private_variable = "private";
};

myApplication();

// выводится сообщение об ошибке, говорящее, что переменная не определена
console.log( private_variable );
```

Но это многословный и неуклюжий способ. Насколько лучше и короче было бы обойтись без определения функции, сохранения ее в переменной и последующего выполнения. И знаете... это возможно!

```
(function () {
    var private_variable = "private";
})();
```



```
// выводится сообщение об ошибке, говорящее, что переменная не определена
console.log( private_variable );
```

Это называется *самовыполняющейся анонимной функцией*, потому что функция определена без имени и не сохраняется в переменной, а сразу выполняется. Чтобы выполнить такую функцию, нам нужно только окружить ее скобками и поставить еще пару скобок после нее (см. табл. 2.5). Если сопоставить с явно вызываемой функцией, то синтаксис не покажется таким уж удивительным.

Таблица 2.5. Сравнение явного вызова и самовыполняющейся функции. Результат один и тот же: функция создается и сразу вызывается

Явный вызов	Самовыполняющаяся функция
<pre>var foo = function () { // что-то сделать }; foo();</pre>	<pre>(function () { // что-то сделать })();</pre>

Самовыполняющиеся анонимные функции применяются для создания области видимости переменных, чтобы предотвратить просачивание имен в другие места программы. Таким образом можно создавать подключаемые модули на JavaScript, которые не будут конфликтовать с кодом приложения, потому что не вносят никаких переменных в глобальное пространство имен. В следующем разделе мы продемонстрируем еще более интересный способ применения, которым будем активно пользоваться в этой книге. Он называется *паттерном модуля* и позволяет определять закрытые переменные и методы. Но сначала посмотрим, как работает область видимости переменных в самовыполняющейся анонимной функции. Если это покажется вам знакомым, не удивляйтесь: идея-то та же, что и раньше, только синтаксис другой.

```
// сообщение об ошибке "local_var is not defined"
console.log(local_var);
```

← Локальная переменная, объявленная внутри функции, недоступна вне этой функции.

```
(function () {
    // local_var не определена
    console.log(local_var);
```

← Внутри самовыполняющейся анонимной функции и до объявления переменная равна undefined, потому что она объявлена на первом проходе интерпретатора JavaScript по функции и будет инициализирована, только когда интерпретатор увидит объявление на втором проходе.

```
var local_var = 'Локальная переменная!';
// local_var равна 'Локальная переменная!'
console.log(local_var);
```

← После того как переменная объявлена и инициализирована внутри функции, ее значение становится доступным.

```
}());
```

```
// сообщение об ошибке "local_var is not defined"
console.log(local_var);
```

Вне самовыполняющейся анонимной функции переменная не определена.

Сравните с таким кодом:

```
console.log(global_var);
var global_var = 'Глобальная переменная!';
console.log(global_var);
```

global_var равна undefined, но все-таки объявлена.

global_var равна «Глобальная переменная!»

Здесь глобальное пространство имен засорено переменной `global_var`, и есть риск конфликта с другими одноименными переменными в нашей программе или используемой нами внешней библиотеке. Возможно, вы встречали выражение «засорение глобального пространства имен» в разговорах на тему JavaScript – теперь знаете, что оно означает.

Самовыполняющиеся анонимные функции позволяют решить проблему перезаписи глобальных переменных сторонней библиотекой или даже – случайно – вашим собственным кодом. Передав такой функции некоторое значение в качестве параметра, вы можете быть уверены, что это значение останется таким, как вы ожидаете, в течение всего времени жизни данного контекста выполнения, потому что внешний код воздействовать на него не может.

Сначала посмотрим, как передать параметр самовыполняющейся анонимной функции.

```
(function (what_to_eat) {
    var sentence = 'Я собираюсь съесть ' + what_to_eat;
    console.log(sentence);
})('бутерброд');
```

Выводится «Я собираюсь съесть бутерброд».

Значение бутерброд передано самовыполняющейся анонимной функции в качестве параметра `what_to_eat`.

Возможно, этот синтаксис приводит вас в замешательство, но, по существу, мы просто передаем значение бутерброд анонимной функции в качестве первого параметра. Сравните с вызовом обычной функции:

```
var eatFunction = function (what_to_eat) {
    var sentence='Я собираюсь съесть ' + what_to_eat;
    console.log( sentence );
};
eatFunction( 'бутерброд' );
```

```
// то же самое, что
```

```
(function (what_to_eat) {
```

```
var sentence = 'Я собираюсь съесть ' + what_to_eat;
console.log(sentence);
})('бутерброд');
```

Единственная разница состоит в том, что мы убрали переменную `eatFunction` и окружили определение функции скобками.

Известный пример предотвращения перезаписи переменной дают JavaScript-библиотеки `jQuery` и `Prototype`. В обеих широко используется переменная `$`. Если вы включите в приложение обе библиотеки, то контроль над `$` получит та, что загружена последней. Технику передачи переменной самовыполняющейся анонимной функции можно использовать, для того чтобы гарантировать, что `jQuery` сможет использовать переменную `$` в некотором блоке кода.

Разбирая этот пример, следует знать, что переменные `jQuery` и `$` – синонимы. Передавая переменную `jQuery` самовыполняющейся анонимной функции, которая обращается к ней как к параметру `$`, мы предотвращаем перехват переменной `$` библиотекой `Prototype`:

```
( function ( $ ) { ← До этого момента $ – имя из библиотеки Prototype.
  console.log( $ ); ← Внутри области видимости функции $ – объект jQuery. Это простой пример:
})( jQuery );       даже функции, определенные внутри самовыполняющейся анонимной функции,
                    будут ссылаться на объект jQuery по имени $.
```

2.6.3. Паттерн модуля – привнесение в JavaScript закрытых переменных

Отлично, мы можем обернуть приложение самовыполняющейся анонимной функцией и тем самым защитить его от сторонних библиотек (а их – от нас), но одностраничное приложение – очень большая программа, которую не поместишь в один файл. Было бы неплохо как-нибудь разбить файл на модули, в каждом из которых есть собственные закрытые переменные. Ну, вы понимаете, к чему я веду, – это можно сделать!

Посмотрим, как разбить файл на несколько модулей, не отказываясь при этом от управления областью видимости переменных с помощью самовыполняющихся анонимных функций.

Все еще не привыкли к синтаксису самовыполняющихся анонимных функций?

Давайте взглянем еще раз. Вот этот странный синтаксис:

```
var prison = (function() {
  return 'Майк в тюрьме';
})();
```

Он практически ничем не отличается от такого:

```
function makePrison() {
    return 'Майк в тюрьме';
}
var prison = makePrison();
```

В обоих случаях значением переменной `prison` будет строка «Майк в тюрьме». Единственное практическое различие состоит в том, что мы нигде не сохраняем функцию `makePrison`, которая вызывается всего один раз, а создаем и сразу вызываем.

```
var prison = (function () {
    var prisoner_name = 'Майк Миковски',
        jail_term = 'срок 20 лет';
    return {
        prisoner: prisoner_name + ' - ' + jail_term,
        sentence: jail_term
    };
})();
```

Значение, возвращенное самовыполняющейся анонимной функцией, сохраняется в переменной `prison`.

Самовыполняющаяся анонимная функция возвращает объект с теми атрибутами, которыми хотим снабдить переменную `prison`.

// выводится undefined, переменная `prisoner_name` не видна.

```
console.log( prison.prisoner_name );
```

Выражение `prison.prisoner_name` равно `undefined`, потому что это не атрибут объекта, возвращенного самовыполняющейся анонимной функцией.

```
// выводится 'Майк Миковски - срок 20 лет'
console.log( prison.prisoner );
```

```
// выводится 'срок 20 лет'
console.log( prison.sentence );
```

Наша самовыполняющаяся анонимная функция выполняется немедленно и возвращает объект со свойствами `prisoner` и `sentence`. Сама анонимная функция не сохраняется в переменной `prison`, она уже выполнена, а в переменной `prison` сохранено *возвращенное ей значение*.

В глобальную область видимости мы помещаем не две переменные, `prisoner_name` и `jail_term`, а только одну, `prison`. Чем больше модуль, тем значительнее уменьшение количества глобальных переменных.

Проблема такого объекта – в том, что переменные, определенные в самовыполняющейся анонимной функции, пропадают, как только выполнение функции завершится, поэтому их невозможно изменить. `prisoner_name` и `jail_term` – не свойства объекта, сохраненного в переменной `prison`, поэтому через эту переменную до них не добраться. На самом деле это переменные, которые мы использовали для определения атрибутов `prisoner` и `sentence` объекта, возвращенного анонимной функцией; вот к этим-то атрибутам получить доступ через `prison` можно.

```
...
// выводится undefined
```

```

console.log( prison.jail_term );
prison.jail_term = 'Срок наказания сокращен';
// теперь выводится 'Срок наказания сокращен', но ...
console.log( prison.jail_term );
// здесь выводится 'Майк Миковски - срок 20 лет'... извини, Майк
console.log( prison.prisoner );

```

Выражение `prison.jail_term` равно `undefined`, потому что это не атрибут объекта, возвращенного самовыполняющейся анонимной функцией.

`prison` — объект, поэтому в нем можно определить атрибут `jail_term`...

... но атрибут `prison.prisoner` при этом не изменится.

Атрибут `prison.prisoner` не изменяется по нескольким причинам. Во-первых, `jail_term` — вообще не атрибут объекта `prison` или его прототипа; это была переменная в том контексте выполнения, где этот объект был создан и сохранен в переменной `prison`, а этот контекст больше не существует, потому что выполнение функции уже закончилось. Во-вторых, атрибуты устанавливаются один раз, когда анонимная функция выполняется, и больше никогда не изменяются. Чтобы их можно было изменить, мы должны преобразовать атрибуты в методы, которые обращаются к переменным.

```

var prison = (function () {
    var prisoner_name = 'Mike Mikowski',
        jail_term = 'срок 20 лет';

    return {
        prisoner: function () {
            return prisoner_name + ' - ' + jail_term;
        },
        setJailTerm: function ( term ) {
            jail_term = term;
        }
    };
})();

```

При каждом вызове функция `prisoner()` читает переменные `prisoner_name` и `jail_term`.

Возвращается объект с двумя методами.

При каждом вызове функция `setJailTerm()` находит и изменяет переменную `jail_term`.

```

// выводится 'Майк Миковски - срок 20 лет'
console.log( prison.prisoner() );

prison.setJailTerm( 'Срок наказания сокращен' );

// теперь здесь выводится 'Срок наказания сокращен'
console.log( prison.prisoner() );

```

Хотя самовыполняющаяся функция завершилась, переменные `prisoner_name` и `jail_term` остаются доступными методам `prisoner` и `setJailTerm`. Теперь они выступают в роли закрытых атрибутов объекта `prison`. Обратиться к ним можно только с помощью методов объ-

екта, возвращенного анонимной функцией, а непосредственно через объект или его прототип они недоступны. А вам доводилось слышать, что замыкания – это сложно? Стоп-стоп-стоп, прошу прощения... Я же еще не объяснил, что такое замыкание. Ну что же, вернемся немного назад.

Что такое замыкание?

Если рассматривать замыкания как абстрактную концепцию, то можно мозги свернуть, поэтому, перед тем как отвечать на вопрос «Что такое замыкание?», необходимо подготовить почву. Немного потерпите – и в конце раздела получите ответ на поставленный вопрос. Во время работы программа выделяет и использует память для самых разных вещей, в том числе для хранения значений переменных. Если бы программы никогда не освобождали ставшую ненужной память, то компьютер в конце концов «грохнулся» бы. В некоторых языках, например С, управление памятью возлагается на программиста, который должен писать программу так, чтобы она гарантированно освобождала выделенную память и как можно быстрее, – а это, вообще говоря, нелегко.

В других языках, например Java и JavaScript, реализована система автоматического освобождения ставшей ненужной памяти. Такие системы называются сборщиками мусора, быть может, потому что ненужные переменные, которые только занимают место, начинают пованивать. Есть разные мнения о том, что лучше – автоматическое или ручное управление памятью, но к теме данной книги это не относится. Достаточно знать, что в JavaScript есть сборщик мусора.

Наивный подход к управлению памятью заключается в том, чтобы освободить всю память, выделенную внутри функции, как только она закончит выполнение, – и тем самым уничтожить все в ней созданное. Действительно, раз функция завершилась, то доступ к чему-то внутри ее контекста выполнения вроде бы уже не нужен.

```
var prison = function () {  
    var prisoner = 'Джош Пауэлл';  
};  
prison();
```

После завершения функции `prison` нам больше не нужен доступ к переменной `prisoner`, поэтому Джош может идти, куда ему вздумается. Но такая запись громоздка, поэтому преобразуем ее в самовыполняющуюся анонимную функцию.

```
(function () {  
  var prisoner = 'Джош Пауэлл';  
})();
```

И здесь то же самое: функция завершилась, и переменную `prisoner` больше нет смысла хранить в памяти. Пока, Джош!

А теперь посмотрим, что при этом произойдет в паттерне модуля.

```
var prison = (function () {  
  var prisoner = 'Джош Пауэлл';
```

```
    return { prisoner: prisoner };  
})();
```

Сохранение переменной или функции в одноименном атрибуте объекта, возвращаемого из модуля, скоро войдет у нас в привычку, так как используется в книге повсеместно.

```
// выводится 'Джош Пауэлл'  
console.log( prison.prisoner );
```

Нам по-прежнему не нужен доступ к переменной `prisoner` после завершения анонимной функции. Поскольку строка Джош Пауэлл теперь сохранена в атрибуте `prison.prisoner`, нет смысла хранить переменную `prisoner` в памяти модуля, потому что она все равно недоступна. Значением `prison.prisoner` является строка Джош Пауэлл, этот атрибут больше не указывает на переменную `prisoner` – хотя из текста программы это не очевидно.

```
var prison = (function () {  
  var prisoner = 'Джош Пауэлл';
```

```
    return {  
      prisoner: function () {  
        return prisoner;  
      }  
    }  
  }  
})();
```

```
// выводится 'Джош Пауэлл'  
console.log( prison.prisoner() );
```

А теперь к переменной `prisoner` производится обращение всякий раз при выполнении метода `prison.prisoner()`. Этот метод возвращает текущее значение переменной `prisoner`. Если сборщик мусора вмешается и удалит эту переменную из памяти, то вызов `prison.prisoner()` вернет не Джош Пауэлл, а `undefined`.

Вот теперь наконец мы можем ответить на вопрос «Что такое замыкание?». Замыкание – это механизм, который не дает сборщику мусора удалить переменную из памяти за счет того, что сохраняет возможность доступа к переменной вне того контекста выполнения,

в котором она была создана. Замыкание создается, когда функция `prisoner` сохраняется в объекте `prison`. Замыкание создается вследствие сохранения функции, имеющей динамический доступ к переменной `prisoner` вне текущего контекста выполнения, — это не дает сборщику мусора удалить переменную `prisoner` из памяти.

Рассмотрим еще несколько примеров замыканий.

```
var makePrison = function ( prisoner ) {
    return function () {
        return prisoner;
    }
};

var joshPrison = makePrison( 'Джош Пауэлл' );
var mikePrison = makePrison( 'Майк Миковски' );

// выводится 'Джош Пауэлл', переменная prisoner сохранена в замыкании.
// Замыкание создано, потому что функция makePrison возвращает анонимную
// функцию, которая обращается к переменной prisoner.
console.log( joshPrison() );

// выводится 'Майк Миковски', переменная prisoner сохранена в замыкании.
// Замыкание создано, потому что функция makePrison возвращает анонимную
// функцию, которая обращается к переменной prisoner.
console.log( mikePrison() );
```

Еще одно типичное применение замыканий — использование сохраненных переменных после возврата из Ajax-вызова. В методах JavaScript переменная `this` ссылается на объект:

```
var prison = {
    names: 'Майк Миковски и Джош Пауэлл',
    who: function () {
        return this.names;
    }
};

// возвращает 'Майк Миковски и Джош Пауэлл'
prison.who();
```

Если ваш метод выполняет Ajax-вызов с помощью jQuery, то `this` ссылается уже не на ваш объект, а на Ajax-вызов:

```
var prison = {
    names: 'Майк Миковски и Джош Пауэлл',
    who: function () {
        $.ajax({
            success: function () {
                console.log( this.names );
            }
        });
    }
};
```



```
    }  
  });  
}  
};
```

```
// выводится undefined, 'this' - это объект ajax  
prison.who();
```

А как же тогда обратиться к своему объекту? На помощь приходят замыкания! Напомним, что замыкание создается, когда функция, имеющая доступ к переменной в текущем контексте выполнения, вызывается вне этого контекста. В следующем примере замыкание создается, потому что `this` сохраняется в переменной `that`, а доступ к `that` производится из функции, которая выполняется после возврата Ajax-вызова. Ajax-вызов асинхронный, поэтому ответ приходит, когда программа уже вышла из контекста выполнения, в котором этот вызов был сделан.

```
var prison = {  
  names: 'Mike Mikowski and Josh Powell',  
  who: function () {  
    var that = this;  
    $.ajax({  
      success: function () {  
        console.log( that.names );  
      }  
    });  
  }  
};  
};
```

Аjax-вызов асинхронный, поэтому к моменту, когда придет ответ, вызов `who()` уже завершится.

```
// выводится 'Майк Миковски и Джош Пауэлл'  
prison.who();
```

Но хотя метод `who()` и завершился к моменту возврата Ajax-вызова, переменная `that` не была удалена сборщиком мусора и остается доступной методу `success`. Хочется надеяться, что наше объяснение замыканий помогло вам понять, что это такое и как они работают. И теперь мы можем копнуть глубже и разобраться в том, как замыкания реализованы.

2.6.4. Замыкания

Как работают замыкания? Мы теперь понимаем, *что такое* замыкание, но остается вопрос, *как* они реализованы. Ответ кроется в объектах контекста выполнения. Рассмотрим пример из предыдущего раздела.

```
var makePrison = function ( prisoner ) {  
    return function () {  
        return prisoner;  
    }  
};  
  
var joshPrison = makePrison( 'Джош Пауэлл' );  
var mikePrison = makePrison( 'Майк Миковски' );  
  
// выводится 'Джош Пауэлл'  
console.log( joshPrison() );  
  
// выводится 'Майк Миковски'  
console.log( mikePrison() );
```

При вызове `makePrison` создается объект контекста выполнения для этого *конкретного* вызова, и переменной `prisoner` присваивается переданное значение. Напомним, что объект контекста выполнения – часть интерпретатора JavaScript, прямого доступа из скрипта к нему нет.

В примере выше мы обратились к `makePrison` дважды и сохранили результаты в переменных `joshPrison` и `mikePrison`. Поскольку `makePrison` возвращает функцию, то при присваивании результата переменной `joshPrison` счетчик ссылок на данный *конкретный* объект контекста выполнения становится равен 1, а так как счетчик больше нуля, то интерпретатор JavaScript не уничтожает данный *конкретный* объект контекста выполнения. Как только счетчик ссылок станет равен нулю, интерпретатор будет знать, что объект можно убирать в мусор.

При втором вызове `makePrison` и присваивании результата переменной `mikePrison` создается новый объект контекста выполнения, и счетчик ссылок на этот объект становится равным 1. В этот момент у нас есть два указателя на два объекта контекста выполнения, для каждого из которых счетчик ссылок равен единице, хотя созданы они в результате выполнения одной и той же функции.

Если бы мы снова вызвали `joshPrison`, она воспользовалась бы значением, установленным в объекте контекста выполнения, который был создан при вызове `makePrison` и сохранен в `joshPrison`. Единственный способ избавиться от сохраненного объекта контекста выполнения (если не считать закрытия веб-страницы, умники) – удалить переменную `joshPrison`. В этом случае счетчик ссылок на связанный с ней объект контекста выполнения обратится в нуль, и интерпретатор сможет удалить его, когда сочтет удобным.

Давайте создадим сразу несколько объектов контекста выполнения и посмотрим, что происходит.

Листинг 2.12 ❖ Объекты контекста выполнения

```
var curryLog, logHello, logStayinAlive, logGoodbye;

curryLog = function ( arg_text ){
    var log_it = function (){ console.log( arg_text ); };
    return log_it;
};

logHello = curryLog('привет');
logStayinAlive = curryLog('жив, курилка!');
logGoodbye = curryLog('пока');

// Здесь не создается ссылок на контекст выполнения, а значит,
// объект контекста выполнения может быть сразу удален
// сборщиком мусора JavaScript
curryLog('fred');

logHello();           // выводится 'привет'
logStayinAlive();     // выводится 'жив, курилка!'
logGoodbye();         // выводится 'пока'
logHello();           // снова выводится 'привет'

// уничтожается ссылка на контекст выполнения 'привет'
delete window.logHello;

// уничтожается ссылка на контекст выполнения 'жив, курилка'
delete window.logStayinAlive;

logGoodbye();         // выводится 'пока'
logStayinAlive();     // undefined - контекст выполнения уничтожен
```

Следует помнить, что при каждом вызове функции создается новый объект контекста выполнения. По завершении функции этот объект удаляется, *если только вызывающая программа не сохраняет ссылку на него*. Если функция возвращает число, то сохранить ссылку на объект контекста выполнения, вообще говоря, невозможно. С другой стороны, если функция возвращает более сложную структуру, например функцию, объект или массив, то в результате сохранения возвращенного значения в переменной часто создается ссылка на контекст выполнения – иногда непреднамеренно.

Можно создавать цепочки ссылок на контексты выполнения очень большой глубины. И это хорошо в тех случаях, когда требуется (вспомните о *наследовании объектов*). Но бывает, что нам вовсе не нужны такие замыкания, потому что они приводят к *утечке памяти*.

Правила и инструменты, описанные в приложении А, помогут вам избежать непреднамеренного создания замыканий.

Замыкания – еще раз!

Поскольку замыкания – очень важная часть JavaScript, вызывающая немало путаницы, то, прежде чем двигаться дальше, предпримем еще одну попытку объяснения. Если вам уже все с замыканиями ясно, можете пропустить эту врезку.

```
var menu, outer_function,
    food = 'cake';

outer_function = function () {
    var fruit, inner_function;

    fruit = 'apple';

    inner_function = function () {
        return { food: food, fruit: fruit };
    }

    return inner_function;
};

menu = outer_function();

// возвращается { food: 'cake', fruit: 'apple' }
menu();
```

При выполнении `outer_function` создается контекст выполнения. `inner_function` определена внутри этого контекста.

Поскольку `inner_function` определена внутри контекста выполнения `outer_function`, она имеет доступ ко всем переменным в области видимости `outer_function` – в данном случае `food`, `fruit`, `outer_function`, `inner_function` и `menu`.

Если вы думаете, что после завершения `outer_function` все находящееся внутри ее контекста выполнения уничтожается сборщиком мусора, то вы неправы.

Уничтожения не происходит, потому что ссылка на `inner_function` сохранена в переменной `menu`, определенной в глобальной области видимости. Поскольку `inner_function` должна сохранить доступ ко всем переменным, которые находились в области видимости, когда она была объявлена, то она «замыкает» контекст выполнения `outer_function`, чтобы не дать сборщику мусора удалить его. Это и есть замыкание.

Это возвращает нас к первому примеру – посмотрим, почему переменная `scoped_var` доступна после возврата из Ajax-вызова.

```
function sendAjaxRequest() {
    var scoped_var = 'yay';
```

```
$.ajax({
  success: function () {
    console.log(scoped_var);
  }
});
}
sendAjaxRequest(); ← Когда Ajax-вызов успешно завершится, выводится «уау».
```

Она доступна, потому что метод `success` определен в контексте выполнения, созданном при вызове `sendAjaxRequest`, а переменная `scoped_var` в этот момент находилась в области видимости. Если замыкания все еще вам непонятны, не отчаивайтесь. Замыкания – одна из самых трудных концепций в JavaScript, и если, прочитав этот раздел несколько раз, вы так и не ухватили сути, просто читайте дальше; может быть, для понимания вам просто необходим практический опыт. Надеемся, что к концу книги у вас будет столько практического опыта, что замыкания станут вашей второй натурой.

И на этом мы завершаем стремительный, а временами углубленный, обзор некоторых деталей JavaScript. Конечно, он был неполным, но мы решили, что вместо подробного изучения всех аспектов языка лучше заняться разработкой больших SPA. Надеемся, что прогулка вам понравилась.

2.7. Резюме

В этой главе мы рассмотрели некоторые концепции, пусть даже не уникальные для JavaScript, а встречающиеся и в других широко распространенных языках программирования. Знакомство с ними важно для написания одностраничных приложений – без этого вы могли бы растеряться.

Понимать, что такое область видимости переменных, а также понятие переменных и функций, необходимо, чтобы сбросить покров тайны с переменных в JavaScript. А чтобы понимать, как работают области видимости и поднятие, важно разобраться в объектах контекста выполнения.

Зная, как в JavaScript создаются объекты с помощью прототипов, вы сможете писать на этом языке повторно используемый код. Программисты, не понимающие, что такое прототипическое наследование, часто прибегают для написания повторно используемого кода к библиотекам, которые предоставляют модель на основе классов, но на самом деле лишь обертывают прототипическую модель. Для построения одностраничных приложений мы будем использовать мо-

дель на основе прототипов по двум причинам: мы считаем, что для наших целей она проще, и что, работая на JavaScript, нужно пользоваться его собственными, а не привнесенными механизмами.

Самовыполняющиеся анонимные функции инкапсулируют область видимости переменных и тем самым помогают предотвратить случайное засорение глобального пространства имен, а значит, позволяют писать библиотеки и приложения, не конфликтующие с другими библиотеками.

Понимание паттерна модуля и механизма закрытых переменных позволяет раскрывать продуманный открытый API объектов и скрывать внутренние методы и переменные, о которых другим объектам знать не нужно. В результате становится понятным, какие методы предназначены для клиентов, а какие служат лишь подспорьем для реализации API.

Наконец, мы потратили немало времени на изучение одной из самых трудных концепций JavaScript: замыканий. Если вы еще не до конца поняли, как они работают, то надеемся, что обширный практический опыт, который вы накопите, читая эту книгу, поможет закрепить материал. А теперь, вооружившись знаниями, перейдем к следующей главе и начнем создавать SPA промышленного качества.

Клиентская часть одностраничного приложения

Клиентская часть SPA – нечто гораздо большее, чем пользовательский интерфейс традиционного сайта. Некоторые говорят, что SPA-клиенты могут отзываться так же быстро, как персональные приложения, но правильнее будет сказать, что хорошо написанный SPA-клиент и есть персональное приложение.

Как и персональное приложение, SPA-клиент существенно отличается от традиционной веб-страницы. При замене обычного сайта одностраничным приложением изменяется весь программный стек, включая сервер базы данных и систему HTML-шаблонов. Компании, достаточно дальновидные, чтобы совершить успешный переход от традиционных сайтов к SPA, поняли, что старые способы и организация работы должны измениться. Они перенацелили инженеров, методики и тестировщиков на разработку клиентской части. Сервер остается важным компонентом, но теперь его основной обязанностью является предоставление данных в формате JSON.

Поэтому забудем все, что знаем о разработке клиентской части традиционных веб-сайтов. Впрочем, не все – знать, что такое JavaScript, HTML5, CSS3, SVG, CORS и еще ряд акронимов, по-прежнему необходимо. Но, читая последующие главы, вы должны постоянно помнить, что *создаете персональное приложение*, а не традиционный сайт. Во второй части мы рассмотрим следующие вопросы:

- создание и тестирование масштабируемого, тестопригодного и функционально насыщенного SPA-клиента;
- наделение кнопки «Назад», закладок и других элементов управления историей привычной функциональностью;
- проектирование, реализация и тестирование надежных модулей и их API;

- обеспечение естественного интерфейса как на мобильных, так и на настольных устройствах;
- организация модулей и пространств имен с расчетом на удобство тестирования, коллективную разработку и проектирование с учетом качества.

А вот чего мы не будем обсуждать, так это использования конкретного каркаса для разработки SPA. Тому есть много причин (см. подробную аргументацию на врезке в главе 6). Мы хотим объяснить принципы работы хорошо написанного SPA, а не вдаваться в тонкости реализации одного-единственного каркаса или библиотеки. Вместо этого мы будем пользоваться архитектурой, которую оттачивали на протяжении шести лет на многих коммерческих продуктах. В этой архитектуре заложены тестопригодность, удобочитаемость и проектирование с учетом качества. Кроме того, она позволяет без труда распределить работу между несколькими разработчиками клиентской части. При таком подходе читатель, который *хочет* воспользоваться каким-нибудь каркасом, сможет принять обоснованное решение и добиться большего успеха.

Глава 3

Разработка оболочки

В этой главе:

- ✧ Модуль Shell и его место в нашей архитектуре.
- ✧ Структура файлов и пространств имен.
- ✧ Создание и стилизация функциональных контейнеров.
- ✧ Применение обработчика событий для переключения функционального контейнера.
- ✧ Использование паттерна якорного интерфейса для управления состоянием приложения.

В этой главе мы опишем модуль оболочки Shell, являющийся неотъемлемым компонентом нашей архитектуры. Мы разработаем макет страницы, содержащей функциональные контейнеры, а затем настроим Shell для их отрисовки. Затем мы покажем, как Shell управляет функциональными контейнерами, применив его для сворачивания и раскрытия окна чата. Потом поручим ему перехват события щелчка мышью, по которому сворачивается и раскрывается окно. Наконец, мы воспользуемся якорем в URI для организации API-состояния, применив *паттерн якорного интерфейса*. Это даст пользователям привычные элементы управления браузером – кнопки «Вперед» и «Назад», браузерную историю и закладки.

К концу главы у нас будет готов фундамент масштабируемого управляемого SPA. Но не будем забегать вперед. Сначала нужно понять, что такое Shell.

3.1. Знакомимся с Shell

Shell – это главный контроллер SPA, необходимая часть нашей архитектуры. Роль модуля Shell сопоставима с ролью корпуса самолета.

Корпус самолета (его называют также монококом или каркасом) придает летательному аппарату форму и структуру. С помощью различных крепежных конструкций к корпусу крепятся сиденья, обеденные столики и двигатели. Все узлы делаются максимально независимыми, потому что никто не хочет, чтобы самолет резко накренился вправо, как только тетушка Молли откинёт свой столик.

Модуль Shell придает форму и структуру приложению. Функциональные модули типа чата, входа в систему и навигации «крепятся» к Shell с помощью API. Все функциональные модули настолько независимы, насколько возможно, потому что никому не понравится, что приложение закрывает окно браузера, как только тетушка Молли вводит в своем чате текст «ROTFLMAO!!! Нам всем конец!»¹.

Shell – лишь одна часть той архитектуры, которую мы оттачивали на многих коммерческих проектах. Эта архитектура – и место в ней модуля Shell – показана на рис. 3.1. Мы предпочитаем сначала писать Shell, потому что это столп всей конструкции. Он координирует взаимодействие *функциональных модулей*, в которых сосредоточена бизнес-логика, с универсальными браузерными интерфейсами, например URI и куками. Когда пользователь нажимает кнопку «Назад», входит в приложение или делает еще что-то, изменяющее состояние приложения настолько, что имеет смысл создавать новую закладку, модуль Shell координирует изменения.

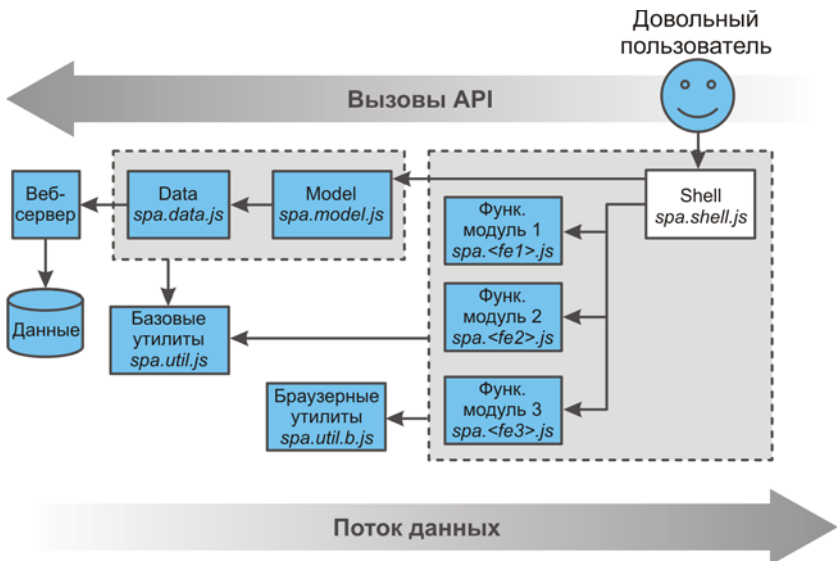


Рис. 3.1 ❖ Место модуля Shell в архитектуре SPA

¹ Rolling On The Floor Laughing My Ass/Arse Off – Катаюсь по полу, надрываясь со смеху. – Прим. перев.

Читатели, знакомые с архитектурой модель–представление–контроллер (MVC), могут считать модуль Shell главным контроллером, поскольку он координирует работу контроллеров всех подчиненных функциональных модулей.

Shell отвечает за решение следующих задач:

- отрисовка и управление функциональными контейнерами;
- управление состоянием приложения;
- координация функциональных модулей.

Подробнее о координации функциональных модулей мы будем говорить в следующей главе. А в этой рассмотрим отрисовку функциональных контейнеров и управление состоянием приложения. Но сначала подготовим файлы и пространства имен.

3.2. Организация файлов и пространств имен

При организации файлов и пространств имен мы будем следовать стандартам кодирования, описанным в приложении А. В частности, каждому пространству имен JavaScript будет соответствовать ровно один JS-файл, а чтобы предотвратить засорение глобального пространства имен, мы будем использовать самовыполняющиеся анонимные функции. Кроме того, мы создадим параллельное дерево CSS-файлов. Такое соглашение ускоряет разработку, повышает качество и упрощает сопровождение. Его полезность тем очевиднее, чем больше в проекте модулей и разработчиков.

3.2.1. Создание дерева файлов

Мы решили назвать корневое пространство имен нашего приложения `spr`. Имена файлов, содержащих JavaScript и CSS, имена пространства имен JavaScript и имена CSS-селекторов, будут согласованы. Так намного проще сопоставлять JavaScript-код с таблицей CSS-стилей.

Планирование структуры каталогов и файлов

Веб-разработчики часто помещают HTML-файл в некоторый каталог, а CSS- и JavaScript-файлы – в его подкаталоги. Мы не видим причин отступать от этого соглашения. Создадим каталоги и файлы, как показано в листинге 3.1.

Листинг 3.1 ❖ Файлы и каталоги, первый проход

```

spa ← spa – наш корневой каталог и корневое пространство имен.
+-- css ← Этот каталог содержит все файлы стилей.
| +-- spa.css
| '-- spa.shell.css
+-- js ← Этот каталог содержит все JavaScript-файлы.
| +-- jq ← Этот каталог содержит JavaScript-файлы, относя-
| +-- spa.js ← щиеся к jQuery, в том числе подключаемые модули.
| '-- spa.shell.js ←
+-- layout.html ←
'-- spa.html ←

```

Файл `spa.js` содержит наше корневое пространство имен `spa`. Соответствующая ему таблица стилей находится в файле `css/spa.css`.

Файл `spa.shell.js` содержит пространство имен модуля `Shell` – `spa.shell`. Соответствующая ему таблица стилей находится в файле `css/spa.shell.css`.

Этот файл браузер читает для запуска нашего SPA.

Подготовив место, установим jQuery.

Установка jQuery и подключаемого модуля

Библиотека jQuery и подключаемые к ней модули обычно предлагаются в виде минимизированных или полных файлов. Мы почти всегда устанавливаем полные файлы, потому что так проще отлаживать приложение, а минимизация все равно производится нашей системой сборки. Не думайте пока об этом – всемо свое время.

Библиотека jQuery содержит полезные кросс-платформенные средства манипулирования моделью DOM и другие утилиты. Мы используем версию 1.9.1, которую можно скачать со страницы http://docs.jquery.com/Downloading_jQuery. Поместим этот файл в каталог jQuery:

```

...
+-- js
| +-- jq
| | +-- jquery-1.9.1.js
...

```

Подключаемый к jQuery модуль `uriAnchor` предоставляет средства для работы с якорным компонентом URI-адреса. Его можно скачать из хранилища github по адресу <https://github.com/mmikowski/urianchor>. Поместим его тоже в каталог jQuery:

```

...
+-- js
| +-- jq
| | +-- jquery.uriAnchor-1.1.3.js
...

```

Теперь дерево файлов и каталогов выглядит, как показано в листинге 3.2:

Листинг 3.2 ❖ Дерево файлов и каталогов после добавления jQuery и подключаемого модуля

```

spa
+-- css
| +-- spa.css
| `-- spa.shell.css
+-- js
| +-- jq
| | +-- jquery-1.9.1.js
| | `-- jquery.uriAnchor-1.1.3.js
| +-- spa.js
| `-- spa.shell.js
+-- layout.html
`-- spa.html

```

Расставив все файлы по местам, приступим к написанию HTML, CSS и JavaScript.

3.2.2. HTML-файл приложения

Открыв в браузере наш документ (spa/spa.html), мы можем насладиться прелестью созданного к данному моменту SPA. Правда, пока что файл пуст, и вся прелесть сводится к отсутствию ошибок на безопаснейшей странице, которая не делает абсолютно ничего. Надо бы вылечить этот синдром «пустой страницы».

Документ, открываемый в браузере (spa/spa.html), был и останется небольшим. У него есть только одна роль – загрузить библиотеки и таблицы стилей, а потом запустить приложение. Откройте свой любимый текстовый редактор и добавьте весь код, который потребуется нам в этой главе (см. листинг 3.3).

Листинг 3.3 ❖ HTML-файл приложения – spa/spa.html

```

<!doctype html>
<html>
<head>
  <title>SPA Starter</title>
  <!-- таблицы стилей -->
  <link rel="stylesheet" href="css/spa.css" type="text/css"/>
  <link rel="stylesheet" href="css/spa.shell.css" type="text/css"/>

  <!-- сторонний javascript -->
  <script src="js/jq/jquery-1.9.1.js" ></script>
  <script src="js/jq/jquery.uriAnchor-1.1.3.js"></script>

  <!-- наш javascript -->
  <script src="js/spa.js" ></script>

```

Затем загружаем сторонний JavaScript-код. Пока что единственный сторонний код – библиотека jQuery и подключаемый модуль для манипулирования якорями.

Сначала загружаются таблицы стилей. Это повышает скорость загрузки. Все сторонние таблицы стилей нужно загружать первыми.

Затем загружаем наши JavaScript-библиотеки в порядке возрастания уровня вложенности пространства имен. Это важно, потому что объект пространства имен spa должен быть объявлен раньше своих потомков, например spa.shell.

```

<script src="js/spa.shell.js"></script>
<script>
  $(function () { spa.initModule( $('#spa') ); }); ←
</script>
</head>
<body>
  <div id="spa"></div>
</body>
</html>

```

Инициализируем приложение, как только модель DOM будет готова к работе. Читатели, знакомые с jQuery, заметили, что мы используем сокращенную запись: `$(function (...))` – то же самое, что `$(document).ready(function (...))`.

Разработчики, пекущиеся о производительности, могли бы спросить: «Почему скрипты не помещены в конец контейнера `body`, как в традиционных веб-страницах?» Вопрос справедливый, поскольку при такой организации страница обычно отрисовывается быстрее, так как статическую HTML-разметку, стилизованную с помощью CSS, можно отобразить еще до того, как закончится загрузка JavaScript-кода. Но SPA-то работают не так. Они генерируют HTML-разметку с помощью JavaScript-кода, поэтому помещать скрипты вне заголовка бессмысленно – никакого ускорения не будет. Поэтому мы поместили все внешние скрипты в секцию `head`, чтобы организация страницы была отчетливо видна.

3.2.3. Создание корневого пространства имен CSS

Наше корневое пространство имен называется `spa`, и по соглашению, описанному в приложении А, корневая таблица стилей должна называться `spa/css/spa.css`. Этот файл мы создали раньше, а теперь его надо наполнить. Поскольку это корневая таблица стилей, то в ней будет немного больше секций, чем в других CSS-файлах. Снова воспользуйтесь своим любимым редактором и добавьте правила, показанные в листинге 3.4.

Листинг 3.4 ❖ Корневое пространство имен CSS – `spa/css/spa.css`

```

/*
 * spa.css
 * Стили в корневом пространстве имен
 */

/** Начало установки начальных значений */ ←
* {
  margin : 0;
  padding : 0;
  -webkit-box-sizing : border-box;
  -moz-box-sizing : border-box;
  box-sizing : border-box;

```

Сбросить большинство селекторов. Мы не доверяем умолчаниям браузера. Среди авторов CSS-стилей эта практика широко распространена, хотя и не без разногласий.

```

}
h1,h2,h3,h4,h5,h6,p { margin-bottom : 10px;
ol,ul,dl { list-style-position : inside;}
/** Конец установки начальных значений */

/** Начало стандартных селекторов */
body {
    font : 13px 'Trebuchet MS', Verdana, Helvetica, Arial, sans-serif;
    color : #444;
    background-color : #888;
}
a { text-decoration : none; }
a:link, a:visited { color : inherit; }
a:hover { text-decoration: underline; }

strong {
    font-weight : 800;
    color : #000;
}
/** Конец стандартных селекторов */

/** Начало селекторов в пространстве имен spa */
#spa {
    position : absolute;
    top : 8px;
    left : 8px;
    bottom : 8px;
    right : 8px;

    min-height : 500px;
    min-width : 500px;
    overflow : hidden;

    background-color : #fff;
    border-radius : 0 8px 0 8px;
}
/** Конец селекторов в пространстве имен spa */

/** Начало служебных селекторов */
.spa-x-select {}
.spa-x-clearfloat {
    height : 0 !important;
    float : none !important;
    visibility : hidden !important;
    clear : both !important;
}
/** Конец служебных селекторов */

```

Настроить стандартные селекторы. И снова мы не доверяем умолчаниям браузера, а также хотим, чтобы элементы некоторых типов одинаково выглядели во всем приложении. Внешний вид конкретных элементов может быть — и будет — уточнен более специфичными селекторами в других файлах.

Определяем селектор пространства имен. Обычно в этом качестве используется селектор элемента с идентификатором, совпадающим с именем корневого пространства имен JavaScript, — в нашем случае #spa.

Определяем служебные селекторы, используемые в разных модулях. Все они снабжаются префиксом spa-x-.

Согласно нашим стандартам кодирования, все идентификаторы и имена классов CSS в этом файле должны начинаться префиксом spa-.

Теперь, когда у нас есть CSS-файл для корневого приложения, создадим соответствующее пространство имен JavaScript.

3.2.4. Создание корневого пространства имен JavaScript

Наше корневое пространство имен называется `spa`, и, по соглашению из приложения А, корневой JavaScript-файл должен называться `spa/js/spa.js`. Минимально необходимый JavaScript-код сводится к предложению `var spa = {};`. Но мы хотим добавить метод для инициализации приложения и удостовериться, что код проходит проверки JSLint. Мы можем воспользоваться шаблоном из приложения А, сократив его, так как нам нужны не все секции. Откройте файл в редакторе и скопируйте в него текст из листинга 3.5:

Листинг 3.5 ❖ Корневое пространство имен JavaScript – `spa/js/spa.js`

```
/*
 * spa.js
 * Модуль корневого пространства имен
 */

/*jslint      browser : true,   continue : true,
  devel : true,   indent : 2,     maxerr : 50,
  newcap : true,  nomen : true,   plusplus : true,
  regexp : true,  sloppy : true,   vars : false,
  white : true
*/
/*global $, spa */

var spa = (function () {
  var initModule = function ( $container ) {
    $container.html(
      '<h1 style="display:inline-block; margin:25px;">'
      + 'hello world!'
      + '</h1>'
    );
  };

  return { initModule: initModule };
})();
```

Флаги JSLint, задаваемые в шаблоне модуля, см. в приложении А.

Сообщаем JSLint, что глобальные переменные `spa` и `$` ожидаемы. Поймав себя на добавлении еще каких-то собственных переменных в этот список после `spa`, задумайтесь, а то ли вы делаете.

Используем паттерн модуля из главы 2 для создания пространства имен `spa`. Этот модуль экспортирует единственный метод `initModule`, который инициализирует приложение.

Мы хотим быть уверены, что наш код не содержит типичных ошибок или порицаемых приемов. В приложении А показано, как установить и запустить полезнейшую утилиту JSLint, которая именно такую гарантию и дает. Там же описано, что означают флаги `/*jslint ... */` в начале файла. JSLint обсуждается не только в приложении, но и в главе 5.

Проверим наш код, набрав `jslint spa/js/spa.js` в командной строке, — не должно появиться никаких ошибок или предупреждений. Теперь можно открыть в браузере документ `spa/spa.html` и наблюдать обязательную демонстрацию «hello world», показанную на рис. 3.2.



Рис. 3.2 ❖ Обязательное приветствие «hello world»

Поприветствовав мир и воодушевившись сладким чувством успеха, приложим свои силы к более амбициозному предприятию. В следующем разделе мы начнем строить наше первое «настоящее» SPA.

3.3. Создание функциональных контейнеров

Модуль Shell создает и управляет контейнерами, которые используют функциональные модели. К примеру, контейнер нашего окна чата, следуя общепринятым соглашениям, привязан к правому нижнему углу окна браузера. Shell отвечает за сам контейнер чата, но не за его содержимое — управление поведением контейнера возлагается на функциональный модуль Chat, который мы будем рассматривать в главе 6.

Поместить окно чата в макет сравнительно просто. На рис. 3.3 схематически показано желательное расположение контейнеров на странице.

Разумеется, это только схема. Ее еще нужно превратить в HTML и CSS. Поговорим о том, как это сделать.

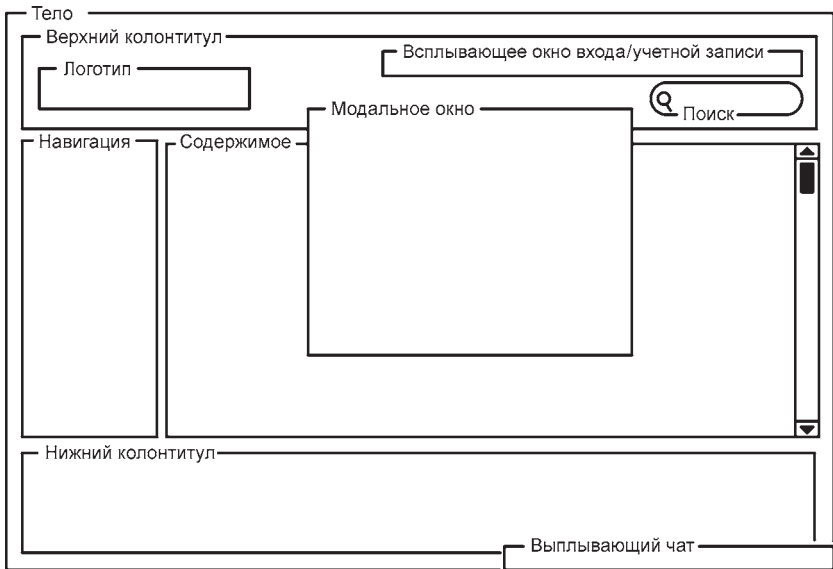


Рис. 3.3 ❖ Схема контейнеров приложения

3.3.1. Стратегия

Мы будем разрабатывать HTML- и CSS-стили функциональных контейнеров прямо в файле макета `spa/layout.html`. И лишь получив удовлетворительное представление контейнеров, мы перенесем код в файлы, содержащие CSS- и JavaScript-код модуля Shell. Это обычно самый быстрый и эффективный способ создания начального макета, потому что не нужно заботиться о взаимодействиях с другим кодом.

Сначала напишем HTML-код, а затем добавим стили.

3.3.2. HTML-код модуля Shell

Замечательной особенностью HTML5 и CSS3 является возможность полностью отделить стилизацию от содержимого. На схеме показано, какие нам нужны контейнеры и как они вкладываются друг в друга. Это все, что необходимо для уверенного написания HTML-кода контейнеров. Откройте файл макета (`spa/layout.html`) и скопируйте в него HTML-код из листинга 3.6.

Листинг 3.6 ❖ Создание HTML-кода контейнеров – `spa/layout.html`

```
<!doctype html>
<html>
```

```

<head>
  <title>HTML Layout</title>
  <link rel="stylesheet" href="css/spa.css" type="text/css"/>
</head>
<body>
  <div id="spa">
    <div class="spa-shell-head">
      <div class="spa-shell-head-logo"></div>
      <div class="spa-shell-head-acct"></div>
      <div class="spa-shell-head-search"></div>
    </div>
    <div class="spa-shell-main">
      <div class="spa-shell-main-nav"></div>
      <div class="spa-shell-main-content"></div>
    </div>
    <div class="spa-shell-foot"></div>
    <div class="spa-shell-chat"></div>
    <div class="spa-shell-modal"></div>
  </div>
</body>
</html>

```

Вкладываем Logo, параметры учетной записи (acct) и поле поиска в контейнер head.

Помещаем контейнеры навигации (nav) и содержимого (content) внутрь главного контейнера.

Создаем контейнер footer.

Привязываем контейнер chat к правому нижнему углу внешнего контейнера.

Создаем контейнер modal, расположенный над всем остальным содержимым.

Теперь необходимо убедиться, что HTML-код не содержит ошибок. Нам нравится почтенный инструмент Tidy, который умеет находить пропущенные теги и другие типичные ошибки в HTML-разметке. Программу Tidy можно использовать в онлайнном режиме на сайте <http://infohound.net/tidy/> или скачать с сайта <http://tidy.sourceforge.net/>. Если вы работаете с каким-нибудь дистрибутивом Linux, например Ubuntu или Fedora, то Tidy, скорее всего, уже имеется в стандартных репозиториях программ. Теперь свяжем с контейнерами стили.

3.3.3. CSS-стили модуля Shell

Мы напишем CSS-стили, имея в виду *эластичную верстку*, когда ширина и высота содержимого адаптируются к размерам окна браузера, если они не выходят за пределы разумного. Мы раскрасим контейнеры в разные цвета, чтобы они были сразу видны. Мы также откажемся от рамок, потому что они могут изменить размер блоков CSS. Это делает процесс быстрого создания прототипа несколько скучным, но, как только нас устроит внешний вид контейнеров, мы сможем добавить рамки, если захотим.

Эластичная верстка

Если макет достаточно сложный, то для придания ему *эластичности* может потребоваться JavaScript-код. Часто создается обработчик события изменения размера окна, в котором вычисляются и применяются к стилям новые размеры. Эту технику мы продемонстрируем в главе 4.

Поместим CSS-код в секцию `<head>` документа `spa/layout.html`. Можно расположить его сразу после ссылки на таблицу стилей `spa.css`, как показано в листинге 3.7. Изменения выделены **полужирным шрифтом**.

Листинг 3.7 ❖ Создание CSS-стилей для контейнеров – `spa/layout.html`

```
...
<head>
  <title>HTML Layout</title>
  <link rel="stylesheet" href="css/spa.css" type="text/css"/>
  <style>
    .spa-shell-head, .spa-shell-head-logo, .spa-shell-head-acct,
    .spa-shell-head-search, .spa-shell-main, .spa-shell-main-nav,
    .spa-shell-main-content, .spa-shell-foot, .spa-shell-chat,
    .spa-shell-modal {
      position : absolute;
    }
    .spa-shell-head {
      top    : 0;
      left   : 0;
      right  : 0;
      height : 40px;
    }
    .spa-shell-head-logo {
      top      : 4px;
      left     : 4px;
      height   : 32px;
      width    : 128px;
      background : orange;
    }
    .spa-shell-head-acct {
      top      : 4px;
      right    : 0;
      width    : 64px;
      height   : 32px;
      background : green;
    }
    .spa-shell-head-search {
      top      : 4px;
      right    : 64px;
      width    : 248px;
      height   : 32px;
      background : blue;
    }
    .spa-shell-main {
      top      : 40px;
      left     : 0;
      bottom   : 40px;
      right    : 0;
```

```
}
.spa-shell-main-content,
.spa-shell-main-nav {
  top    : 0;
  bottom : 0;
}
.spa-shell-main-nav {
  width    : 250px;
  background : #eee;
}
.spa-x-closed .spa-shell-main-nav {
  width : 0;
}
.spa-shell-main-content {
  left    : 250px;
  right   : 0;
  background : #ddd;
}
.spa-x-closed .spa-shell-main-content {
  left : 0;
}
.spa-shell-foot {
  bottom : 0;
  left   : 0;
  right  : 0;
  height : 40px;
}
.spa-shell-chat {
  bottom    : 0;
  right     : 0;
  width     : 300px;
  height    : 15px;
  background : red;
  z-index   : 1;
}
.spa-shell-modal {
  margin-top    : -200px;
  margin-left   : -200px;
  top          : 50%;
  left         : 50%;
  width        : 400px;
  height       : 400px;
  background    : #fff;
  border-radius : 3px;
  z-index       : 2;
}
</style>
</head>
...
```

Открыв файл `spa/layout.html` в браузере, мы увидим страницу, которая очень напоминает схему размещения контейнеров (рис. 3.4). Изменив размер браузера, можно наблюдать, как меняются размеры функциональных контейнеров. У нашего эластичного макета есть ограничение – если высота или ширина окна меньше 500 пикселей, появляются полосы прокрутки. Так сделано потому, что далее уменьшать размеры области содержимого нецелесообразно.



Рис. 3.4 ❖ HTML и CSS для контейнеров – `spa/layout.html`

Для экспериментов со стилями, которые не используются в первом варианте макета, мы можем воспользоваться инструментами разработчика в Chrome. Например, добавим класс `spa-x-closed` к контейнеру `spa-shell-main`. Тем самым мы закроем панель навигации в левой части страницы. Если убрать этот класс, то панель навигации восстановится (рис. 3.5).

3.4. Отрисовка функциональных контейнеров

Созданный макет (`spa/layout.html`) – неплохое начало. Теперь мы попробуем использовать его в нашем SPA. Первый шаг – отказаться от статического HTML и CSS и поручить отрисовку контейнеров модулю Shell.

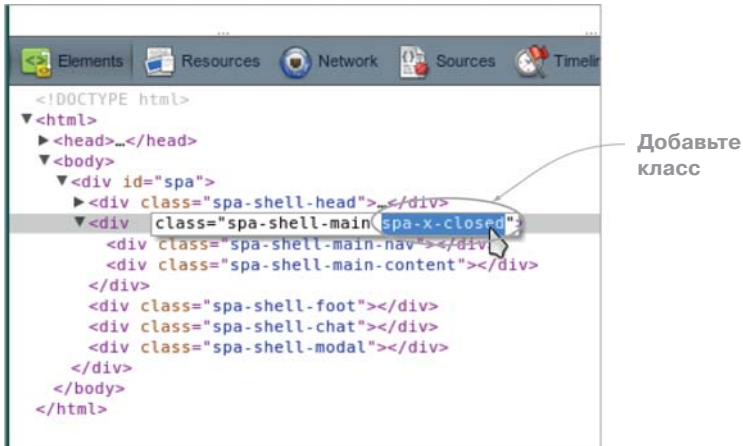


Рис. 3.5 ❖ Дважды щелкните мышью внутри HTML-разметки, отображаемой на панели инструментов разработчика в Chrome, чтобы добавить класс

3.4.1. Преобразование HTML в JavaScript-код

Мы хотим, чтобы все изменения в документе управлялись JavaScript-кодом; поэтому необходимо преобразовать написанную ранее HTML-разметку в JavaScript-строку. Чтобы было проще читать и сопровождать код, мы сохраним отступы, как показано в листинге 3.8:

Листинг 3.8 ❖ Представление HTML-шаблона в виде конкатенации строк

```

var main_html = String()
+ '<div class="spa-shell-head">'
+ '<div class="spa-shell-head-logo"></div>'
+ '<div class="spa-shell-head-acct"></div>'
+ '<div class="spa-shell-head-search"></div>'
+ '</div>'
+ '<div class="spa-shell-main">'
+ '<div class="spa-shell-main-nav"></div>'
+ '<div class="spa-shell-main-content"></div>'
+ '</div>'
+ '<div class="spa-shell-foot"></div>'
+ '<div class="spa-shell-chat"></div>'
+ '<div class="spa-shell-modal"></div>';

```

Нас не беспокоит снижение производительности из-за конкатенации строк. Когда мы будем готовы к промышленной эксплуатации, минимизатор JavaScript объединит строки.


```

+ '</div>'
+ '<div class="spa-shell-main">'
+ '<div class="spa-shell-main-nav"></div>'
+ '<div class="spa-shell-main-content"></div>'
+ '</div>'
+ '<div class="spa-shell-foot"></div>'
+ '<div class="spa-shell-chat"></div>'
+ '<div class="spa-shell-modal"></div>'
},
stateMap = { $container : null }, ←
jqueryMap = {}, ← Кэшируем коллекции jQuery в объекте jqueryMap.

setjQueryMap, initModule; ←
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

В этой секции объявляем все переменные в области
видимости модуля. Многим из них значения будут при-
своены позже.

//----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ ----- ←
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

В секцию «Служебные методы» помещаются функции,
которые не взаимодействуют с элементами страницы.

//----- НАЧАЛО МЕТОДОВ DOM ----- ←
// Начало метода DOM /setjQueryMap/
setjQueryMap = function () { ←
    var $container = stateMap.$container;
    jqueryMap = { $container : $container };
};
// Конец метода DOM /setjQueryMap/
//----- КОНЕЦ МЕТОДОВ DOM -----

Функция setjQueryMap служит для кэширования коллекций jQuery. Она должна
присутствовать практически во всех написанных нами оболочках и функциональных
модулях. Кэш jqueryMap позволяет существенно уменьшить количество прохо-
дов jQuery по документу и повысить производительность.

Секция «Обработчики событий» зарезервирована для функций обработки событий.

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ ----- ←
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ ----- ←
// Начало открытого метода /initModule/
initModule = function ( $container ) { ←
    stateMap.$container = $container;
    $container.html( configMap.main_html );
    setjQueryMap();
};
// Конец открытого метода /initModule/

В секцию «Открытые методы» помещаются
открытые методы.

Открытый метод initModule используется
для инициализации модуля.

Явно экспортируем открытые методы, возвращая их
в хэше. В настоящее время есть только один откры-
тый метод – initModule.

return { initModule : initModule }; ←
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();

```

Теперь у нас есть модуль, который отрисовывает функциональные контейнеры, но нам еще необходимо заполнить CSS-файл и сообщить модулю в корневом пространстве имен (`spa/js/spa.js`) о необходимости использовать модуль Shell (`spa/js/spa.shell.js`) вместо вывода достопочтенного сообщения «hello world». Займемся этим.

3.4.3. Создание таблицы стилей для Shell

В соответствии с соглашениями, приведенными в приложении А, селекторы, начинающиеся префиксом `spa-shell-*`, необходимо поместить в файл `spa/css/spa.shell.css`. Мы можем просто скопировать CSS-стили из файла `spa/layout.html` в этот файл, как показано в листинге 3.10:

Листинг 3.10 ❖ CSS-стили для Shell, попытка 1 – `spa/css/spa.shell.css`

```
/*
 * spa.shell.css
 * Стили для Shell
 */
.spa-shell-head, .spa-shell-head-logo, .spa-shell-head-acct,
.spa-shell-head-search, .spa-shell-main, .spa-shell-main-nav,
.spa-shell-main-content, .spa-shell-foot, .spa-shell-chat,
.spa-shell-modal {
    position : absolute;
}
.spa-shell-head {
    top    : 0;
    left   : 0;
    right  : 0;
    height : 40px;
}
.spa-shell-head-logo {
    top      : 4px;
    left     : 4px;
    height   : 32px;
    width    : 128px;
    background : orange;
}
.spa-shell-head-acct {
    top      : 4px;
    right    : 0;
    width    : 64px;
    height   : 32px;
    background : green;
}
.spa-shell-head-search {
    top      : 4px;
    right    : 64px;
```

```

width      : 248px;
height     : 32px;
background : blue;
}
.spa-shell-main {
  top      : 40px;
  left     : 0;
  bottom   : 40px;
  right    : 0;
}
.spa-shell-main-content, ← Определяем общие правила CSS.
.spa-shell-main-nav {
  top      : 0;
  bottom   : 0;
}
.spa-shell-main-nav {
  width     : 250px;
  background : #eee;
}
.spa-x-closed .spa-shell-main-nav { ←
  width : 0;
}
.spa-shell-main-content {
  left      : 250px;
  right     : 0;
  background : #ddd;
}
.spa-x-closed .spa-shell-main-content { ←
  left : 0;
}
.spa-shell-foot {
  bottom : 0;
  left   : 0;
  right  : 0;
  height : 40px;
}
.spa-shell-chat {
  bottom : 0;
  right  : 0;
  width  : 300px;
  height : 15px;
  background : red;
  z-index : 1;
}
.spa-shell-modal {
  margin-top : -200px;
  margin-left : -200px;
  top        : 50%;
  left       : 50%;
  width      : 400px;

```

Используем родительские классы, чтобы оказать влияние на дочерние элементы. Пожалуй, это одно из самых мощных средств CSS, применяемое недостаточно часто.

Производные селекторы формируются с отступом и размещаются сразу под родительским селектором. Производными мы называем селекторы, смысл которых неотделим от родителей.

```

height      : 400px;
background  : #fff;
border-radius : 3px;
z-index     : 2;
}

```

Имена всех селекторов начинаются префиксом `spa-shell-`. У такого соглашения есть несколько плюсов:

- видно, что эти классы управляются модулем Shell (`spa/js/spa.shell.js`);
- предотвращаются конфликты со сторонними скриптами и другими нашими модулями;
- во время отладки и инспектирования HTML-кода документа мы сразу видим, какие элементы порождаются и управляются модулем Shell.

Благодаря этим преимуществам мы предохраняем себя от погружения в разверстый пылающий ад имен селекторов CSS. Всякий, кому доводилось сопровождать даже сравнительно небольшие таблицы стилей, знает, о чем мы говорим.

3.4.4. Настройка приложения для использования Shell

Теперь изменим корневой модуль (`spa/js/spa.js`), так чтобы он не тупо копировал строку «hello world» в DOM, а использовал наш модуль Shell. Модификация показана **полужирным** шрифтом.

```

/*
 * spa.js
 * Модуль с корневым пространством имен
 */
...
/*global $, spa */

var spa = (function () {
  var initModule = function ( $container ) {
    spa.shell.initModule( $container );
  };

  return { initModule: initModule };
})();

```

Теперь, открыв файл `spa/spa.html` в браузере, мы увидим картину, изображенную на рис. 3.6. С помощью инструментов разработчика в Chrome мы можем убедиться, что документ, сгенерированный нашим SPA, соответствует макету (`spa/layout.html`).

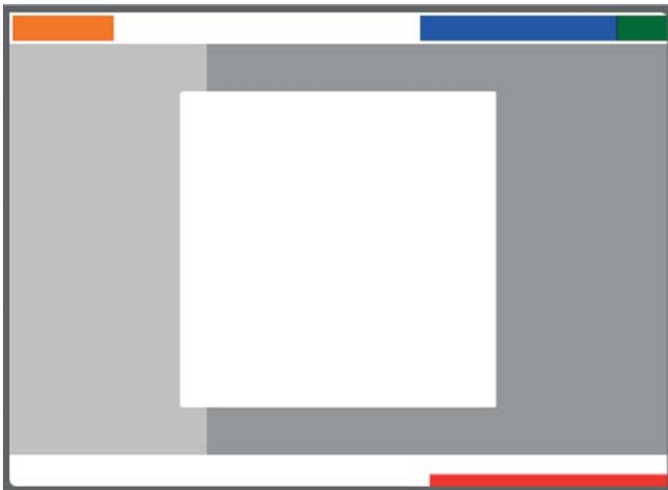


Рис. 3.6 ❖ Дежавю – spa/spa.html

Заложив фундамент, мы теперь заставим Shell управлять функциональными контейнерами. Самое время сделать перерыв, потому что следующий раздел будет весьма насыщенным.

3.5. Управление функциональными контейнерами

Модуль Shell занимается отрисовкой и управлением *функциональными контейнерами*. Это контейнеры «верхнего уровня» – обычно DIV’ы, – в которых находится содержимое, составляющее функциональность страницы. Shell инициализирует все функциональные модули приложения и координирует их работу. Кроме того, Shell указывает функциональным модулям, когда создавать и изменять содержимое функциональных контейнеров. Теме функциональных модулей будет посвящена глава 4.

В этом разделе мы сначала напишем метод, который будет сворачивать и раскрывать функциональный контейнер всплывающего чата. Затем мы напишем обработчик события нажатия, который позволит пользователю в любой момент свернуть или раскрыть окно чата. Далее мы проверим работоспособность того, что сделали, и поговорим о следующей важной теме – управлении состоянием страницы с помощью фрагмента URI-адреса – части, следующей за знаком решетки.

3.5.1. Метод сворачивания и раскрытия окна чата

Мы не будем предъявлять чрезмерных требований к функциональности чата. Он должен быть достаточно качественным для включения в промышленное приложение, но никакая экстравагантность нам не нужна. Вот чего мы хотим достичь.

1. Дать разработчику возможность настраивать скорость анимации и высоту окна чата.
2. Создать единый метод сворачивания и раскрытия окна чата.
3. Избежать гонки – ситуации, когда окно чата одновременно сворачивается и раскрывается.
4. Дать разработчику возможность задавать необязательную функцию обратного вызова, которая вызывается по завершении анимации окна чата.
5. Написать тест, проверяющий правильность работы чата.

Изменим модуль Shell в соответствии с этими требованиями, как показано в листинге 3.11¹. Изменения выделены **полужирным** шрифтом. Не пропускайте аннотаций – в них содержатся замечания о связи изменений с требованиями.

Листинг 3.11 ❖ Модуль Shell, в который добавлены средства сворачивания и раскрытия окна чата, – `spa/js/spa.shell.js`

```
...
spa.shell = (function () {
  //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  var
    configMap = {
      main_html : String()
      ...
      chat_extend_time      : 1000,
      chat_retract_time    : 300,
      chat_extend_height    : 450,
      chat_retract_height  : 15
    },
    stateMap = { $container : null },
    jqueryMap = {},

    setJqueryMap, toggleChat, initModule;
  //----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

  //----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
  //----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----
```

Сохраняем время сворачивания и раскрытия, а также высоту в обоих состояниях в конфигурационном хэше модуля в согласии с требованием 1 «Дать разработчику возможность настраивать скорость анимации и высоту окна чата».

Добавляем метод `toggleChat` в список переменных в области видимости модуля.

¹ Самое время поблагодарить небесные светила за библиотеку jQuery, потому что без нее сделать это было бы куда труднее.

```

//----- НАЧАЛО МЕТОДОВ DOM -----
// Начало метода DOM /setJqueryMap/
setJqueryMap = function () {
    var $container = stateMap.$container;

    jqueryMap = {
        $container : $container,
        $chat : $container.find( '.spa-shell-chat' )
    };
};
// Конец метода DOM /setJqueryMap/

// Начало метода DOM /toggleChat/
// Назначение : свернуть или раскрыть окно чата
// Аргументы :
// * do_extend - если true, раскрыть окно; если false - свернуть
// * callback - необязательная функция, которая вызывается в конце
// * анимации
// Параметры :
// * chat_extend_time, chat_retract_time
// * chat_extend_height, chat_retract_height
// Возвращает : булево значение
// * true - анимация окна чата начата
// * false - анимация окна чата не начата
//
toggleChat = function ( do_extend, callback ) {
    var
        px_chat_ht = jqueryMap.$chat.height(),
        is_open = px_chat_ht === configMap.chat_extend_height,
        is_closed = px_chat_ht === configMap.chat_retract_height,
        is_sliding = ! is_open && ! is_closed;

    // во избежание гонки
    if ( is_sliding ){ return false; }

    // Начало раскрытия окна чата
    if ( do_extend ) {
        jqueryMap.$chat.animate(
            { height : configMap.chat_extend_height },
            configMap.chat_extend_time,
            function () {
                if ( callback ){ callback( jqueryMap.$chat ); }
            }
        );
        return true;
    }
};
// Конец раскрытия окна чата

// Начало сворачивания окна чата
jqueryMap.$chat.animate(

```

Кэшируем в jqueryMap коллекцию jQuery, содержащую окно чата.

Добавляем метод toggleChat с согласии с требованием 2 «Создать единый метод сворачивания и раскрытия окна чата».

Предотвращаем гонку, отказываясь начинать операцию, если окно чата уже находится в процессе анимации, в согласии с требованием 3 «Избежать гонки – ситуации, когда окно чата одновременно сворачивается и раскрывается».

Вызываем переданную функцию по завершении анимации в согласии с требованием 4 «Дать разработчику возможность задавать необязательную функцию обратного вызова, которая вызывается по завершении анимации окна чата».

```

    { height : configMap.chat_retract_height },
    configMap.chat_retract_time,
    function () {
        if ( callback ){ callback( jqueryMap.$chat ); } ←
    }
);
return true;
// Конец сворачивания окна чата
};
// Конец метода DOM /toggleChat/
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /initModule/
initModule = function ( $container ) {
    // загрузить HTML и кэшировать коллекции jQuery
    stateMap.$container = $container;
    $container.html( configMap.main_html );
    setJqueryMap();

    // тестировать переключение
    setTimeout( function () {toggleChat( true ); }, 3000 );
    setTimeout( function () {toggleChat( false ); }, 8000 );
};
// Конец открытого метода /initModule/

return { initModule : initModule };
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();

```

Раскрываем окно чата через 3 секунды после загрузки страницы и сворачиваем через 8 секунд – в согласии с требованием 5 «Написать тест, проверяющий правильность работы чата».

Если вы прорабатываете примеры, то сначала проверьте код с помощью JSLint, выполнив команду `jslint spa/js/spa.shell.js`, – не должно быть ни ошибок, ни предупреждений. Затем загрузите документ `spa/spa.html` в браузер и убедитесь, что окно чата раскрывается через три секунды и снова сворачивается – через восемь. Научившись анимировать окно чата, мы можем инициировать это действие по щелчку мыши.

3.5.2. Добавление обработчика события щелчка мышью по окну чата

Как правило, пользователи ожидают, что щелчок по окну чата приведет к его сворачиванию или раскрытию, потому что это общепринятое соглашение. Приведем список требований:

1. Задать всплывающую подсказку, сообщающую пользователю, что делать, например: «Щелкните, чтобы свернуть».
2. Добавить обработчик события щелчка, вызывающий метод `toggleChat`.
3. Связать обработчик события щелчка с событием jQuery.

Изменим модуль `Shell` в соответствии с этими требованиями, как показано в листинге 3.13. Изменения снова выделены **полужирным** шрифтом, а в аннотациях приведены замечания о связи изменений с требованиями.

Листинг 3.12 ❖ Модуль `Shell`, в который добавлена обработка события щелчка мышью по окну чата, – `spa/js/spa.shell.js`

```
...
spa.shell = (function () {
    //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
    var
        configMap = {
            ...
            chat_retract_height : 15,
            chat_extended_title : 'Щелкните, чтобы свернуть',
            chat_retracted_title : 'Щелкните, чтобы раскрыть'
        },
        stateMap = {
            $container : null,
            is_chat_retracted : true
        },
        jqueryMap = {},

        setJqueryMap, toggleChat, onClickChat, initModule;

    //----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
    ...
    //----- НАЧАЛО МЕТОДОВ DOM -----
    // Начало метода DOM /setJqueryMap/
    ...
    // Конец метода DOM /setJqueryMap/

    // Начало метода DOM /toggleChat/
    // Назначение : свернуть или раскрыть окно чата
    ...
    // Состояние : устанавливает stateMap.is_chat_retracted
    // * true - окно свернуто
    // * false - окно раскрыто
    //
    toggleChat = function ( do_extend, callback ) {
        var
            px_chat_ht = jqueryMap.$chat.height(),
            is_open = px_chat_ht === configMap.chat_extend_height,
            is_closed = px_chat_ht === configMap.chat_retract_height,
```

Добавляем в `configMap` тексты всплывающих подсказок в каждом состоянии – в согласии с требованием 1 «Задать всплывающую подсказку, сообщающую пользователю, что делать...».

Добавляем `is_chat_retracted` в `stateMap`. Считается хорошим тоном перечислять все ключи в `stateMap`, чтобы их было легко найти и проинспектировать. Этот ключ используется в методе `toggleChat`.

Добавляем `onClickChat` в список имен функций в области видимости модуля.

Изменяем документацию метода `toggleChat`, отразив тот факт, что он устанавливает переменную `stateMap.is_chat_retracted`.

```

is_sliding = ! is_open && ! is_closed;

// во избежание гонки
if ( is_sliding ){ return false; }

// Начало раскрытия окна чата
if ( do_extend ) {
    jqueryMap.$chat.animate(
        { height : configMap.chat_extend_height },
        configMap.chat_extend_time,
        function () {
            jqueryMap.$chat.attr(
                'title', configMap.chat_extended_title
            );
            stateMap.is_chat_retracted = false;
            if ( callback ){ callback( jqueryMap.$chat ); }
        }
    );
    return true;
}
// Конец раскрытия окна чата

// Начало сворачивания окна чата
jqueryMap.$chat.animate(
    { height : configMap.chat_retract_height },
    configMap.chat_retract_time,
    function () {
        jqueryMap.$chat.attr(
            'title', configMap.chat_retracted_title
        );
        stateMap.is_chat_retracted = true;
        if ( callback ){ callback( jqueryMap.$chat ); }
    }
);
return true;
// Конец сворачивания окна чата
};
// Конец метода DOM /toggleChat/
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
onClickChat = function ( event ) { ←
    toggleChat( stateMap.is_chat_retracted );
    return false;
};
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /initModule/
initModule = function ( $container ) {

```

Изменяем метод toggle-Chat, так чтобы он управлял показом всплывающего текста и устанавливал переменную stateMap.is_chat_retracted – в согласии с требованием 1 «Задать всплывающую подсказку, сообщающую пользователю, что делать...».

Добавляем обработчик события onClickChat в согласии с требованием 2 «Добавить обработчик события щелчка, вызывающий метод toggleChat».

```

// загрузить HTML и кэшировать коллекции jQuery
stateMap.$container = $container;
$container.html( configMap.main_html );
setJqueryMap();

// инициализировать окно чата и привязать обработчик щелчка
stateMap.is_chat_retracted = true;
jqueryMap.$chat
    .attr( 'title', configMap.chat_retracted_title )
    .click( onClickChat );
};
// Конец открытого метода /initModule/

return { initModule : initModule };
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();

```

Инициализируем обработчик события, устанавливая значение переменной `stateMap.is_chat_retracted` и всплывающий текст. Затем связываем обработчик с событием щелчка в соответствии с требованием 3 «Связать обработчик события щелчка с событием jQuery».

Команда `jslint spa/js/spa.shell.js` снова не должна напечатать ни ошибок, ни предупреждений.

У обработчиков событий в jQuery есть одна особенность, которую мы считаем очень важной и предлагаем запомнить: возвращаемое значение интерпретируется jQuery как признак продолжения или завершения обработки. Обычно мы возвращаем `false`, и это означает следующее:

- отменить действие по умолчанию, например следование по ссылке или выделение текста. Того же эффекта можно достичь, вызвав внутри обработчика события метод `event.preventDefault()`;
- запретить возбуждение того же события в родительском элементе DOM (это поведение часто называют *всплытием*). Того же эффекта можно достичь, вызвав внутри обработчика события метод `event.stopPropagation()`;
- завершить обработку события. Если с элементом, по которому щелкнули мышью, связаны еще какие-то обработчики этого события, будет вызван следующий в очереди (если мы не хотим, чтобы вызывались последующие обработчики, то можем выполнить метод `event.preventDefaultPropagation()`).

Обычно нам именно это и нужно от обработчика события. Но вскоре мы напишем обработчики, в которых такое поведение нежелательно. В этом случае обработчик должен вернуть значение `true`.

Модуль Shell не обязан обрабатывать щелчок мышью. Он мог бы вместо этого предоставить возможность манипулировать окном чата

в функции обратного вызова модуля чата – и мы рекомендуем именно такой способ. Но поскольку этот модуль еще не написан, мы обрабатываем событие щелчка прямо в Shell.

Теперь добавим изюминку к стилям Shell. Изменения показаны в листинге 3.13.

Листинг 3.13 ❖ Добавление изюминки в стили модуля Shell – spa/css/spa.shell.css

```
...
.spa-shell-foot {
    ...
}
.spa-shell-chat {
    bottom      : 0;
    right       : 0;
    width       : 300px;
    height      : 15px;
    cursor      : pointer;
    background  : red;
    border-radius : 5px 0 0 0;
    z-index     : 1;
}
.spa-shell-chat:hover {
    background : #a00;
}
.spa-shell-modal { ... }
...
```

Когда курсор мыши наведен на окно чата, он принимает форму указателя. Тем самым мы сообщаем пользователю, что при щелчке мышью что-то произойдет.

Скругляем угол, чтобы окно чата выглядело симпатичнее.

Когда курсор мыши наведен на окно чата, цвет последнего изменяется. Это дополнительная подсказка пользователю о том, что за щелчком последует какое-то действие.

Перезагрузив документ spa/spa.html, мы можем щелкнуть по окну чата и наблюдать, как оно раскрывается (рис. 3.7).

Окно чата раскрывается гораздо медленнее, чем сворачивается. Скорость анимации можно изменить с помощью конфигурационных параметров модуля Shell (spa/js/spa.shell.js), например:

```
...
configMap = {
    main_html : String()
    ...
    chat_extend_time : 250,
    chat_retract_time : 300,
    ...
},
...
```

В следующем разделе мы научим приложение лучше управлять своим состоянием. Когда мы закончим, все средства браузера, имеющие отношение к истории, – закладки, кнопки «Вперед» и «Назад» – будут работать для окна чата так, как ожидает пользователь.

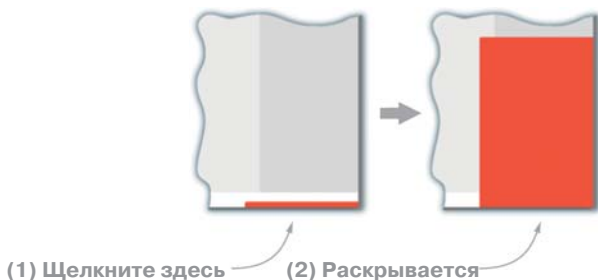


Рис. 3.7 ❖ Раскрытие окна чата – `spa/spa.html`

3.6. Управление состоянием приложения

В информатике *состоянием* называется уникальное сочетание данных в приложении. Персональные и веб-приложения обычно стремятся сохранять какое-то состояние между сеансами. Например, если сохранить документ в текстовом процессоре, а затем снова открыть его, то документ восстанавливается в том виде, в котором был сохранен. Приложение может также сохранять размер окна, пользовательские настройки, положение курсора и текущую страницу. Вот и наше SPA должно сохранять состояние, потому что пользователи браузеров привыкли к определенному поведению.

3.6.1. Какого поведения ожидает пользователь браузера?

В части сохраняемого состояния персональные и веб-приложения сильно различаются. Персональное приложение может опустить кнопку «Предыдущее», если оно не поддерживает функцию «вернуться». Но в случае веб-приложения кнопка браузера «Назад» – одна из самых востребованных, кстати, – уже торчит перед глазами пользователя и умоляет нажать на нее. Убрать ее невозможно.

То же самое относится к кнопке «Вперед», кнопке «Закладки» и истории просмотра. Пользователь ожидает, что эти элементы управления работают. А если это не так, пользователь начнет брюзжать, и нашему приложению никогда не выиграть премию Вебби. В табл. 3.1 показано примерное соответствие между элементами управления историей в настольных и веб-приложениях.

Поскольку мы вознамерились завоевать премию Вебби, то должны обеспечить работу элементов управления историей в соответствии с ожиданиями пользователей. Обсудим, как этого добиться.

Таблица 3.1. Элементы управления в персональных и веб-приложениях

Элемент управления в браузере	Элемент управления в персональном приложении	Примечания
Кнопка «Назад»	Отмена	Вернуться в предыдущее состояние
Кнопка «Вперед»	Повтор	Восстановить состояние, предшествующее последней команде «отмена» или «назад»
Закладка	Сохранить как	Сохранить состояние приложения для использования или ссылки в будущем
История просмотра	История отмены	Показать шаги в последовательности операций отмены/повтора

3.6.2. Стратегия работы с элементами управления историей

Оптимальная стратегия, обеспечивающая работу с элементами управления историей, должна отвечать следующим требованиям.

1. Элементы управления историей должны работать, как ожидает пользователь (см. табл. 3.1).
2. Разработка механизма поддержки не должна обходиться слишком дорого. Она не должна требовать существенно больше времени или быть существенно сложнее, чем разработка без этих элементов.
3. Качество приложения не должно снижаться. Время реакции на действия пользователя не должно увеличиться, а пользовательский интерфейс не должен стать сложнее.

Возьмем такой пример использования чата и последующего взаимодействия с пользователем.

1. Сюзанна заходит в SPA и щелкает по окну чата, чтобы открыть чат.
2. Она ставит закладку на SPA и уходит на другой сайт.
3. Позже она решает вернуться к нашему приложению и нажимает на свою закладку.

Рассмотрим три стратегии, позволяющие обеспечить Сюзанне привычное поведение закладок. Запоминать их необязательно, мы просто хотим сравнить достоинства¹.

¹ Есть и другие стратегии, например использование сохраняемых куков или IFRAME, но, откровенно говоря, они слишком ограничены и запутанны, так что не заслуживают рассмотрения.

Стратегия 1. После щелчка мышью обработчик события сразу вызывает функцию `toggleChat`, игнорируя URI-адрес. Вернувшись к своей закладке, Сюзанна обнаружит окно чата в положении по умолчанию, то есть свернутым. Сюзанна недовольна, так как закладки работают неожиданно. Разработчик Джеймс тоже недоволен, потому что менеджер продукта счел, что приложение неудобно для пользователей, и подверг Джеймса разносу.

Стратегия 2. После щелчка мышью обработчик события сразу вызывает функцию `toggleChat`, а затем модифицирует URI-адрес, чтобы отразить состояние. Когда Сюзанна возвращается к своей закладке, приложение анализирует параметр в URI и действует соответственно. Сюзанна довольна. Но Джеймс недоволен, потому что теперь он должен поддерживать два условия, которые открывают окно чата: событие щелчка во время выполнения и параметр в URI во время загрузки. Менеджер продукта тоже недоволен, потому что наличие двух путей замедляет разработку и может привести к ошибкам и несогласованности.

Стратегия 3. После щелчка мышью обработчик события изменяет URI, а затем возвращает управление. Обработчик события `hashchange` в модуле Shell обнаруживает изменение и передает управление функции `toggleChat`. Когда Сюзанна возвращается к своей закладке, URI-адрес разбирается той же самой функцией, и окно чата восстанавливается в открытом состоянии. Сюзанна довольна, так как закладки работают ожидаемо. Джеймс тоже доволен, потому что для реализации всех допускающих сохранение в закладках состояний используется единый подход. Доволен и менеджер продукта, поскольку разработка ведется быстро и количество ошибок не увеличивается.

Мы предпочитаем *стратегию 3*, потому что она поддерживает все элементы управления историей (требование 1), минимизирует проблемы на этапе разработки (требование 2) и обеспечивает качество приложения, так как затрагивает только те части страницы, которые должны быть изменены при использовании элемента управления историей (требование 3). Поскольку в этом решении состояние страницы определяется URI-адресом, мы называем его паттерном якорного интерфейса (рис. 3.8).

Мы вернемся к этому паттерну в главе 4. А теперь реализуем выбранную стратегию.

3.6.3. Изменение якоря при возникновении события истории

Компонент «якорь» в URI-адресе говорит браузеру, какую часть страницы показывать. По-другому его называют *компонент закладки* или

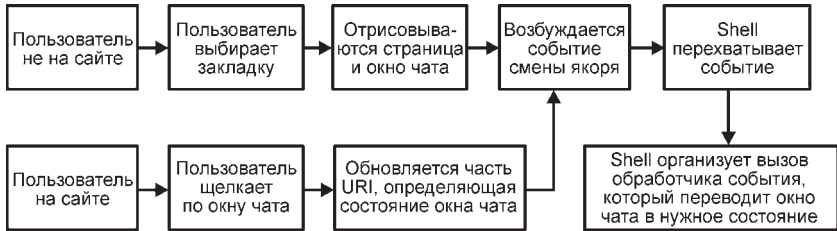


Рис. 3.8 ❖ Паттерн якорного интерфейса

#-фрагмент. Якорь начинается со знака # и в следующем примере выделен **полужирным** шрифтом:

`http://localhost/spa.html#!chat=open`

Традиционно веб-разработчики использовали механизм якорей для облегчения перехода между разделами длинного документа. Например, в начале страницы может быть размещено оглавление, каждый заголовок в котором является ссылкой на соответствующий раздел документа. А в конце каждого раздела может находиться ссылка «В начало». В блогах и форумах этот механизм применяется сплошь и рядом.

Важная особенность якорей заключается в том, что браузер *не* перегружает страницу при изменении якоря. Якорь – чисто клиентский механизм, что и делает его идеальным средством для хранения состояния приложения. Эта техника используется во многих SPA.

Изменение состояния приложения, которое мы хотим хранить в истории браузера, будем называть *событием истории*. Поскольку мы решили, что открытие или закрытие чата – это событие истории (вы тогда прогуляли), то можем поручить обработчику события щелчка изменить якорь, чтобы отразить смену состояния окна чата. Скучные детали можно возложить на подключаемый к jQuery модуль `uriAnchor`. В листинге 3.14 показано, что нужно доработать в модуле `Shell`, чтобы щелчок мышью изменял URI.

Листинг 3.14 ❖ Подключаемый модуль `uriAnchor` в действии – `spa/js/spa.shell.js`

```

...
//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
onClickChat = function ( event ) {
  if ( toggleChat( stateMap.is_chat_retracted ) ) {
    $.uriAnchor({
      chat : ( stateMap.is_chat_retracted ? 'open' : 'closed' )
    });
  }
  return false;
}

```



```
};  
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----  
...
```

Теперь, щелкнув по окну чата, мы увидим, что якорь в URI-адресе изменился, – но только если функция `toggleChat` завершается успешно и возвращает `true`. Например, открыв, а потом закрыв окно чата, мы увидим такой адрес:

```
http://localhost/spa.html#!chat=closed
```

О восклицательном знаке

Восклицательный знак после символа решетки (!) в этом URI говорит Google и другим поисковым системам, что данный URI можно индексировать для поиска. Подробнее о поисковой оптимизации мы будем говорить в главе 9.

Мы должны сделать так, чтобы при изменении якоря модификации подвергалась только соответствующая часть приложения. Это ускорит работу приложения и позволит избежать раздражающего «мелькания» из-за стирания и повторной отрисовки неизменившихся частей страницы. Предположим, к примеру, что, открыв чат щелчком по его окну, Сюзанна видит тысячу профилей пользователей. При нажатии кнопки «Назад» приложение должно просто свернуть окно чата – перерисовывать профили не нужно.

Решая, заслуживает ли изменение поддержки в истории, мы должны задать себе три вопроса:

- в какой мере пользователю необходимо ставить закладку на только что произошедшее изменение?
- насколько важна пользователю возможность вернуться к состоянию страницы до изменения?
- во что обойдется реализация поддержки?

Хотя благодаря паттерну якорного интерфейса добавочные затраты на поддержку состояния обычно невелики, бывают ситуации, когда она обходится дорого или вообще невозможна. Например, отменить покупку, сделанную в интернет-магазине, нажатием кнопки «Назад» очень трудно. В такой ситуации необходимо вообще запретить занесение новой записи в историю. К счастью, подключаемый модуль `uriAnchor` позволяет это сделать.

3.6.4. Использование якоря для управления состоянием приложения

Мы хотим, чтобы якорь полностью определял состояние приложения, сохраняемое в закладках. Тогда функциональность истории бу-

дет ожидаемой. Следующий псевдокод описывает желательный алгоритм обработки события истории.

- Когда происходит событие истории, изменить якорь в URI-адресе, так чтобы он отражал новое состояние:
 - обработчик, получивший событие, вызывает служебный метод Shell для изменения якоря;
 - после этого обработчик события возвращает управление.
- Обработчик события `hashchange` в модуле Shell замечает, что URI изменился, и выполняет соответствующие действия:
 - сравнивает текущее состояние с тем, которое предлагает новый якорь;
 - на основании результатов сравнения пытается изменить те участки приложения, которые в этом нуждаются;
 - если запрошенные изменения невозможны, оставляет текущее состояние и соответственно восстанавливает якорь.

Наметив программу, преобразуем псевдокод в настоящий код.

Модификация Shell для работы с якорем

Внесем в модуль Shell изменения, необходимые для того, чтобы якорь определял состояние приложения (листинг 3.15). Тут довольно много нового кода, но не пугайтесь – в свое время все получит объяснение.

Листинг 3.15 ❖ Использование якоря для управления состоянием приложения – `spa/js/spa.shell.js`

```
...
spa.shell = (function () {
  //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  var
    configMap = {
      anchor_schema_map : { ←
        chat : { open : true, closed : true } ←
      },
      main_html : String(),
      ...
    },
    stateMap = {
      $container      : null,
      anchor_map : {}, ←
      is_chat_retracted : true
    },
    jqueryMap = {},
    copyAnchorMap, setJqueryMap, toggleChat, ←
    changeAnchorPart, onHashchange,
```

Определяем хэш, который uriAnchor использует для валидации.

Текущие значения якорей сохраняются в хэше stateMap. anchor_map, который является частью состояния модуля.

Объявляем три дополнительных метода: copyAnchorMap, changeAnchorPart и onHashchange.

```

onClickChat, initModule;
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

//----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
// Возвращает копию сохраненного хэша якорей; минимизация издержек ←
copyAnchorMap = function () {
    return $.extend( true, {}, stateMap.anchor_map );
};
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

Используем метод jQuery extend() для копирования
объекта. Это необходимо, потому что в JavaScript все
объекты передаются по ссылке, и правильное копи-
рование объекта – нетривиальная задача.

//----- НАЧАЛО МЕТОДОВ DOM -----
...
// Начало метода DOM /changeAnchorPart/ ←
// Назначение: изменяет якорь в URI-адресе
// Аргументы:
// * arg_map – хэш, описывающий, какую
//   часть якоря мы хотим изменить.
// Возвращает : булево значение
// * true  – якорь в URI обновлен
// * false – не удалось обновить якорь в URI
// Действие:
// Текущая часть якоря сохранена в stateMap.anchor_map.
// Обсуждение кодировки см. в документации по uriAnchor.
// Этот метод
// * Создает копию хэша, вызывая copyAnchorMap().
// * Модифицирует пары ключ-значение с помощью arg_map.
// * Управляет различием между зависимыми и независимыми
//   значениями в кодировке.
// * Пытается изменить URI, используя uriAnchor.
// * Возвращает true в случае успеха и false – в случае ошибки.
//
changeAnchorPart = function ( arg_map ) {
    var
        anchor_map_revise = copyAnchorMap(),
        bool_return = true,
        key_name, key_name_dep;

    // Начало объединения изменений в хэше якорей
    KEYVAL:
    for ( key_name in arg_map ) {
        if ( arg_map.hasOwnProperty( key_name ) ) {

            // пропустить зависимые ключи
            if ( key_name.indexOf( '_' ) === 0 ) { continue KEYVAL; }

            // обновить значение независимого ключа
            anchor_map_revise[key_name] = arg_map[key_name];

```

```

    // обновить соответствующий зависимый ключ
    key_name_dep = '_' + key_name;
    if ( arg_map[key_name_dep] ) {
        anchor_map_revise[key_name_dep] = arg_map[key_name_dep];
    }
    else {
        delete anchor_map_revise[key_name_dep];
        delete anchor_map_revise['_s' + key_name_dep];
    }
}
}
// Конец объединения изменений в хэше якорей

// Начало попытки обновления URI; в случае ошибки ←
// восстановить исходное состояние
try {
    $.uriAnchor.setAnchor( anchor_map_revise );
}
catch ( error ) {
    // восстановить исходное состояние в URI
    $.uriAnchor.setAnchor( stateMap.anchor_map, null, true );
    bool_return = false;
}
// Конец попытки обновления URI... ←

return bool_return;
};
// Конец метода DOM /changeAnchorPart/
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
// Начало обработчика события /onHashchange/ ←
// Назначение: обрабатывает событие hashchange
// Аргументы:
// * event - объект события jquery.
// Параметры: нет
// Возвращает: false
// Действие:
// * Разбирает якорь в URI.
// * Сравнивает предложенное состояние приложения с текущим.
// * Вносит изменения, только если предложенное состояние
// отличается от текущего.
//
onHashchange = function ( event ) {
    var
        anchor_map_previous = copyAnchorMap(),
        anchor_map_proposed,
        _s_chat_previous, _s_chat_proposed,
        s_chat_proposed;

```

Не устанавливаем якорь, если он не соответствует схеме (uriAnchor возбудит исключение). В таком случае возвращаем якорь в исходное состояние.

Добавляем обработчик события изменения якоря в URI – onHashchange. С помощью подключаемого модуля uriAnchor преобразуем якорь в хэш и сравниваем с предыдущим состоянием, чтобы определить требуемое действие. Если предлагаемое изменение якоря недопустимо, восстанавливаем предыдущее состояние якоря.

```
// пытаемся разобрать якорь
try { anchor_map_proposed = $.uriAnchor.makeAnchorMap(); }
catch ( error ) {
    $.uriAnchor.setAnchor( anchor_map_previous, null, true );
    return false;
}
stateMap.anchor_map = anchor_map_proposed;

// вспомогательные переменные
_s_chat_previous = anchor_map_previous._s_chat;
_s_chat_proposed = anchor_map_proposed._s_chat;

// Начало изменения компонента Chat
if ( ! anchor_map_previous
    || _s_chat_previous !== _s_chat_proposed
) {
    s_chat_proposed = anchor_map_proposed.chat;
    switch ( s_chat_proposed ) {
        case 'open' :
            toggleChat( true );
            break;
        case 'closed' :
            toggleChat( false );
            break;
        default :
            toggleChat( false );
            delete anchor_map_proposed.chat;
            $.uriAnchor.setAnchor( anchor_map_proposed, null, true );
    }
}
// Конец изменения компонента Chat

return false;
};
// Конец обработчика события /onHashchange/

// Начало обработчика события /onClickChat/
onClickChat = function ( event ) {
    changeAnchorPart({
        chat: ( stateMap.is_chat_retracted ? 'open' : 'closed' )
    });
    return false;
};
// Конец обработчика события /onClickChat/

//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /initModule/
```

Изменяем обработчик события onClickChat, так чтобы модифицировался только параметр chat в якорь.

```

initModule = function ( $container ) {
    ...
    // настраиваем uriAnchor на использование нашей схемы
    $.uriAnchor.configModule({
        schema_map : configMap.anchor_schema_map
    });
    // Обрабатываем события изменения якоря в URI.
    // Это делается /после/ того, как все функциональные модули
    // сконфигурированы и инициализированы, иначе они будут не готовы
    // возбудить событие, которое используется, чтобы гарантировать
    // учет якоря при загрузке.
    //
    $(window) ←
        .bind( 'hashchange', onHashchange )
        .trigger( 'hashchange' );
    // Привязываем обработчик события hashchange
    // и сразу возбуждаем событие, чтобы модуль учи-
    // тывал закладку на этапе начальной загрузки.
};
// Конец открытого метода /initModule/

return { initModule : initModule };
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
}());

```

После этой модификации все элементы управления историей – кнопки «Вперед» и «Назад», закладки и история просмотра – должны работать, как положено. А якорь «починит себя», если мы попытаемся вручную записать в него неподдерживаемые параметры или значения, – например, попробуем заменить якорь в адресной строке браузера на `#!chat=barney` и нажать **Enter**.

Итак, элементы управления историей работают, и мы можем обсудить, как использовать якорь для управления состоянием приложения. Начнем с вопроса о том, как подключаемый модуль `uriAnchor` кодирует и декодирует якорь.

Как uriAnchor кодирует и декодирует якорь

Мы используем событие jQuery `hashchange`, чтобы распознать изменение якоря. Состояние приложения кодируется с помощью *независимых* и *зависимых* пар ключ–значение. Рассмотрим следующий пример, в котором якорь выделен полужирным шрифтом:

`http://localhost/spa.html#!chat=profile: on: uid, suzie|status, green`

Здесь *независимым* является ключ `profile`, значение которого равно `on`. Ключи, определяющие состояние профиля, *зависимы*, они перечисляются после двоеточия. Это ключ `uid` со значением `suzie` и ключ `status` со значением `green`.

Подключаемый модуль `uriAnchor`, `js/jq/jquery.uriAnchor-1.1.3.js`, берет на себя заботу о кодировании зависимых и независимых ключей. С помощью метода `$.uriAnchor.setAnchor()` мы можем построить URI, соответствующий примеру выше:

```
var anchorMap = {
  profile : 'on',
  _profile : {
    uid : 'suzie',
    status : 'green'
  }
};
$.uriAnchor.setAnchor( anchorMap );
```

Для разбора якоря и преобразования его в хэш служит метод `makeAnchorMap`:

```
var anchorMap = $.uriAnchor.makeAnchorMap();
console.log( anchorMap );

// Если URI с якорем имеет вид
// http://localhost/spa.html#!chat=profile:on:uid,suzie|status:green
//
// то console.log( anchorMap ) покажет следующее:
//
// { profile : 'on',
//   _profile : {
//     uid : 'suzie',
//     status : 'green'
//   }
// };
//
```

Надеемся, вы стали лучше понимать, как модуль `uriAnchor` используется для кодирования и декодирования состояния приложения в яоре URI. Теперь поговорим о том, как использовать якорь для управления состоянием приложения.

Как изменение якоря определяет состояние приложения

Наша стратегия управления историей заключается в том, что обработчик любого события, которое изменяет запоминаемое в закладках состояние, должен делать две вещи:

- 1) изменить якорь;
- 2) быстро вернуть управление.

Мы добавили в модуль `Shell` метод `changeAnchorPart`, который позволяет обновить только часть якоря, гарантируя правильность об-

работки зависимых и независимых ключей и значений. Он инкапсулировал всю логику управления якорем, и *только с его помощью приложение модифицирует якорь*.

Под словами «быстро вернуть управление» мы понимаем, что обязанности этого обработчика события исчерпываются изменением якоря. Он не изменяет элементы на странице. Он не обновляет никаких переменных и флагов. Он не стоит на проходе и не ходит по газонам. Он просто возвращается к событию, в ответ на которое был вызван. Это видно на примере нашего обработчика `onClickChat`:

```
onClickChat = function ( event ) {
  changeAnchorPart({
    chat: ( stateMap.is_chat_retracted ? 'open' : 'closed' )
  });
  return false;
};
```

Этот обработчик вызывает метод `changeAnchorPart`, чтобы изменить параметр `chat` в якорь, а затем сразу возвращается. Изменение якоря порождает событие браузера `hashchange`. Shell обнаруживает это событие и предпринимает действия, зависящие от содержимого якоря. Например, увидев, что значение `chat` изменилось с `opened` на `closed`, Shell свернет окно чата.

Можете считать, что якорь, модифицированный методом `changeAnchorPart`, – это и есть API для работы с состояниями, запоминаемыми в закладках. Красота этого подхода – в том, что совершенно не важно, *почему* был изменен якорь: потому что так пожелало приложение, потому что пользователь нажал на закладку или на одну из кнопок «Вперед» либо «Назад» или потому что URI был введен непосредственно в адресную строку. В любом случае, результат получается один и тот же, и к тому же с помощью единого кода.

3.7. Резюме

Мы завершили реализацию двух основных обязанностей модуля Shell. Мы создали и стилизовали функциональные контейнеры и подготовили каркас для управления состоянием приложения с помощью якоря в URI-адресе. Эти идеи были продемонстрированы на примере обновленного окна чата.

Но рассмотрение Shell еще не закончено, так как осталась третья обязанность: координация работы функциональных модулей. В следующей главе мы покажем, как устроены функциональные модули,

как сконфигурировать и инициализировать их из Shell и как они вызываются. Инкапсуляция конкретных функций в отдельных модулях повышает надежность, удобство сопровождения, масштабируемость и улучшает организацию разработки. Кроме того, стимулируются использование и разработка сторонних модулей. Так что не уходите – сейчас начнется самое интересное.

Глава 4

Добавление функциональных модулей

В этой главе:

- ✧ Определение функциональных модулей и их места в нашей архитектуре.
- ✧ Сравнение функциональных и сторонних модулей.
- ✧ Паттерн проектирования «фрактальный MVC» и его роль в нашей архитектуре.
- ✧ Структура файлов и каталогов для функциональных модулей.
- ✧ Определение и реализация API функциональных модулей.
- ✧ Реализация часто требующихся возможностей функциональных модулей.

К этому моменту вы должны были проработать главы 1–3. Вы также должны были создать описанный в главе 3 проект со всеми файлами, поскольку мы собираемся и дальше работать с ним. Мы рекомендуем целиком скопировать созданное в главе 3 дерево каталогов со всеми файлами в новый каталог «chapter_4» и уже там вносить изменения.

Функциональный модуль предоставляет SPA четко определенную и достаточно узкую функциональность. В этой главе мы перенесем функциональность всплывающего чата, разработанную в главе 3, в функциональный модуль, попутно усовершенствовав ее. Помимо всплывающего чата, в качестве примеров функциональных модулей можно привести средство просмотра изображений, панель управления учетной записью или мастерскую, в которой пользователь может собирать графические объекты.

Мы проектируем функциональные модули, так чтобы их взаимодействие с приложением осуществлялось так же, как со сторонними модулями, – через четко определенный API со строгой изоляцией. Это позволяет выпускать новые версии раньше и с высоким качеством, потому что мы можем сосредоточиться на создании модулей, увеличивающих ценность продукта, оставляя разработку вспомогательных

модулей сторонним организациям. Эта стратегия обеспечивает также понятный путь улучшения, так как мы можем избирательно заменять сторонние модули более качественными, когда есть время и ресурсы. Дополнительный бонус – возможность использовать одни и те же модули в разных проектах.

4.1. Стратегия функциональных модулей

Рассмотренный в главе 3 модуль Shell отвечает за задачи, относящиеся к приложению в целом, например управление куками и якорями в URI-адресах. Более специфические задачи он поручает тщательно изолированным функциональным модулям. У каждого такого модуля есть собственное представление и контроллер и своя часть модели, разделяемая с Shell. Общая архитектура показана на рис. 4.1¹.

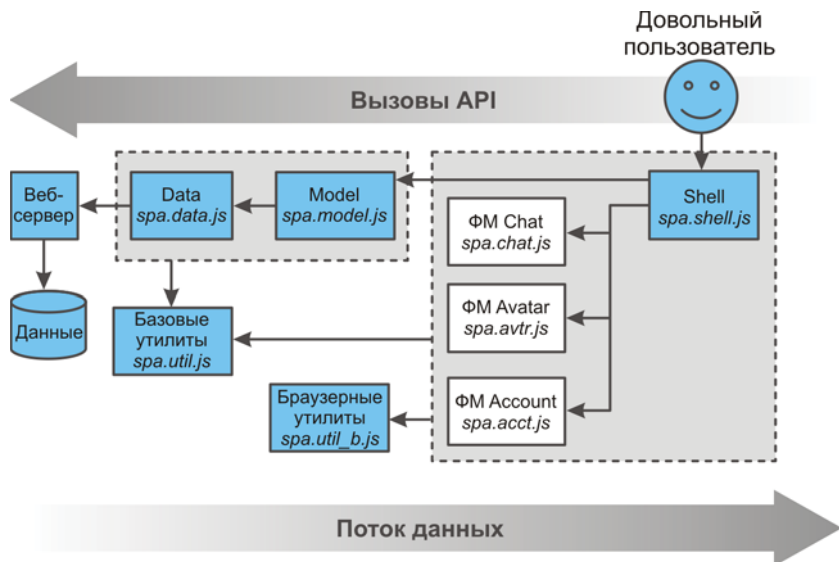


Рис. 4.1 ❖ Функциональные модули в архитектуре SPA (показаны белыми блоками)

Вот несколько примеров функциональных модулей: `spa.wb.js` для создания эскизов в мастерской, `spa.acct.js` для управления учетной записью, в частности входа и выхода, `spa.chat.js` для интерфейса с ча-

¹ Эта диаграмма висит на стене над столом автора.

том. Поскольку мы так успешно занялись чатом, то сейчас сосредоточимся именно на этом модуле.

4.1.1. Сравнение со сторонними модулями

Функциональные модули во многом напоминают сторонние модули, которые предоставляют разнообразную функциональность современным сайтам¹. Примеры сторонних модулей: добавление комментариев в блог (*DisQus* или *LiveFyre*), реклама (*DoubleClick* или *ValueClick*), аналитика (*Google* или *Overture*), обобществление (*AddThis* или *Share-This*) и социальные сервисы (кнопки *Facebook* «Мне нравится» или *Google* «+1»). Они пользуются огромной популярностью, потому что позволяют владельцам сайтов добавлять высококачественную функциональность по цене, составляющей лишь малую толику от той, что пришлось бы заплатить в случае разработки своими силами². Обычно сторонний модуль делается частью сайта путем включения тега script в состав статической веб-страницы или добавления вызова функции в SPA. Многие функции сайтов были бы невозможны без сторонних модулей, поскольку стоимость разработки соответствующей функциональности непомерно высока.

Хорошо написанные сторонние модули обладают следующими общими характеристиками:

- *отрисовываются в отдельном контейнере*, который либо резервируется автором сайта, либо добавляется ими в конец документа самостоятельно;
- *предоставляют четко определенный API* для управления своим поведением;
- *не засоряют объемлющую страницу*, заботясь об изоляции своего JavaScript-кода, данных и CSS-стилей.

У сторонних модулей есть и недостатки. Основная проблема заключается в том, что у «сторонней организации» могут быть собственные деловые цели, вступающие в противоречие с вашими. Это может проявляться разными способами.

- *Мы зависим от чужого кода и сервисов*. Если сторонняя организация прекратит свою деятельность, то и сервиса не будет. Если

¹ Подробнее о сторонних модулях и их создании можно прочитать в книге Ben Vinegar, Anton Kovalyov «Third-Party JavaScript» (Manning, 2012).

² Трудно точно изменить популярность сторонних модулей, но практически невозможно найти коммерческий сайт, на котором не было бы хотя бы одного из них. Например, на момент написания этой книги мы насчитали 16 крупных сторонних модулей на сайте TechCrunch.com, из них, по меньшей мере, пять аналитических служб, а тегов script было аж 53 штуки – впечатляет!

они напортачат в очередной версии, то наш сайт может перестать работать. Как это ни печально, такое бывает куда чаще, чем хотелось бы.

- *Сторонний модуль нередко работает медленнее*, чем специализированный, из-за перегрузки сервера или чрезмерного количества функций. Если какой-то один сторонний модуль тормозит, то может замедлиться все наше приложение.
- *Вопрос конфиденциальности тоже не праздный*, потому что у каждого стороннего модуля есть свои «Условия предоставления услуги», в которых почти всегда за разработчиком оставлено право вносить изменения без предупреждения.
- *Функции стороннего модуля не всегда органично интегрируются* из-за различий в форматах данных, стилях или недостаточной гибкости.
- *Межфункциональная коммуникация* может оказаться труднодостижимой или вовсе невозможной, если нам не удастся интегрировать сторонние данные с нашим SPA.
- *Адаптация модуля к нашим нуждам* может оказаться трудной или невозможной.

Наши функциональные модули обладают положительными свойствами сторонних модулей, но в силу отсутствия сторонней организации лишены их недостатков. Это означает, что для любой функции Shell предоставляет контейнер, который функциональный модуль заполняет и контролирует, как показано на рис. 4.2. Функциональный модуль предоставляет модулю Shell согласованный API для конфигурирования,

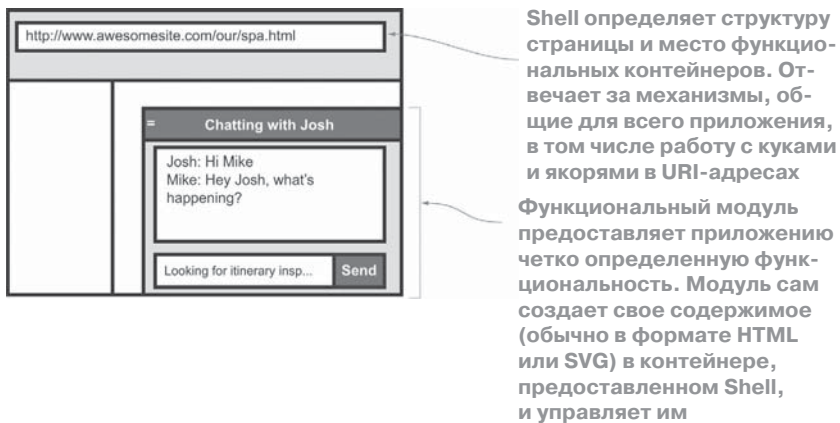


Рис. 4.2 ❖ Обязанности Shell и функциональных модулей

инициализации и использования. Его функциональность изолирована от прочей функциональности благодаря уникальным и скоординированным пространствам имен JavaScript и CSS и запрету на любые внешние вызовы, кроме обращений к разделяемым библиотекам.

Разработка функциональных модулей, так же как сторонних, позволяет нам воспользоваться преимуществами написания JavaScript в этом стиле:

- *команды могут работать эффективнее*, потому что модули можно распределить между разными разработчиками. Давайте говорить честно: если вы работаете в команде, то несторонними для вас будут только те модули, за которые вы лично отвечаете. Все прочие члены команды должны знать лишь внешний API модуля;
- *производительность приложения обычно не страдает*, потому что модули управляют лишь той его частью, за которую отвечают, и «заточены» под наши задачи, без чрезмерного изобилия неиспользуемых или ненужных возможностей;
- *сопровождение и повторное использование кода существенно упрощаются*, потому что модули тщательно изолированы. Многие изощренные модули, подключаемые к jQuery, например выбор даты, – это, по существу, сторонние приложения. Подумайте, насколько проще использовать готовый модуль выбора даты, чем писать свой собственный.

И разумеется, у методики разработки функциональных модулей по аналогии со сторонними есть еще одно *гигантское* преимущество: мы вполне можем начать с использования в нашем приложении сторонних модулей для второстепенной функциональности, а затем избирательно заменять их – когда позволят время и ресурсы – собственными функциональными модулями, которые могут быть лучше интегрированы, быстрее, лучше инкапсулированы или все перечисленное вместе.

4.1.2. Функциональные модули и паттерн «фрактальный MVC»

Многие веб-разработчики знакомы с паттерном проектирования *Модель–Представление–Контроллер (MVC)*, поскольку он лежит в основе многих каркасов, например Ruby on Rails, Django (Python), Catalyst (Perl), Spring MVC (Java) и MicroMVC (PHP). И так как значительная часть читателей в курсе этого паттерна, объясним, какое

отношение он имеет к нашей архитектуре SPA, а особенно к функциональным модулям.

Напомним, что паттерн MVC используется для разработки приложения. Он состоит из следующих частей:

- *модель*, которая предоставляет данные и бизнес-правила;
- *представление*, которое дает чувственно воспринимаемое (обычно зрительное, но часто также звуковое) представление данных модели;
- *контроллер*, который преобразует запросы пользователя в команды, обновляющие модель и (или) представление приложения.

Разработчики, знакомые с каким-нибудь веб-каркасом на основе MVC, не должны встретить затруднений при чтении этой главы. Важнейшее различие между традиционным взглядом на паттерн MVC и нашей архитектурой SPA заключается в следующем:

- задача SPA – переместить как можно большую часть приложения в браузер;
- паттерн MVC у нас повторяется на разных уровнях, как фрактал.

Напомним, что *фракталом* называется структура, обладающая свойством самоподобия на любом уровне. Простой пример фрактала изображен на рис. 4.3 – на расстоянии мы видим общую структуру, а приглядевшись, обнаруживаем, что она повторяется на более мелких уровнях детализации.

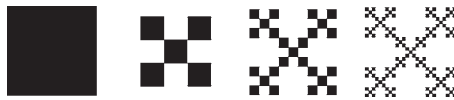


Рис. 4.3 ❖ Квадрат как фрактал

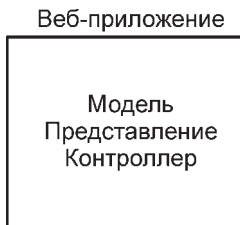


Рис. 4.4 ❖ Взгляд на наше приложение издали

В нашей архитектуре SPA применяется паттерн MVC, повторяющийся на нескольких уровнях, поэтому мы называем его «фрактальным MVC», или FMVC. Эта концепция не нова, под тем же названием ее обсуждают по меньшей мере десять лет. Какая часть фрактала видна, зависит от того, откуда смотреть. Рассматривая веб-приложение на расстоянии, как на рис. 4.4, мы видим единственный паттерн MVC – контроллер обрабатывает URI и данные, введенные

пользователем, взаимодействует с моделью и выводит представление в браузере.

При небольшом увеличении, как на рис. 4.5, мы замечаем, что веб-приложение состоит из двух частей: серверной, где паттерн MVC применяется для отправки данных клиенту, и SPA, где MVC лежит в основе просмотра браузерной модели и взаимодействия с ней пользователя. Серверная модель включает данные, хранящиеся в базе, представление в этом случае – данные, отправляемые браузеру, а контроллер – код, который координирует управление данными и взаимодействие с браузером. На стороне клиента модель включает данные, полученные от сервера, представление – это пользовательский интерфейс, а контроллер координирует клиентские данные и интерфейс.

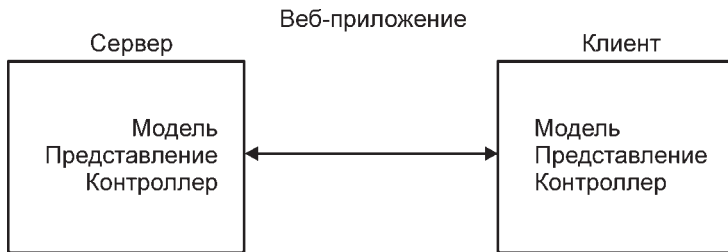


Рис. 4.5 ❖ Взгляд на приложение с более близкого расстояния

При еще большем увеличении, как на рис. 4.6, мы различим дополнительные паттерны MVC. Например, в серверном приложении MVC используется для реализации API работы с данными поверх протокола HTTP. У базы данных, с которой работает сервер, есть собственный паттерн MVC. Клиентская часть нашего приложения использует MVC и при этом Shell вызывает подчиненные функциональные модули, которые и сами построены на основе того же паттерна.

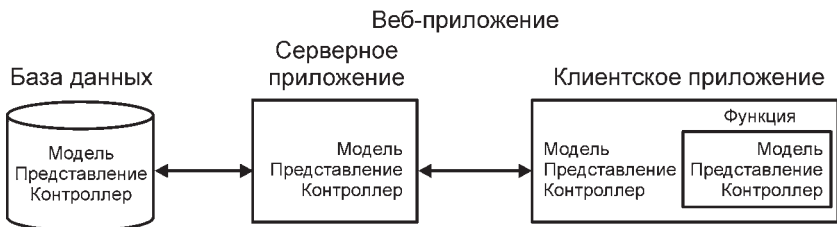


Рис. 4.6 ❖ Взгляд на приложение вблизи

Почти все современные сайты устроены таким образом, даже если разработчики не осознают этого. Например, добавляя в свой блог функцию комментирования из модуля *DisQus* или *LiveFyre* (и вообще практически любой сторонний модуль), разработчик добавляет еще один паттерн MVC.

Наша архитектура SPA впитала паттерн фрактального MVC. Иными словами, работа нашего SPA мало зависит от того, интегрируем мы сторонний модуль или функциональный модуль, написанный собственными руками. На рис. 4.7 показано, как паттерн MVC используется в модуле чата.

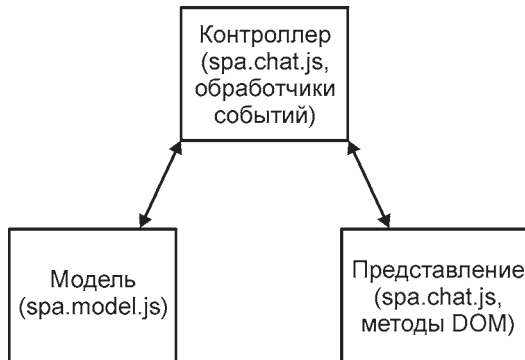


Рис. 4.7 ❖ Паттерн MVC
в функциональном модуле Chat

Мы рассказали о месте функциональных модулей в нашей архитектуре, о том, что они похожи на сторонние модули, и о том, как мы используем паттерн фрактального MVC. В следующем разделе мы применим эти концепции к созданию своего первого функционального модуля.

4.2. Подготовка файлов функционального модуля

Нашим первым функциональным модулем для SPA будет модуль чата, который мы далее будем называть *Chat*. Мы остановились на нем, потому что уже проделали большую работу в главе 3 и потому что преобразование плодов этой работы в функциональный модуль поможет выявить определяющие характеристики последнего.

4.2.1. Планируем структуру каталогов и файлов

Мы рекомендуем скопировать всю созданную в главе 3 структуру каталогов в новый каталог «chapter_4» и там вносить изменения. Структура, на которой мы остановились в главе 3, показана в листинге 4.1:

Листинг 4.1 ❖ Структура каталогов и файлов из главы 3

```
spa
+-- css
| +-- spa.css
| `-- spa.shell.css
+-- js
| +-- jq
| | +-- jquery-1.9.1.js
| | `-- jquery.uriAnchor-1.1.3.js
| +-- spa.js
| `-- spa.shell.js
+-- layout.html
`-- spa.html
```

А вот какие изменения мы собираемся внести:

- создать для Chat таблицу стилей в пространстве имен;
- создать для Chat JavaScript-модуль в пространстве имен;
- создать заглушку для браузерной модели;
- создать служебный модуль, который содержит функции, используемые всеми остальными модулями;
- изменить HTML-документ, включив в него новые файлы;
- удалить файл, в котором мы разрабатывали макет.

После того как мы закончим, дерево файлов и каталогов будет выглядеть, как показано в листинге 4.2. Все файлы, которые нам предстоит создать или модифицировать, выделены **полужирным** шрифтом.

Листинг 4.2 ❖ Пересмотренная структура каталогов и файлов для Chat

```
spa
+-- css
| +-- spa.chat.css ← Добавляем таблицу стилей для Chat.
| +-- spa.css
| `-- spa.shell.css
+-- js
| +-- jq
| | +-- jquery-1.9.1.js
| | `-- jquery.uriAnchor-1.1.3.js
| +-- spa.chat.js ← Добавляем JavaScript для Chat.
| +-- spa.js
| +-- spa.model.js ← Добавляем JavaScript для модели.
| +-- spa.shell.js ← Модифицируем Shell с учетом Chat.
| `-- spa.util.js ← Добавляем новый служебный модуль.
+-- spa.html ← Модифицируем HTML-документ, включая в него новые файлы.
    ← Удаляем файл, в котором вели разработку макета spa/layout.html.
```

Решив, какие файлы нужно добавить и изменить, откроем текстовый редактор и приступим к делу. Мы будем рассматривать файлы в том порядке, в котором описывали их выше.

4.2.2. Создание файлов

Первым рассмотрим файл с таблицей стилей для Chat – `spa/css/spa.chat.css`. Создайте его и заполните, как показано в листинге 4.3. Поначалу это будет *заглушка*¹.

Листинг 4.3. Наша таблица стилей (заглушка) – `spa/css/spa.chat.css`

```
/*
 * spa.chat.css
 * Стили для функционального модуля Chat
 */
```

Далее создадим сам функциональный модуль Chat, файл `spa/js/spa.chat.js`, как показано в листинге 4.4, следуя шаблону модуля из приложения А. Это только первый проход, и мы заполним контейнер окна чата каким-нибудь тривиальным HTML-кодом:

Листинг 4.4 ❖ Наш модуль Chat с ограниченной функциональностью – `spa/js/spa.chat.js`

```
/*
 * spa.chat.js
 * Функциональный модуль Chat для SPA
 */
/*jslint      browser : true,   continue : true,
 devel      : true,   indent : 2,   maxerr : 50,
 newcap     : true,   nomen  : true, plusplus : true,
 regexp     : true,   sloppy : true, vars    : false,
 white      : true
 */

/*global $, spa */

spa.chat = (function () { ← Создаем пространство имен этого модуля, spa.chat.
//----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
var
  configMap = {
    main_html : String()
      + '<div style="padding:1em; color:#fff;">' ←
      + 'Say hello to chat'
      + '</div>',
    ← Сохраняем в configMap HTML-шаблон
    ← выплывающего чата. Можете заменить
    ← неприятное сообщение своим.
```

¹ Заглушкой называется ресурс, который намеренно оставлен незаконченным. Например, в главе 5 мы создадим заглушку модуля данных, которая будет имитировать взаимодействие с сервером.

```

    settable_map : {}
  },
  stateMap = { $container : null },
  jqueryMap = {},
  setJqueryMap, configModule, initModule
;
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

//----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

//----- НАЧАЛО МЕТОДОВ DOM -----
// Начало метода DOM /setJqueryMap/
setJqueryMap = function () {
  var $container = stateMap.$container;
  jqueryMap = { $container : $container };
};
// Конец метода DOM /setJqueryMap/
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /configModule/ ←
// Назначение: настроить допустимые ключи
// Аргументы: хэш настраиваемых ключей и их значений
// * color_name – какой цвет использовать
// Параметры:
// * configMap.settable_map объявляет допустимые ключи
// Возвращает: true
// Исключения: нет
//
configModule = function ( input_map ) {
  spa.util.setConfigMap({
    input_map : input_map,
    settable_map : configMap.settable_map,
    config_map : configMap
  });
  return true;
};
// Конец открытого метода /configModule/

// Начало открытого метода /initModule/ ←
// Назначение: инициализирует модуль
// Аргументы :
// * $container элемент jquery, используемый этим модулем
// Возвращает: true
// Исключения: none
//

```

Создаем метод `configModule`. Для изменения параметров любого функционального модуля мы всегда используем метод с одним и тем же именем и служебную функцию `spa.util.setConfigMap`.

Добавляем метод `initModule`. Такой метод есть почти во всех наших модулях. Он начинает выполнение модуля.

```

initModule = function ( $container ) {
    $container.html( configMap.main_html );
    stateMap.$container = $container;
    setJqueryMap();
    return true;
};
// Конец открытого метода /initModule/

// вернуть открытые методы
return {
    configModule : configModule,
    initModule   : initModule
};
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
}());

```

Заполняем контейнер выплывающего чата, копируя в него HTML-шаблон.

Экспортируем методы модуля configModule и initModule. Эти методы стандартно присутствуют почти во всех функциональных модулях.

Теперь создадим модель, как показано в листинге 4.5. Это тоже заглушка. Как и во всех наших модулях, имя файла (spa.model.js) включает имя содержащегося в нем пространства имен (spa.model):

Листинг 4.5 ❖ Наша модель (заглушка) – spa/js/spa.model.js

```

/*
 * spa.model.js
 * Модуль, содержащий модель
 */

/*jslint      browser : true,    continue : true,
   devel : true, indent : 2,      maxerr : 50,
   newcap : true, nomen : true,   plusplus : true,
   regexp : true, sloppy : true,  vars : false,
   white : true
 */

/*global $, spa */

spa.model = (function () { return {}; }());

```

Далее напомним служебный модуль, в который поместим все функции, полезные другим модулям (листинг 4.6). Метод `makeError` позволяет легко создать объект ошибки. Метод `setConfigMap` дает простой и единообразный способ изменения настроек модулей. Поскольку оба метода открыты, мы подробно описываем порядок обращения к ним – в интересах других разработчиков.

Листинг 4.6 ❖ Общие служебные методы – spa/js/spa.util.js

```

/*
 * spa.util.js
 * Общие служебные методы
 */

```

```

* Michael S. Mikowski - mmikowski at gmail dot com
* Эти функции я создавал и модифицировал, начиная с 1998 года,
* черпая вдохновение из веб.
*
* Лицензия МТИ
*
*/
/*jslint      browser : true,    continue : true,
   devel    : true,    indent : 2,      maxerr : 50,
   newcap   : true,    nomen  : true,   plusplus : true,
   regexp   : true,    sloppy  : true,   vars     : false,
   white    : true
*/
/*global $, spa */

spa.util = (function () {
    var makeError, setConfigMap;

    // Начало открытого конструктора /makeError/
    // Назначение: обертка для создания объекта ошибки
    // Аргументы:
    //   * name_text - имя ошибки
    //   * msg_text  - длинное сообщение об ошибке
    //   * data      - необязательные данные, сопровождающие объект ошибки
    // Возвращает: вновь сконструированный объект ошибки
    // Исключения: нет
    //
    makeError = function ( name_text, msg_text, data ) {
        var error    = new Error();
        error.name   = name_text;
        error.message = msg_text;

        if ( data ){ error.data = data; }

        return error;
    };

    // Конец открытого конструктора /makeError/

    // Начало открытого метода /setConfigMap/
    // Назначение: установка конфигурационных параметров в функциональных
    // модулях
    // Аргументы :
    //   * input_map   - хэш ключей и значений, устанавливаемых в config
    //   * settable_map - хэш допустимых ключей
    //   * config_map  - хэш, к которому применяются новые параметры
    // Возвращает: true
    // Исключения: если входной ключ недопустим
    //
    setConfigMap = function ( arg_map ){
        var

```

```

    input_map    = arg_map.input_map,
    settable_map = arg_map.settable_map,
    config_map   = arg_map.config_map,
    key_name, error;

    for ( key_name in input_map ){
        if ( input_map.hasOwnProperty( key_name ) ){
            if ( settable_map.hasOwnProperty( key_name ) ){
                config_map[key_name] = input_map[key_name];
            }
            else {
                error = makeError( 'Bad Input',
                                    'Setting config key |' + key_name + '|' is not supported'
                                );
                throw error;
            }
        }
    }
};
// Конец открытого метода /setConfigMap/
return {
    makeError    : makeError,
    setConfigMap : setConfigMap
};
}());

```

И напоследок свяжем все эти изменения вместе, изменив HTML-документ, в котором загружаются файлы JavaScript и CSS. Сначала загружаем таблицы стилей, потом JavaScript. Порядок перечисления JavaScript-библиотек *важен*: сторонние библиотеки следует загружать раньше, потому что от них часто зависят последующие библиотеки; к тому же такая практика помогает устранить иногда возникающую неразбериху в сторонних пространствах имен (см. врезку «Почему наши библиотеки загружаются в последнюю очередь»). Затем идут наши библиотеки, которые должны быть упорядочены в соответствии с иерархией пространств имен, например модули, содержащие пространства имен `spa`, `spa.model` и `spa.model.user`, следует загружать именно в таком порядке. Всякое дополнительное упорядочение сверх указанного выше – вопрос соглашения, следовать которому необязательно. Мы предпочитаем такое соглашение: корень -> основные утилиты -> Model -> утилиты для работы с браузером -> Shell -> функциональные модули.

Почему наши библиотеки загружаются в последнюю очередь

Мы предпочитаем, чтобы за нашими библиотеками оставалось последнее слово в вопросе о пространствах имен, поэтому загружаем их в по-

следнюю очередь. Если какая-нибудь хулиганская сторонняя библиотека объявит пространство имен `spa.model`, то наша библиотека, загруженная после нее, «вернет все на круги своя». В такой ситуации наше SPA, скорее всего, будет работать нормально, а какая-то функция сторонней библиотеки работать перестанет. Если изменить порядок загрузки библиотек на противоположный, то наше SPA почти наверняка окажется *полностью* неработоспособным. Мы предпочли бы исправлять ошибку, относящуюся, скажем, к сторонней функции добавления комментариев, а не объяснять генеральному директору, почему наш сайт *вообще перестал работать* в полночь.

В листинге 4.7 приведена новая версия HTML-документа. Изменения, по сравнению с главой 3, выделены **полужирным** шрифтом.

Листинг 4.7 ❖ Изменения в HTML-документе – `spa/spa.html`

```
<!doctype html>
<!--
  spa.html
  HTML-документ SPA
-->

<html>
<head>
  <!-- поддержка последних стандартов в части отрисовки в ie9+ -->
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <title>SPA Chapter 4</title>
  <!-- сторонние таблицы стилей -->
  <link rel="stylesheet" href="css/spa.css" type="text/css"/>
  <link rel="stylesheet" href="css/spa.chat.css" type="text/css"/>
  <link rel="stylesheet" href="css/spa.shell.css" type="text/css"/>
  <!-- сторонний javascript -->
  <script src="js/jq/jquery-1.9.1.js" ></script>
  <script src="js/jq/jquery.uriAnchor-1.1.3.js"></script>
  <!-- наш javascript -->
  <script src="js/spa.js" ></script>
  <script src="js/spa.util.js" ></script>
  <script src="js/spa.model.js"></script>
  <script src="js/spa.shell.js"></script>
  <script src="js/spa.chat.js" ></script>
</script>
```

Добавляем заголовки, чтобы работал IE9+.

Изменяем название, упомянув в нем новую главу. Извини, друг, глава 3 осталась позади.

Добавляем секцию для сторонних таблиц стилей.

Включаем наши таблицы стилей. Чтобы упростить сопровождение, перечисляем их в том же порядке, в каком JS-файлы.

Сначала включаем сторонний JavaScript. О причинах см. врезку.

Включаем наши библиотеки в порядке вложенности пространств имен. Как минимум, пространство имен `spa` должно быть загружено первым.

Включаем нашу библиотеку служебных функций, общих для всех модулей.

Включаем браузерную модель, которая пока является заглушкой.

После Shell загружаем функциональные модули.


```

    $(function () { spa.initModule( $('#spa') ); });
  </script>

</head>
<body>
  <div id="spa"></div>
</body>
</html>

```

Теперь добавим в Shell код, который настраивает и инициализирует модуль Chat, как показано в листинге 4.8.

Листинг 4.8 ❖ Изменения в модуле Shell – spa/js/spa.shell.js

```

...
// настраиваем uriAnchor на использование нашей схемы
$.uriAnchor.configModule({
  schema_map : configMap.anchor_schema_map
});

// настраиваем и инициализируем функциональные модули
spa.chat.configModule( {} );
spa.chat.initModule( jqueryMap.$chat );

// Обрабатываем события изменения якоря в URI.
...

```

Первый проход закончен. Работы оказалось довольно много, но большую часть этих шагов в последующих функциональных модулях повторять не придется. Посмотрим, что же у нас получилось.

4.2.3. Что мы соорудили

Если загрузить в браузер документ spa/spa.html, то окно чата будет выглядеть, как показано на рис. 4.8.

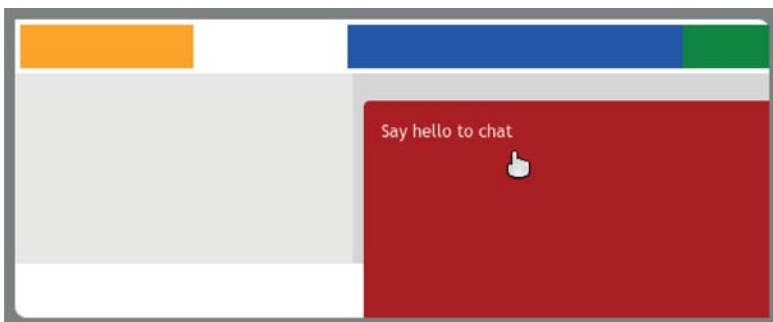


Рис. 4.8 ❖ Обновленный HTML-документ – spa/spa.html

Наличие текста `Say hello to chat` показывает, что чат сконфигурирован и инициализирован правильно, и содержимое выплывающего чата показывается. Но этот интерфейс отнюдь не впечатляет. В следующем разделе мы его значительно улучшим.

4.3. Проектирование API модуля

Согласно нашей архитектуре, модуль `Shell` может обращаться к любому подчиненному модулю в SPA. Функциональные модули имеют право вызывать только методы из служебных модулей; вызовы одного функционального модуля из другого *запрещены*. Единственным дополнительным источником данных или возможностей для функционального модуля является `Shell`, и поступать они могут только в виде аргументов открытых методов модуля, например на этапе конфигурирования или инициализации. Это разбиение на уровни показано на рис. 4.9.

Такая изоляция – сознательное решение, потому что она предотвращает распространение ошибок, имеющих в функциональном модуле, на уровень приложения или в другие функциональные модули¹.

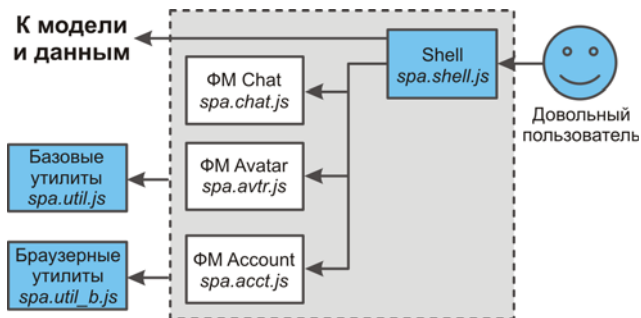


Рис. 4.9 ❖ Функциональные модули крупным планом – допустимые вызовы

4.3.1. Паттерн якорного интерфейса

Напомним (см. главу 3), что мы хотели, чтобы якорь в URI управлял состоянием страницы, а не наоборот. Иногда кажется, что путь выполнения трудно проследить, потому что за управление якорем

¹ Коммуникация между функциональными модулями координируется с помощью `Shell` или модели.

отвечает Shell, а за визуализацию чата – Chat. Мы опираемся на *паттерн якорного интерфейса* для поддержки якорей в URI и состояний, управляемых пользовательскими событиями, используя в обоих случаях *одно и то же событие jQuery hashchange*. Поскольку изменить состояние приложения можно только одним способом, то мы получаем URL-адреса, безопасные относительно истории¹, согласованное поведение и ускорение разработки. Этот паттерн показан на рис. 4.10.

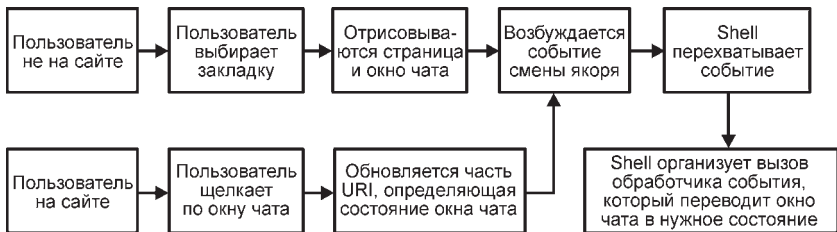


Рис. 4.10 ❖ Паттерн якорного интерфейса для Chat

В предыдущей главе значительная часть поведения Chat уже была реализована. Теперь перенесем оставшийся код чата в его модуль. Определим также вызовы API, с помощью которых будут взаимодействовать модули Chat и Shell. Это сразу принесет свои плоды и заодно существенно упростит повторное использование кода. В спецификации API должно быть подробно описано, какие требуются ресурсы и какие предоставляются возможности. Спецификация является развивающимся документом, который актуализируется после каждого изменения API.

Мы хотим, в частности, чтобы модуль Chat предоставлял стандартный открытый метод `configModule`, который будет вызываться для изменения параметров перед инициализацией. У Chat, как и у любого функционального модуля, должен быть также метод инициализации `initModule`. Кроме того, нам понадобится метод `setSliderPosition`, чтобы Shell мог запросить установку состояния выплывающего чата. В следующих разделах мы займемся проектированием API этих методов.

¹ «Безопасность относительно истории» означает, что все встроенные в браузер элементы управления историей – кнопки «Вперед» и «Назад», закладки и история посещений – работают в соответствии с ожиданиями пользователя.

4.3.2. API конфигурирования модуля Chat

На этапе *конфигурирования* модуля мы задаем параметры, которые не должны изменяться на протяжении сеанса работы с пользователем. В модуле Chat к таковым относятся следующие:

- функция, обеспечивающая настройку параметров чата в якорю URI;
- объект, предоставляющий методы для отправки и получения сообщений от модели;
- объект, предоставляющий методы для взаимодействия со списком пользователей (полученным от модели);
- параметры, описывающие поведение, например высота открытого окна чата, время открытия и закрытия окна.

Темные стороны аргументов в JavaScript

Напомним, что только простые значения – строки, числа и булевы величины – передаются функциям непосредственно. Все составные типы данных (объекты, массивы и функции) в JavaScript *передаются по ссылке*. Это означает, что они *никогда* не копируются, как в некоторых других языках. Передается лишь адрес в памяти. Обычно такой подход гораздо быстрее копирования, но у него есть недостаток: очень просто непреднамеренно изменить переданный по ссылке объект или массив.

Если функция ожидает получить ссылку на функцию в качестве аргумента, то такую ссылку принято называть *обратным вызовом*. Это мощный механизм, но управлять им не всегда легко. В главах 5 и 6 мы покажем, как можно сократить потребность в обратных вызовах за счет использования глобальных специализированных событий jQuery.

Исходя из этих ожиданий, мы предлагаем спецификацию метода `configModule`, показанную в листинге 4.9. Самой программе наличие или отсутствие документации безразлично.

Листинг 4.9 ❖ Спецификация метода `configModule` из API модуля Chat – `spa/js/spa.chat.js`

```
// Начало открытого метода /configModule/
// Пример: spa.chat.configModule({ slider_open_em : 18 });
// Назначение: сконфигурировать модуль до инициализации
// Аргументы :
// * set_chat_anchor – обратный вызов для модификации якоря в URI,
//   чтобы отразить состояние: открыт или закрыт. Обратный вызов должен
//   возвращать false, если установить указанное состояние невозможно.
// * chat_model – объект модели чата, который предоставляет методы для
//   взаимодействия с нашей системой мгновенного обмена сообщениями.
// * people_model – объект модели пользователя, который предоставляет
//   методы для управления списком пользователей, хранящимся в модели.
// * параметры slider_*. Все это необязательные скаляры.
```

```
// Полный перечень см. в mapConfig.settable_map.  
// Пример: slider_open_em – высота в открытом состоянии в единицах em  
// Действие:  
// Внутренняя структура, в которой хранятся конфигурационные параметры  
// (configMap), обновляется в соответствии с переданными аргументами.  
// Больше никаких действий не предпринимается.  
// Возвращает: true  
// Исключения: объект ошибки JavaScript и трассировка стека в случае  
// недопустимых или недостающих аргументов  
//
```

Теперь, имея API для конфигурирования чата, перейдем к спецификации обратного вызова `setChatAnchor` из Shell. Листинг 4.10 дает неплохую отправную точку. Самой программе наличие или отсутствие документации безразлично.

Листинг 4.10 ❖ Спецификация обратного вызова `setChatAnchor` из Shell – `spa/js/spa.shell.js`

```
// Начало метода обратного вызова /setChatAnchor/  
// Пример: setChatAnchor( 'closed' );  
// Назначение: изменить компонент якоря, относящийся к чату  
// Аргументы:  
// * position_type – допустимы значения 'closed' или 'opened'  
// Действие:  
// Заменяет параметр 'chat' в ядре указанным значением, если это  
// возможно.  
// Возвращает:  
// * true – часть якоря была обновлена  
// * false – часть якоря не была обновлена  
// Исключения: нет  
//
```

Покончив с проектированием API конфигурирования чата и обратного вызова из Shell, мы можем заняться инициализацией модуля Chat.

4.3.3. API инициализации модуля Chat

Инициализируя функциональный модуль, мы просим его отрисовать свою HTML-разметку и начать предоставление услуг пользователю. В отличие от конфигурирования, инициализация функционального модуля может производиться несколько раз на протяжении сеанса. В случае модуля Chat мы хотим передать в качестве единственного аргумента коллекцию jQuery, которая будет содержать один элемент – тот, в который мы собираемся добавить всплывающий чат. Набросок API показан в листинге 4.11.

Листинг 4.11 ❖ Спецификация метода `initModule` из API модуля Chat – `spa/js/spa.chat.js`

```
// Начало открытого метода /initModule/
// Пример: spa.chat.initModule( $('#div_id') );
// Назначение:
//   Требуется, чтобы Chat начал предоставлять свою функциональность
//   пользователю
// Аргументы:
//   * $append_target (example: $('#div_id')).
//     Коллекция jQuery, которая должна содержать
//     единственный элемент DOM – контейнер
// Действие:
//   Добавляет выплывающий чат в конец указанного контейнера и заполняет
//   его HTML-содержимым. Затем инициализирует элементы, события и
//   обработчики, так чтобы предоставить пользователю интерфейс для работы
//   с чатом.
// Возвращает: true в случае успеха, иначе false
// Исключения: нет
//
```

И последний метод API модуля Chat, который мы опишем в этой главе, – `setSliderPosition`. Он будет открывать и закрывать выплывающее окно чата.

4.3.4. Метод `setSliderPosition` из API модуля Chat

Мы решили наделить модуль Chat открытым методом `setSliderPosition`, который позволит запрашивать установку состояния окна чата из Shell. Наше решение включить состояние чата в якорь в составе URI поднимает несколько интересных вопросов, которые придется разрешить.

Модуль Chat не всегда может изменить состояние выплывающего окна, как указано в запросе. Например, он может решить, что чат нельзя открыть, потому что пользователь еще не аутентифицировался. Поэтому метод `setSliderPosition` будет возвращать `true` или `false`, чтобы Shell знал, успешно ли выполнен запрос.

Если Shell обращается к обратному вызову `setSliderPosition`, а тот не может удовлетворить запрос (то есть возвращает `false`), то Shell должен восстановить предыдущее значение параметра `chat` в якорь.

В листинге 4.12 определен API, отвечающий этим требованиям.

Листинг 4.12 ❖ Спецификация метода `setSliderPosition` из API модуля Chat – `spa/js/spa.chat.js`

```
// Начало открытого метода /setSliderPosition/
//
// Пример: spa.chat.setSliderPosition( 'closed' );
```

```
// Назначение: установить окно чата в требуемое состояние
//Аргументы:
// * position_type - enum('closed', 'opened', 'hidden')
// * callback - необязательная функция, вызываемая по завершении анимации.
// (в качестве аргумента обратному вызову передается элемент DOM,
// представляющий выплывающий чат)
// Действие:
// Оставляет окно чата в текущем состоянии, если новое состояние совпадает
// с текущим, иначе анимирует переход в новое состояние.
//Возвращает:
// * true - запрошенное состояние установлено
// * false - запрошенное состояние не установлено
// Исключения: нет
//
```

Определив API, мы уже почти готовы к написанию кода. Но сначала рассмотрим порядок конфигурирования и инициализации в нашем приложении.

4.3.5. Каскадное конфигурирование и инициализация

Конфигурирование и инициализация в нашей архитектуре устроены единообразно. Сначала `тег script` в головном HTML-документе конфигурирует и инициализирует модуль, содержащий *корневое пространство имен* `spa.js`. Затем корневой модуль конфигурирует и инициализирует модуль `Shell` `spa.shell.js`. Далее модуль `Shell` конфигурирует и инициализирует функциональный модуль `spa.chat.js`. Это каскадное конфигурирование и инициализация показаны на рис. 4.11.



Рис. 4.11 ❖ Каскадное конфигурирование и инициализация

Любой из наших модулей предоставляет открытый метод `initModule`. Метод `configModule` необходим лишь в том случае, когда модуль поддерживает задание параметров. На данном этапе конфигурирование допускает только модуль `Chat`.

Когда документ `spa/spa.html` открывается в браузере, он загружает все наши файлы CSS и JavaScript. Затем находящийся на этой странице скрипт производит кое-какие служебные действия и инициализирует модуль, содержащий корневое пространство имен (`spa/js/spa.js`), сообщая ему, какой элемент страницы (`DIV spa`) использовать:

```
$(function () {
    // служебные действия ...

    //если бы нужно было сконфигурировать корневой модуль,
    // то мы сначала вызвали бы метод spa.configModule

    spa.initModule( $('#spa' ) );
})();
```

После инициализации корневой модуль (`spa/js/spa.js`) производит служебные действия на своем уровне, а затем конфигурирует и инициализирует модуль `Shell` (`spa/js/spa.shell.js`), сообщая ему, какой элемент страницы (`$container`) использовать:

```
var initModule = function ( $container ) {
    // служебные действия ...

    //если бы нужно было сконфигурировать Shell,
    // то мы сначала вызвали бы метод spa.shell.configModule

    spa.shell.initModule( $container );
};
```

Модуль `Shell` (`spa/js/spa.shell.js`) затем производит служебные действия на своем уровне, после чего конфигурирует и инициализирует все функциональные модули, например `Chat` (`spa/js/spa.chat.js`), сообщая каждому, какой элемент страницы (`jQueryMap.$chat`) использовать:

```
initModule = function ( $container ) {
    // служебные действия ...

    //конфигурируем и инициализируем функциональные модули
    spa.chat.configModule( {} );
    spa.chat.initModule( jQueryMap.$chat );

    // ...
};
```


Вы должно хорошо понимать этот каскад, потому что он относится ко всем функциональным модулям. Например, мы могли бы вынести часть функциональности Chat (`spa/js/spa.chat.js`) в подчиненный модуль, который занимается управлением списком пользователей (назовем его Roster), и создать соответствующий ему файл `spa/js/spa.chat.roster.js`. Тогда модуль Chat должен был бы вызвать метод `spa.chat.roster.configModule` для конфигурирования этого модуля и метод `spa.chat.roster.initModule` — для его инициализации. Chat должен был бы также предоставить модулю Roster контейнер jQuery, где тот показывал бы список пользователей.

Вот теперь мы готовы изменить приложение, приведя его в соответствие с предложенным API. Кое-какие из внесенных изменений приведут к частичной потере работоспособности, но не пугайтесь — скоро мы все исправим.

4.4. Реализация API функционального модуля

Нашей основной целью в этом разделе будет реализация определенного ранее API. Но попутно мы решим и несколько других задач.

Полностью перенесем код конфигурирования чата в модуль Chat. Единственный аспект чата, до которого есть дело модулю Shell, — управление якорем в URI.

Сделаем интерфейс больше похожим на чат. В листинге 4.13 перечислены файлы, которые мы собираемся модифицировать, с кратким описанием существа изменений.

Листинг 4.13 ❖ Файлы, которые изменятся в процессе реализации API

```
spa
+-- css
| +-- spa.chat.css # Перенести стили чата из файла spa.shell.css, доработать
| '-- spa.shell.css # Удалить стили чата
'-- js
    +-- spa.chat.js # Перенести функциональность из Shell, реализовать API
    '-- spa.shell.js # Удалить функциональность Chat
                        # и добавить вызов setSliderPosition
```

Будем модифицировать файлы в указанном порядке.

4.4.1. Таблицы стилей

Мы хотим перенести все стили, относящиеся к чату, в отдельный файл (`spa/css/spa.chat.css`) и попутно улучшить макет. Наш специа-

лист по CSS-верстке предложил изящную структуру, показанную на рис. 4.12.

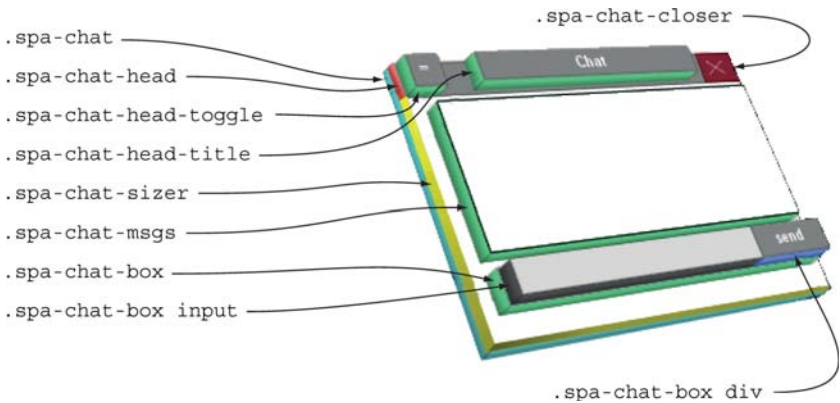


Рис. 4.12 ❖ Трехмерное представление элементов и селекторов – `spa/css/spa.chat.css`

Обратите внимание на введение в CSS пространств имен по аналогии с JavaScript. У такого подхода есть целый ряд достоинств:

- не нужно беспокоиться о конфликтах с другими нашими модулями, потому что мы выбрали уникальный префикс для всех имен классов: `spa-chat`;
- почти никогда не возникает конфликтов со сторонними пакетами. А даже если такое вдруг случится, исправление тривиально (достаточно изменить префикс);
- очень помогает при отладке, потому что, inspectируя элемент, контролируемый модулем `Chat`, мы сразу – по имени класса – понимаем, что он относится к функциональному модулю `spa.chat`;
- имена говорят о том, что в чем содержится (а значит, и контролируется). Например, мы знаем, что элемент `spa-chat-head-toggle` содержится внутри элемента `spa-chat-head`, который содержится внутри `spa-chat`.

Большая часть стилей стереотипна (извини, спец по CSS-верстке). Но есть несколько моментов, заслуживающих упоминания. Во-первых, элемент `spa-chat-sizer` должен иметь фиксированную высоту. Это оставит место для области заголовка и сообщений даже в том случае, когда окно чата свернуто. Если не включить этот элемент, то содержимое окна чата при сворачивании «сомнется», и пользователь

в лучшем случае останется в недоумении. Во-вторых, наш специалист по верстке посоветовал заменить все абсолютные единицы измерения (например, пиксели) относительными: `em` и процентами. В результате наше SPA будет выглядеть одинаково хорошо на экранах с высоким и низким разрешением.

Пиксели и относительные единицы измерения

Гуру в области HTML часто пускаются на всевозможные ухищрения, чтобы использовать в CSS только относительные единицы измерения, и полностью отказываются от единицы `px` – все для того, чтобы их творение выглядело одинаково хорошо на экране любого размера. Мы наблюдали явление, которое заставило нас пересмотреть ценность такого подхода: браузеры лгут, когда речь заходит о размерах их окна в пикселях.

Рассмотрим последние модели ноутбуков, планшетов и смартфонов, оснащенные экранами со сверхвысоким разрешением. На таких устройствах браузер не соотносит единицу `px` с количеством физических пикселей на экране. Вместо этого он нормализует единицу `px`, так чтобы приблизить характеристики просмотра к традиционному настольному монитору с плотностью пикселей в районе от 96 до 120 пикселей/дюйм.

В результате квадрат со стороной 10 `px` может отрисовываться браузером на смартфоне так, что на каждой стороне окажется от 15 до 20 физических пикселей. Это означает, что `px` также оказывается относительной единицей измерения, причем по сравнению с другими (`%`, `in`, `cm`, `mm`, `em`, `ex`, `pt`, `pc`) зачастую более надежной. В числе других устройств у нас есть планшеты с диагональю 10,1 и 7 дюймов с одними и теми же разрешением (1280×800) и операционной системой. Квадрат со стороной 400 `px` помещается на экране 10,1-дюймового планшета, но не помещается на экране 7-дюймового. Почему? Потому что на меньшем планшете количество физических пикселей в одной единице `px` больше. Создается впечатление, что на большем планшете один `px` равен 1,5 пикселя, а на меньшем – 2 пикселя.

Мы не знаем, что в этом отношении готовит будущее, но с недавних пор использование единицы измерения `px` перестало вызывать у нас острое чувство вины.

Все спланировав, мы можем теперь добавить в файл `spa.chat.css` CSS-стили, отвечающие спецификациям (листинг 4.14).

Листинг 4.14 ❖ Добавление улучшенных стилей чата – `spa.chat.css`

```
/*
```

```
* spa.chat.css
```

```
* Стили функционального модуля Chat
```

```
*/
```

Определяем класс `spa-chat` для окна чата. Включаем светлые тени. Как и во всех селекторах чата, переходим к относительным единицам измерения.

```
.spa-chat {
  position      : absolute;
  bottom        : 0;
  right         : 0;
  width         : 25em;
```

```

height      : 2em;
background  : #fff;
border-radius : 0.5em 0 0 0;
border-style : solid;
border-width : thin 0 0 thin;
border-color : #888;
box-shadow   : 0 0 0.75em 0 #888;
z-index      : 1;
}

```

Добавляем общие правила для классов `spa-chat-head` и `spa-chat-closer`. Тем самым мы остаемся верны принципу DRY («Не повторяйся»). Но при этом со всей определенностью скажем: мы ненавидим этот акроним.

```

.spa-chat-head, .spa-chat-closer {
  position      : absolute;
  top           : 0;
  height        : 2em;
  line-height    : 1.8em;
  border-bottom  : thin solid #888;
  cursor        : pointer;
  background     : #888;
  color         : white;
  font-family    : arial, helvetica, sans-serif;
  font-weight    : 800;
  text-align     : center;
}

```

Добавляем уникальные правила для класса `spa-chat-head`. Мы ожидаем, что элемент с таким классом будет содержать элементы с классами `spa-chat-head-toggle` и `spa-chat-head-title`.

```

.spa-chat-head {
  left          : 0;
  right         : 2em;
  border-radius  : 0.3em 0 0 0;
}

```

Определяем класс `spa-chat-closer` для стилизации значка [x] в правом верхнем углу окна. Отметим, что этот элемент не содержится внутри заголовка, потому что мы хотим, чтобы заголовок служил для открытия и закрытия окна чата, а у «закрываателя» (`closer`) другая функция. Мы добавили также псевдокласс `:hover` для подсветки элемента, когда на него наведен курсор.

```

.spa-chat-closer {
  right         : 0;
  width         : 2em;
}

.spa-chat-closer:hover {
  background    : #800;
}

```

Создаем класс `spa-chat-head-toggle` для кнопки-переключателя. Как следует из названия, мы планируем, что элемент с таким классом будет находиться внутри элемента с классом `spa-chat-head`.

```

.spa-chat-head-toggle {
  position      : absolute;
  top           : 0;
  left          : 0;
  width         : 2em;
  bottom        : 0;
  border-radius  : 0.3em 0 0 0;
}

```

Создаем класс `spa-chat-head-title`. И снова из названия вытекает, что элемент с таким классом предположительно должен находиться внутри элемента с классом `spa-chat-head`. Мы используем стандартный прием с «отрицательным полем» для центрирования элемента (о деталях спросите у Google).

```

.spa-chat-head-title {
  position      : absolute;

```

```

    left      : 50%;
    width     : 16em;
    margin-left : -8em;
}

```

Определяем класс `spa-chat-sizer`, чтобы можно было поместить содержимое выплывающего чата внутрь элемента фиксированного размера.

```

.spa-chat-sizer {
    position : absolute;
    top      : 2em;
    left     : 0;
    right    : 0;
}

```

Добавляем класс `spa-chat-messages`, который будет использоваться в элементе для отображения сообщений. Мы скрываем переполнение по оси `x` и всегда показываем вертикальную полосу прокрутки (можно было бы задать правило `overflow-y: auto`, но тогда при появлении полосы прокрутки будет наблюдаться режущее глаз переформатирование текста).

```

.spa-chat-msgs {
    position : absolute;
    top      : 1em;
    left     : 1em;
    right    : 1em;
    bottom   : 4em;
    padding  : 0.5em;
    border   : thin solid #888;
    overflow-x : hidden;
    overflow-y : scroll;
}

```

Создаем класс `spa-chat-box` для элемента, который должен содержать поле ввода и кнопку «Отправить».

```

.spa-chat-box {
    position : absolute;
    height   : 2em;
    left     : 1em;
    right    : 1em;
    bottom   : 1em;
    border   : thin solid #888;
    background : #888;
}

```

Определяем правило для стилизации «любого поля ввода текста внутри любого элемента с классом `spa-chat-box`». Это как раз и будет поле для ввода сообщения в чате.

```

.spa-chat-box input[type=text] {
    float    : left;
    width    : 75%;
    height   : 100%;
    padding  : 0.5em;
    border   : 0;
    background : #ddd;
    color    : #404040;
}

```

Создаем производный псевдокласс `:focus`, увеличивающий контрастность при входе в поле ввода.

```

.spa-chat-box input[type=text]:focus {
    background : #fff;
}

```

Определяем правило для стилизации «любого элемента `div` внутри элемента с классом `spa-chat-box`». Это будет наша кнопка «Отправить».

```

.spa-chat-box div {
    float : left;
}

```

```

width      : 25%;
height     : 2em;
line-height : 1.9em;
text-align : center;
color      : #fff;
font-weight : 800;
cursor     : pointer;
}

```

Создаем производный псевдокласс :hover для подсветки кнопки «Отправить», когда пользователь наводит на нее мышью.

```

.spac-chat-box div: hover {
    background-color : #444;
    color            : #fff;
}

```

Определяем селектор, который выделяет элементы с классом spac-chat-head-toggle, находящиеся внутри элемента с классом spac-chat-head, когда на него наведен курсор мыши.

```

.spac-chat-head: hover spac-chat-head-toggle {
    background : #aaa;
}

```

Имея таблицу стилей для чата, мы можем удалить старые определения из таблицы модуля Shell в файле spac/css/spa.shell.css. Сначала удалим класс **.spac-shell-chat** из списка селекторов с абсолютным позиционированием. В результате должно получиться следующее (комментарий можно опустить):

```

.spac-shell-head, .spac-shell-head-logo, .spac-shell-head-acct,
.spac-shell-head-search, .spac-shell-main, .spac-shell-main-nav,
.spac-shell-main-content, .spac-shell-foot, /* .spac-shell-chat */
.spac-shell-modal {
    position : absolute;
}

```

Мы хотим также удалить все классы **.spac-shell-chat** из файла spac/css/spa.shell.css. Таких всего два:

```

/* удалить это из spac/css/spa.shell.css
.spac-shell-chat {
    bottom : 0;
    right  : 0;
    width  : 300px;
    height : 15px;
    cursor : pointer;
    background : red;
    border-radius : 5px 0 0 0;
    z-index : 1;
}
.spac-shell-chat: hover {
    background : #a00;
} */

```

Наконец, скроем модальный контейнер, чтобы он не путался под ногами нашего окна чата:

```
...  
.spa-shell-modal {  
  ...  
  display: none;  
}  
...
```

Теперь можно открыть файл `spa/spa.html` в браузере и, заглянув на консоль JavaScript в инструментах разработчика в Chrome, убедиться в отсутствии ошибок. Однако окно чата больше не видно. Спокойно, не дергайтесь — мы исправим это, когда закончим модификацию модуля Chat в следующем разделе.

4.4.2. Модификация модуля Chat

Теперь внесем изменения в код модуля Chat с целью реализовать спроектированный ранее API. Вот перечень запланированных изменений:

- добавить HTML-разметку для более детально проработанного окна чата;
- включить дополнительные конфигурационные параметры, например высоту окна и время сворачивания;
- создать служебный метод `getEmSize` для преобразования единиц измерения `em` в `px` (пиксели);
- изменить `setJqueryMap` так, чтобы кэшировались новые элементы, появившиеся в окне чата;
- добавить метод `setPxSizes` для задания размеров окна чата в пикселях;
- реализовать открытый метод `setSliderPosition`, описанный в API;
- создать обработчик события `onClickToggle`, который изменяет якорь в URI и быстро возвращает управление;
- изменить документацию открытого метода `configModule`, приведя ее в соответствие с API;
- изменить открытый метод `initModule`, приведя его в соответствие с API.

Новая реализация модуля Chat приведена в листинге 4.15. Мы скопировали в этот файл написанные ранее спецификации API и пользовались ими как руководством в ходе кодирования. Это ускоряет разработку и гарантирует точность документации, что важно для последующего сопровождения. Изменения выделены **полужирным** шрифтом.

Листинг 4.15 ❖ Модификация модуля Chat в соответствии со спецификацией –spa/js/spa.chat.js

```

/*
 * spa.chat.js
 * Функциональный модуль Chat для SPA
 */

/*jslint      browser : true,   continue : true,
   devel : true,   indent : 2,   maxerr : 50,
   newcap : true,  nomen : true, plusplus : true,
   regexp : true,  sloppy : true, vars : false,
   white : true
 */

/*global $, spa, getComputedStyle */

spa.chat = (function () {
    //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
    var
        configMap = {
            main_html : String()
                + '<div class="spa-chat">'
                + '<div class="spa-chat-head">'
                + '<div class="spa-chat-head-toggle"></div>'
                + '<div class="spa-chat-head-title">'
                + 'Chat'
                + '</div>'
                + '</div>'
                + '<div class="spa-chat-closer">x</div>'
                + '<div class="spa-chat-sizer">'
                + '<div class="spa-chat-msgs"></div>'
                + '<div class="spa-chat-box">'
                + '<input type="text"/>'
                + '<div>send</div>'
                + '</div>'
                + '</div>'
                + '</div>',
            settable_map : {
                slider_open_time : true,
                slider_close_time : true,
                slider_opened_em : true,
                slider_closed_em : true,
                slider_opened_title : true,
                slider_closed_title : true,

                chat_model : true,
                people_model : true,
                set_chat_anchor : true
            }
        }
    }
    //----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
    return {
        main_html : configMap.main_html,
        settable_map : configMap.settable_map
    }
})();

```

Пользуемся шаблоном функционального модуля из приложения А.

Задаем HTML-шаблон для заполнения контейнера выплывающего чата.

Переносим все параметры чата в этот модуль.


```

    },

    slider_open_time    : 250,
    slider_close_time   : 250,
    slider_opened_em    : 16,
    slider_closed_em    : 2,
    slider_opened_title : 'Click to close',
    slider_closed_title : 'Click to open',

    chat_model          : null,
    people_model         : null,
    set_chat_anchor     : null
  },
  stateMap = {
    $append_target      : null,
    position_type       : 'closed',
    px_per_em            : 0,
    slider_hidden_px    : 0,
    slider_closed_px    : 0,
    slider_opened_px    : 0
  },
  jqueryMap = {},

  setJqueryMap, getEmSize, setPxSizes, setSliderPosition,
  onClickToggle, configModule, initModule
;
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

//----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
getEmSize = function ( elem ) {
  return Number(
    getComputedStyle( elem, '' ).fontSize.match(/d+\.?\d+\/)[0]
  );
};
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

//----- НАЧАЛО МЕТОДОВ DOM -----
// Начало метода DOM /setJqueryMap/
setJqueryMap = function () {
  var
    $append_target = stateMap.$append_target,
    $slider = $append_target.find( '.spa-chat' );

  jqueryMap = {
    $slider : $slider,
    $head   : $slider.find( '.spa-chat-head' ),
    $toggle : $slider.find( '.spa-chat-head-toggle' ),
    $title  : $slider.find( '.spa-chat-head-title' ),
    $sizer  : $slider.find( '.spa-chat-sizer' ),
    $msgs   : $slider.find( '.spa-chat-msgs' ),
  }

```

Добавляем метод `getEmSize` для преобразования единиц измерения `em` в пиксели, чтобы можно было производить измерения в jQuery.

Изменяем метод `setJqueryMap`, добавляя кэширование дополнительных коллекций jQuery. Мы предпочитаем использовать классы вместо идентификаторов, потому что это позволяет без каких-либо переделок применить на одной странице несколько окон чата.

```

$box : $slider.find( '.spa-chat-box' ),
$input : $slider.find( '.spa-chat-input input[type=text]' );
};
// Конец метода DOM /setJqueryMap/

// Начало метода DOM /setPxSizes/
setPxSizes = function () {
    var px_per_em, opened_height_em;
    px_per_em = getEmSize( jqueryMap.$slider.get(0) );

    opened_height_em = configMap.slider_opened_em;

    stateMap.px_per_em = px_per_em;
    stateMap.slider_closed_px = configMap.slider_closed_em * px_per_em;
    stateMap.slider_opened_px = opened_height_em * px_per_em;
    jqueryMap.$sizer.css({
        height : ( opened_height_em - 2 ) * px_per_em
    });
};
// Конец метода DOM /setPxSizes/

// Начало открытого метода /setSliderPosition/
//
// Пример: spa.chat.setSliderPosition( 'closed' );
// Назначение: установить окно чата в требуемое состояние
// Аргументы:
// * position_type - enum('closed', 'opened', 'hidden')
// * callback - необязательная функция, вызываемая по завершении
//   анимации (в качестве аргумента обратному вызову передается элемент
//   DOM, представляющий всплывающий чат).
// Действие:
// * Оставляет окно чата в текущем состоянии, если новое состояние
//   совпадает с текущим, иначе анимирует переход в новое состояние.
// Возвращает:
// * true - запрошенное состояние установлено
// * false - запрошенное состояние не установлено
// Исключения: нет
//
setSliderPosition = function ( position_type, callback ) {
    var
        height_px, animate_time, slider_title, toggle_text;

    // вернуть true, если окно чата уже находится в требуемом состоянии
    if ( stateMap.position_type === position_type ){
        return true;
    }

    // подготовить параметры анимации
    switch ( position_type ){
        case 'opened' :

```

Добавляем метод setPxSizes для вычисления размеров управляемых этим модулем элементов в пикселях.

Добавляем метод setSliderPosition, подробно описанный выше в этой главе.

```

        height_px = stateMap.slider_opened_px;
        animate_time = configMap.slider_open_time;
        slider_title = configMap.slider_opened_title;
        toggle_text = '=';
        break;

    case 'hidden' :
        height_px = 0;
        animate_time = configMap.slider_open_time;
        slider_title = '';
        toggle_text = '+';
        break;

    case 'closed' :
        height_px = stateMap.slider_closed_px;
        animate_time = configMap.slider_close_time;
        slider_title = configMap.slider_closed_title;
        toggle_text = '+';
        break;

    // выйти из метода, если position_type имеет неизвестное значение
    default : return false;
}

// анимировать изменение состояния окна чата
stateMap.position_type = '';
jqueryMap.$slider.animate(
    { height : height_px },
    animate_time,
    function () {
        jqueryMap.$toggle.prop( 'title', slider_title );
        jqueryMap.$toggle.text( toggle_text );
        stateMap.position_type = position_type;
        if ( callback ) { callback( jqueryMap.$slider ); }
    }
);
return true;
};
// Конец открытого метода DOM /setSliderPosition/
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
onClickToggle = function ( event ){ ←
    var set_chat_anchor = configMap.set_chat_anchor;
    if ( stateMap.position_type === 'opened' ) {
        set_chat_anchor( 'closed' );
    }
    else if ( stateMap.position_type === 'closed' ){
        set_chat_anchor( 'opened' );
    }
    return false;
}

```

Изменяем обработчик события `onClick`, так чтобы он вызывал метод изменения якоря в URI, а затем быстро возвращал управление, оставляя дальнейшие действия обработчику события `hashchange` в модуле `Shell`.

```

};
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /configModule/ ←
// Пример: spa.chat.configModule({ slider_open_em : 18 });
// Назначение: сконфигурировать модуль до инициализации
// Аргументы:
// * set_chat_anchor – обратный вызов для модификации якоря в URI,
//   чтобы отразить состояние: открыт или закрыт. Обратный вызов должен
//   возвращать false, если установить указанное состояние невозможно.
// * chat_model – объект модели chat, который предоставляет методы для
//   взаимодействия с нашей системой мгновенного обмена сообщениями.
// * reople_model – объект модели reople, который предоставляет
//   методы для управления списком пользователей, хранящимся в модели.
// * параметры slider_*. Все это необязательные скаляры.
// Полный перечень см. в mapConfig.settable_map.
// Пример: slider_open_em – высота в открытом состоянии в единицах em
// Действие:
// Внутренняя структура, в которой хранятся конфигурационные параметры
// (configMap), обновляется в соответствии с переданными аргументами.
// Больше никаких действий не предпринимается.
// Возвращает: true
// Исключения: объект ошибки JavaScript и трассировка стека в случае
//   недопустимых или недостающих аргументов
//
configModule = function ( input_map ) {
    spa.util.setConfigMap({
        input_map      : input_map,
        settable_map   : configMap.settable_map,
        config_map     : configMap
    });
    return true;
};
// Конец открытого метода /configModule/

// Начало открытого метода /initModule/ ←
// Пример: spa.chat.initModule( $('#div_id') );
// Назначение:
// Требуется, чтобы Chat начал предоставлять свою функциональность
// пользователю
// Аргументы:
// * $append_target (example: $('#div_id')).
// Коллекция jQuery, которая должна содержать
// единственный элемент DOM – контейнер
// Действие:
// Добавляет выплывающий чат в конец указанного контейнера и заполняет
// его HTML-содержимым. Затем инициализирует элементы, события и
// обработчики, так чтобы предоставить пользователю интерфейс для работы
// с комнатами в чате.

```

Изменяем метод configModule в соответствии со спецификацией API. Пользуемся служебной функцией spa.util.setConfigMap, как во всех функциональных модулях, допускающих конфигурирование.

Изменяем метод initModule в соответствии со спецификацией API. Как и в Shell, эта функция состоит из трех частей: 1) поместить HTML-разметку в контейнер, 2) кэшировать коллекции jQuery и 3) инициализировать обработчики событий.

```

//Возвращает: true в случае успеха, иначе false
// Исключения: нет
//
  initModule = function ( $append_target ) {
$append_target.append( configMap.main_html );
stateMap.$append_target = $append_target;
setJqueryMap();
setPxSizes();

// установить начальный заголовок и состояние окна чата
jqueryMap.$toggle.prop( 'title', configMap.slider_closed_title );
jqueryMap.$head.click( onClickToggle );
stateMap.position_type = 'closed';

    return true;
  };
  // Конец открытого метода /initModule/

  // вернуть открытые методы ← Экспортируем открытые методы: configModule,
  //                               initModule и setSliderPosition.
  return {
setSliderPosition : setSliderPosition,
    configModule    : configModule,
    initModule      : initModule
  };
  //----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();

```

Если сейчас загрузить HTML-документ (spa/spa.html) в браузер, то на консоли JavaScript не должно быть никаких ошибок. Мы должны увидеть верх окна чата. Но если щелкнуть по нему, то на консоли появится сообщение об ошибке вида «*set_chat_anchor is not a function*». Мы исправим эту ошибку, когда перейдем к модификации модуля Shell.

4.4.3. Модификация модуля Shell

Осталось внести изменения в модуль Shell. Вот что мы хотим сделать:

- убрать большую часть параметров и функций выплывающего чата, потому что они были перемещены в модуль Chat;
- исправить обработчик события `onHashchange`, так чтобы он восстанавливал правильное состояние, если не может перевести окно чата в требуемое состояние;
- добавить метод `setChatAnchor`, описанный в API;
- поправить документацию по методу `initModule`;
- изменить метод `initModule`, так чтобы он конфигурировал Chat в соответствии со спецификацией API.

В листинге 4.16 **полужирным** шрифтом показаны изменения в модуле Shell. Мы скопировали в этот файл написанные ранее спецификации API и пользовались ими как руководством в ходе кодирования.

Листинг 4.16 ❖ Модификация модуля Shell – spa/js/spa.shell.js

```
/*
 * spa.shell.js
 * Модуль Shell для SPA
 */

/*jslint      browser : true,   continue : true,
   devel      : true,   indent   : 2,     maxerr   : 50,
   newcap     : true,   nomen    : true,    plusplus : true,
   regexp     : true,   sloppy   : true,     vars     : false,
   white      : true
 */

/*global $, spa */

spa.shell = (function () {
    //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
    var
        configMap = {
            anchor_schema_map : {
                chat : { opened : true, closed : true }
            },
            main_html : String()
                + '<div class="spa-shell-head">'
                + '<div class="spa-shell-head-logo"></div>'
                + '<div class="spa-shell-head-acct"></div>'
                + '<div class="spa-shell-head-search"></div>'
                + '</div>'
                + '<div class="spa-shell-main">'
                + '<div class="spa-shell-main-nav"></div>'
                + '<div class="spa-shell-main-content"></div>'
                + '</div>'
                + '<div class="spa-shell-foot"></div>'
                + '<div class="spa-shell-modal"></div>'
        },
        stateMap = { anchor_map : {} },
        jqueryMap = {},

        copyAnchorMap,    setJqueryMap,
        changeAnchorPart, onHashchange,
        setChatAnchor,    initModule;

    //----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

    //----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
    // Возвращает копию сохраненного хэша якорей; минимизация издержек
```

Изменяем названия состояний на **opened** и **closed**, чтобы они были одинаковы в модулях Chat и Shell.

Удаляем HTML-разметку и параметры окна чата.

Удаляем **toggleChat** из списка переменных в области видимости модуля.

```

copyAnchorMap = function () {
    return $.extend( true, {}, stateMap.anchor_map );
};
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

//----- НАЧАЛО МЕТОДОВ DOM -----
// Начало метода DOM /setJqueryMap/
setJqueryMap = function () {
    var $container = stateMap.$container;
    jqueryMap = { $container : $container };
};
// Конец метода DOM /setJqueryMap/

// Начало метода DOM /changeAnchorPart/
// Назначение: изменяет якорь в URI-адресе
// Аргументы:
// * arg_map - хэш, описывающий, какую часть якоря
// мы хотим изменить.
// Возвращает: булево значение
// * true - якорь в URI обновлен
// * false - не удалось обновить якорь в URI
// Действие:
// Текущая часть якоря сохранена в stateMap.anchor_map.
// Обсуждение кодировки см. в документации по uriAnchor.
// Этот метод
// * создает копию хэша, вызывая copyAnchorMap().
// * Модифицирует пары ключ-значение с помощью arg_map.
// * Управляет различием между зависимыми и независимыми
// значениями в кодировке.
// * Пытается изменить URI, используя uriAnchor.
// * Возвращает true в случае успеха и false - в случае ошибки.
//
changeAnchorPart = function ( arg_map ) {
    var
        anchor_map_revise = copyAnchorMap(),
        bool_return = true,
        key_name, key_name_dep;

    // Начало объединения изменений в хэше якорей
    KEYVAL:
    for ( key_name in arg_map ) {
        if ( arg_map.hasOwnProperty( key_name ) ) {

            // пропустить зависимые ключи
            if ( key_name.indexOf( '_' ) === 0 ) { continue KEYVAL; }

            // обновить значение независимого ключа
            anchor_map_revise[key_name] = arg_map[key_name];

            // обновить соответствующий зависимый ключ

```

Удаляем метод toggleChat. Удаляем элемент Chat из jqueryMap.

```

    key_name_dep = '_' + key_name;
    if ( arg_map[key_name_dep] ) {
        anchor_map_revise[key_name_dep] = arg_map[key_name_dep];
    }
    else {
        delete anchor_map_revise[key_name_dep];
        delete anchor_map_revise['_s' + key_name_dep];
    }
}
}
// Конец объединения изменений в хэше якорей

// Начало попытки обновления URI; в случае ошибки восстановить
// исходное состояние
try {
    $.uriAnchor.setAnchor( anchor_map_revise );
}
catch ( error ) {
    // восстановить исходное состояние в URI
    $.uriAnchor.setAnchor( stateMap.anchor_map,null,true );
    bool_return = false;
}
// Конец попытки обновления URI...

return bool_return;
};
// Конец метода DOM /changeAnchorPart/
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
// Начало обработчика события /onHashchange/
// Назначение: обрабатывает событие hashchange
// Аргументы:
// * event – объект события jQuery.
// Параметры: нет
// Возвращает: false
// Действие:
// * Разбирает якорь в URI
// * Сравнивает предложенное состояние приложения с текущим
// * Вносит изменения, только если предложенное состояние
//   отличается от текущего
//
onHashchange = function ( event ) {
    var
        _s_chat_previous, _s_chat_proposed, s_chat_proposed,
        anchor_map_proposed,
        is_ok = true,
        anchor_map_previous = copyAnchorMap();

    // пытаемся разобрать якорь

```



```

try { anchor_map_proposed = $.uriAnchor.makeAnchorMap(); }
catch ( error ) {
    $.uriAnchor.setAnchor( anchor_map_previous, null, true );
    return false;
}
stateMap.anchor_map = anchor_map_proposed;

// вспомогательные переменные
_s_chat_previous = anchor_map_previous._s_chat;
_s_chat_proposed = anchor_map_proposed._s_chat;

// Начало изменения компонента чат
if ( ! anchor_map_previous
    || _s_chat_previous !== _s_chat_proposed
) {
    s_chat_proposed = anchor_map_proposed.chat;
    switch ( s_chat_proposed ) {
        case 'opened' :
            is_ok = spa.chat.setSliderPosition( 'opened' );
            break;
        case 'closed' :
            is_ok = spa.chat.setSliderPosition( 'closed' );
            break;
        default :
            toggleChat( false );
            delete anchor_map_proposed.chat;
            $.uriAnchor.setAnchor( anchor_map_proposed, null, true );
    }
}
// Конец изменения компонента чат

// Начало восстановления якоря, если не удалось
// изменить состояние окна чата
if ( ! is_ok ){
    if ( anchor_map_previous ){
        $.uriAnchor.setAnchor( anchor_map_previous, null, true );
        stateMap.anchor_map = anchor_map_previous;
    } else {
        delete anchor_map_proposed.chat;
        $.uriAnchor.setAnchor( anchor_map_proposed, null, true );
    }
}
// Конец восстановления якоря, если не удалось изменить
// состояние окна чата

return false;
};
// Конец обработчика события /onHashchange/
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

```

Используем открытый метод `setSliderPosition` модуля Chat.

Чистим параметр `chat` в якорь URL, если указанное состояние недопустимо, и возвращаемся в состояние по умолчанию. Протестировать это поведение можно, вручную добавив в якорь строку `#!chat=fred`.

Корректно обрабатываем случай, когда `setSliderPosition` вернул `false` (то есть перейти в запрошенное состояние не удалось). Либо восстанавливаем предыдущее состояние в якорь, либо, если такого не существует, используем состояние по умолчанию.

```
//----- НАЧАЛО ОБРАТНЫХ ВЫЗОВОВ -----
// Начало метода обратного вызова /setChatAnchor/ ←
// Пример: setChatAnchor( 'closed' );
// Назначение: изменить компонент якоря, относящийся к чату
// Аргументы:
// * position_type - допустимы значения 'closed' или 'opened'
// Действие:
// * Заменяет параметр 'chat' в яоре указанным значением,
// * если это возможно.
// Возвращает:
// * true - часть якоря была обновлена
// * false - часть якоря не была обновлена
// Исключения: нет
//
setChatAnchor = function ( position_type ){
    return changeAnchorPart({ chat : position_type });
};
// Конец метода обратного вызова /setChatAnchor/
//----- КОНЕЦ ОБРАТНЫХ ВЫЗОВОВ -----
```

Создаем метод обратного вызова setChatAnchor. Он передается модулю Chat в качестве безопасного способа запросить изменение в URI.

```
//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /initModule/ ←
// Пример: spa.chat.initModule( $('#div_id') );
// Назначение:
// * Требуется, чтобы Chat начал предоставлять свою
// * функциональность пользователю
// Аргументы:
// * $append_target (example: $('#div_id')).
// * Коллекция jQuery, которая должна содержать
// * единственный элемент DOM - контейнер
// Действие:
// * Добавляет выплывающий чат в конец указанного контейнера и заполняет
// * его HTML-содержимым. Затем инициализирует элементы, события и
// * обработчики, так чтобы предоставить пользователю интерфейс для работы
// * с чатом.
// Возвращает: true в случае успеха, иначе false
// Исключения: нет
//
initModule = function ( $container ) {
    // загружаем HTML и кэшируем коллекции jQuery
    stateMap.$container = $container;
    $container.html( configMap.main_html );
    setJqueryMap();

    // настраиваем uriAnchor на использование нашей схемы
    $.uriAnchor.configModule({
        schema_map : configMap.anchor_schema_map
    });

    // конфигурируем и инициализируем функциональные модули
```

Документируем метод initModule.

```

spa.chat.configModule({ ←
    set_chat_anchor : setChatAnchor,    Заменяем непосредственную привязку обработчика
    chat_model      : spa.model.chat,   щелчка по окну чата вызовом методов конфигурирова-
    people_model    : spa.model.people  ния и инициализации модуля Chat.
});
spa.chat.initModule( jQueryMap.$container );

// Обработываем события изменения якоря в URI.
// Это делается /после/ того, как все функциональные модули
// сконфигурированы и инициализированы, иначе они будут не готовы
// возбудить событие, которое используется, чтобы гарантировать
// учет якоря при загрузке
//
$(window)
    .bind( 'hashchange', onHashchange )
    .trigger( 'hashchange' );

};
// Конец открытого метода/initModule/

return { initModule : initModule };
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();

```

Теперь, открыв файл `spa/spa.html` в браузере, мы увидим картину, изображенную на рис. 4.13. Полагаем, что переработанное окно чата выглядит более привлекательно. Сообщений в нем пока нет – эту функциональность мы добавим в главе 6.

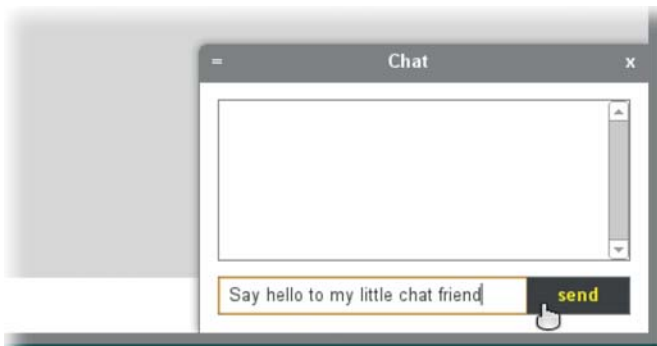


Рис. 4.13 ❖ Более привлекательное окно чата

Добившись правильной работы программы, проанализируем некоторые из ключевых изменений, проследив за тем, как выполняется наше приложение.

4.4.4. Прослеживание выполнения

В этом разделе уделяется внимание только что внесенным в приложение изменениям. Мы посмотрим, как приложение конфигурируется и инициализируется, а затем выясним, что происходит, когда пользователь щелкает мышью по окну чата.

После загрузки документа в браузер скрипт инициализирует корневое пространство имен (`spa/js/spa.js`), передавая ему элемент страницы (`div #spa`), который можно использовать:

```
$(function () { spa.initModule( $('#spa') ); });
```

После этого корневой модуль (`spa/js/spa.js`) инициализирует модуль Shell (`spa/js/spa.shell.js`), тоже передавая ему элемент страницы (`$container`):

```
var initModule = function ( $container ){
    spa.shell.initModule( $container );
};
```

Затем Shell конфигурирует и инициализирует модуль Chat (`spa/js/spa.chat.js`). Но на этот раз шаги несколько отличаются. Теперь конфигурирование производится в соответствии с определенными нами API. Согласно спецификации, конфигурационный параметр `set_chat_anchor` — это обратный вызов:

```
...
// конфигурируем и инициализируем функциональные модули
spa.chat.configModule({
    set_chat_anchor : setChatAnchor,
    chat_model      : spa.model.chat,
    people_model    : spa.model.people
});
spa.chat.initModule(jQueryMap.$container);
...
```

Инициализация модуля Chat имеет одно тонкое отличие от того, что мы видели ранее: вместо контейнера, который может использовать модуль, Shell теперь предоставляет контейнер, в который Chat должен *добавить* всплывающий чат. Это удобное соглашение, если вы доверяете автору модуля. А мы доверяем.

```
...
// * set_chat_anchor - обратный вызов для модификации якоря в URI,
//   чтобы отразить состояние: открыт или закрыт. Обратный вызов должен
//   возвращать false, если установить указанное состояние невозможно.
...
```

Когда пользователь щелкает по кнопке переключения состояния, Chat обращается к обратному вызову `set_chat_anchor` с требованием изменить значение параметра `chat` в якорь, сделав его равным *opened* или *closed*, а затем возвращает управление. События `hashchange` по-прежнему обрабатывает модуль Shell, как видно из файла `spa/js/spa.shell.js`:

```
initModule = function ( $container ){  
    ...  
    $(window)  
        .bind( 'hashchange', onHashchange )  
    ...  
}
```

Итак, после щелчка по окну чата Shell перехватывает событие `hashchange` и вызывает его обработчик `onHashchange`. Если компонент `chat` в якорь изменился, то этот обработчик вызывает метод `spa.chat.setSliderPosition`, требуя установить новое состояние:

```
// Начало изменения компонента Chat  
if ( ! anchor_map_previous  
    || _s_chat_previous !== _s_chat_proposed  
) {  
    s_chat_proposed = anchor_map_proposed.chat;  
    switch ( s_chat_proposed ) {  
        case 'opened' :  
            is_ok = spa.chat.setSliderPosition( 'opened' );  
            break;  
        case 'closed' :  
            is_ok = spa.chat.setSliderPosition( 'closed' );  
            break;  
        ...  
    }  
}  
// Конец изменения компонента Chat
```

Если указанное состояние допустимо, то окно чата переходит в него, и параметр `chat` в URI изменяется.

В результате внесенных изменений мы получили реализацию, удовлетворяющую проектным целям. URI управляет состоянием выплывающего чата, а весь код и пользовательский интерфейс чата перенесен в новый функциональный модуль. Чат выглядит и работает лучше, чем раньше. Теперь добавим еще несколько открытых методов, которые часто находят применение в функциональных модулях.

4.5. Добавление часто используемых методов

Есть несколько открытых методов, которые бывают нужны в функциональных модулях настолько часто, что заслуживают отдельного рассмотрения. Это метод удаления (`removeSlider`) и метод изменения размера окна (`handleResize`). Мы реализуем оба. Сначала объявим имена методов в модуле `Chat` – в конце секции «Переменные в области видимости модуля», а затем экспортируем их как открытые методы в самом конце модуля (листинг 4.17). Изменения выделены полужирным шрифтом.

Листинг 4.17 ❖ Объявление имен методов – `spa/js/spa.chat.js`

```
...
  jqueryMap = {},

  setJqueryMap, getEmSize, setPxSizes, setSliderPosition,
  onClickToggle, configModule, initModule,
  removeSlider, handleResize
;
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
...

// возвращаем открытые методы
return {
  setSliderPosition : setSliderPosition,
  configModule      : configModule,
  initModule        : initModule,
  removeSlider      : removeSlider,
  handleResize      : handleResize
};
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
}());
```

Объявив имена методов, реализуем их в следующих разделах.

4.5.1. Метод `removeSlider`

Как обнаружилось, метод `remove` бывает нужен во многих функциональных модулях. Например, если мы реализуем аутентификацию, то после выхода пользователя из системы выплывающий чат, возможно, должен быть вообще удален. Обычно такие действия предпринимают, чтобы повысить производительность или обеспечить безопасность – предполагая, что метод `remove` успешно справляется с задачей удаления ненужных структур данных.

Наш метод должен удалить контейнер DOM, который добавил метод Chat, и вообще *откатить* все сделанное во время инициализации и конфигурирования – именно в таком порядке. Метод `removeSlider` приведен в листинге 4.18. Изменения выделены **полужирным** шрифтом.

Листинг 4.18 ❖ Метод `removeSlider` – `spa/js/spa.chat.js`

```
...
// Конец открытого метода /initModule/

// Начало открытого метода /removeSlider/
// Назначение:
// * удаляет из DOM элемент chatSlider
// * возвращает в исходное состояние
// * удаляет указатели на методы обратного вызова и другие данные
//Аргументы: нет
// Возвращает: true
// Исключения: нет
//
removeSlider = function () {
    // откатить инициализацию и стереть состояние
    // удалить из DOM контейнер; при этом удаляются и привязки событий
    if ( jqueryMap.$slider ) {
        jqueryMap.$slider.remove();
        jqueryMap = {};
    }
    stateMap.$append_target = null;
    stateMap.position_type = 'closed';

    // стереть значения ключей
    configMap.chat_model = null;
    configMap.people_model = null;
    configMap.set_chat_anchor = null;

    return true;
};
// Конец открытого метода /removeSlider/

// возвращаем открытые методы
...
```

Ни в одном методе `remove` мы не пытаемся особенно «умничать». Задача состоит в том, чтобы не оставить и следа от ранее выполненных конфигурирования и инициализации, – вот и все. Мы тщательно стираем все указатели на данные. Это необходимо для того, чтобы счетчики ссылок на структуры данных обратились в нуль, после чего в дело может вступить сборщик мусора. *В частности, по этой при-*

чине мы всегда перечисляем возможные ключи в `configMap` и `stateMap` в начале модуля – чтобы видеть, что именно нужно очистить.

Протестировать метод `removeSlider` можно, введя на консоли JavaScript в инструментах разработчика такое предложение (не забудьте нажать **Enter!**):

```
spa.chat.removeSlider();
```

Проинспектировав окно браузера, мы увидим, что выплывающего чата не стало. Чтобы вернуть его назад, нужно ввести на консоли такие предложения:

```
spa.chat.configModule({ set_chat_anchor: function () { return true; } });
spa.chat.initModule( $( '#spa' ) );
```

Выплывающий чат, «воскрешенный» с помощью консоли JavaScript, работает не в полном объеме, потому что в качестве обратного вызова `set_chat_anchor` мы указали функцию, которая не делает ничего полезного. По-настоящему восстанавливать модуль чата нужно только из Shell, где имеется доступ к необходимой функции обратного вызова.

Мы могли бы сделать в этом методе гораздо больше, например заставить окно чата исчезать постепенно, но оставим это читателю в качестве упражнения. А сами реализуем другой метод, который часто бывает нужен в функциональных модулях, – `handleResize`.

4.5.2. Метод `handleResize`

Метод `handleResize` также полезен во многих функциональных модулях. При правильном использовании CSS большую часть содержимого SPA можно написать так, что она будет нормально выглядеть в окне любого разумного размера. Но иногда «большой части» недостаточно, и приходится производить пересчет. Сначала реализуем метод `handleResize`, как показано на рис. 4.19, а потом обсудим, что мы сделали. Изменения выделены **полужирным** шрифтом.

Листинг 4.19 ❖ Метод `handleResize` – `spa/js/spa.chat.js`

<pre>... configMap = { ... slider_opened_em : 18, ... slider_opened_min_em : 10, window_height_min_em : 20, ... }</pre>	<p>Немного увеличиваем высоту открытого окна чата.</p> <p>Добавляем конфигурационный параметр, определяющий минимальную высоту открытого окна чата.</p> <p>Добавляем конфигурационный параметр, равный пороговой высоте окна. Если высота окна оказывается меньше пороговой, мы должны минимизировать выплывающий чат. Если высота окна больше или равна пороговой, восстанавливается нормальная высота выплывающего чата.</p>
--	--


```

},
...

// Начало метода DOM /setPxSizes/
setPxSizes = function () {
    var px_per_em, window_height_em, opened_height_em;

    px_per_em = getEmSize( jqueryMap.$slider.get(0) );
    window_height_em = Math.floor( ←———— Вычисляем высоту окна в единицах em.
        ( $(window).height() / px_per_em ) + 0.5
    );
    opened_height_em ←———— Это «секрет фирмы», где мы определяем высоту от-
        = window_height_em > configMap.window_height_min_em
        ? configMap.slider_opened_em
        : configMap.slider_opened_min_em;

    stateMap.px_per_em = px_per_em;
    stateMap.slider_closed_px = configMap.slider_closed_em * px_per_em;
    stateMap.slider_opened_px = opened_height_em * px_per_em;
    jqueryMap.$sizer.css({
        height : ( opened_height_em - 2 ) * px_per_em
    });
};
// Конец метода DOM /setPxSizes/

...

// Начало открытого метода /handleResize/ ←———— Добавляем метод handleResize
// Назначение:                               вместе с документацией.
// В ответ на событие изменения размера окна корректирует
// представление, формируемое данным модулем, если необходимо
// Действия:
// Если высота или ширина окна оказываются меньше заданного
// порога, изменить размер выплывающего чата в соответствии с
// уменьшившимся размером окна.
// Возвращает: булево значение
// * false - изменение размера не учтено
// * true  - изменение размера учтено
// Исключения: нет
//
// Пересчитываем размеры в пикселях
// при каждом вызове handleResize.
handleResize = function () {
    // ничего не делать, если выплывающего контейнера нет
    if ( ! jqueryMap.$slider ) { return false; }
    setPxSizes(); ←———— Гарантируем, что высота выплывающего
    if ( stateMap.position_type === 'opened' ){ ←———— чата установлена равной значению, вычис-
        jqueryMap.$slider.css({ height : stateMap.slider_opened_px });
    }
    ←———— ленному методом setPxSizes, если во
    ←———— время изменения размера чат был открыт.
}

```

```

    return true;
};

// Конец открытого метода /handleResize/

// возвращаем открытые методы
...

```

Метод `handleResize` не вызывает сам себя. Сейчас может появиться искушение реализовать обработчик события `window.resize` для каждого функционального модуля, но это *неудачная мысль*. Беда в том, что частота генерации события `window.resize` сильно зависит от браузера. Допустим, у нас есть пять функциональных модулей, в каждом из которых имеется обработчик события `window.resize`, и пользователь изменяет размер окна браузера. Если событие `window.resize` генерируется каждые 10 миллисекунд, а необходимые изменения изображения достаточно сложны, то так мы можем поставить наше SPA – а возможно, и весь браузер, а то и ОС – на колени.

Гораздо правильнее поручить обработчику события в Shell перехватывать события изменения размера и затем вызывать методы `handleResize` всех подчиненных функциональных модулей. Это позволяет регулировать скорость обработки событий изменения размера и осуществлять диспетчеризацию из единственного обработчика события. Реализуем эту стратегию в Shell, как показано в листинге 4.20. Изменения выделены **полужирным** шрифтом.

Листинг 4.20 ❖ Добавление обработчика события `onResize` – `spa/js/spa.shell.js`

```

...
//----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
var
  configMap = {
    ...
    resize_interval : 200, ← Интервал длительностью 200 миллисекунд
    ...                               для учета события изменения размера.
  },
  stateMap = {
    $container : undefined,
    anchor_map : {},
    resize_idto : undefined ← Заводим переменную, в которой будет храниться иденти-
                                фикатор тайм-аута (подробности см. ниже).
  },
  jqueryMap = {},
  copyAnchorMap, setJqueryMap,
  changeAnchorPart, onHashchange, onResize,
  setChatAnchor, initModule;
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

```

```

...

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
...
// Начало обработчика события /onResize/
onResize = function (){
    if ( stateMap.resize_idto ){ return true; }
    spa.chat.handleResize();
    stateMap.resize_idto = setTimeout(
        function (){ stateMap.resize_idto = undefined; },
        configMap.resize_interval
    );
    return true;
};
// Конец обработчика события /onResize/
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

...
initModule = function (){
    ...
    $(window)
        .bind( 'resize', onResize )
        .bind( 'hashchange', onHashchange )
        .trigger( 'hashchange' );
    };
// Конец открытого метода /initModule/

```

Выполняем логику onResize, только если сейчас не запущен таймер изменения размера.

Возвращаем true из обработчика события window.resize, чтобы jQuery не вызывала preventDefault() или stopPropagation().

Функция обработки тайм-аута стирает идентификатор сработавшего тайм-аута, поэтому через каждые 200 миллисекунд в процессе изменения размера переменная stateMap.resize_idto оказывается равна undefined, и тогда этот метод может быть выполнен в полном объеме.

Привязываем событие window.resize.

Мы хотим подправить таблицу стилей, чтобы лучше видеть плоды своего труда. В листинге 4.21 мы уменьшили минимальный размер окна, перешли к относительным единицам измерения и убрали никому не нужную рамку вокруг содержимого. Изменения выделены **полужирным шрифтом**.

Листинг 4.21 ❖ Изменение стилей для большей наглядности onResize – spa/css/spa.css

```

/** Начало установки начальных значений */
* {
    margin : 0;
    padding : 0;
    -webkit-box-sizing : border-box;
    -moz-box-sizing : border-box;
    box-sizing : border-box;
}
h1,h2,h3,h4,h5,h6,p { margin-bottom : 6pt; }
ol,ul,dl { list-style-position : inside;}
/** Конец установки начальных значений */

/** Начало стандартных селекторов */

```

Переходим к относительным единицам измерения полей (пункты).

```

body {
    font : 10pt 'Trebuchet MS', Verdana, Helvetica, Arial, sans-serif;
    ...
}
/** Конец стандартных селекторов */

/** Начало селекторов в пространстве имен spa */
#spa {
    position : absolute;
    top : 0;
    left : 0;
    bottom : 0;
    right : 0;
    background : #fff;
    min-height : 15em;
    min-width : 35em;
    overflow : hidden;
}
/** Конец селекторов в пространстве имен spa */

/** Начало служебных селекторов */
...

```

Переходим к относительным единицам измерения шрифта (пункты).

Удаляем 8-пиксельное смещение от границ div #spa. В результате div со всех сторон примыкает вплотную к границам окна.

Существенно уменьшаем минимальную ширину и высоту div'a #spa. Переходим к относительным единицам измерения (em).

Удаляем рамку, поскольку она больше не нужна.

Чтобы понаблюдать, как работает событие изменения размера, откройте документ `spa/spa.html` в браузере, а затем увеличьте или уменьшите высоту окна браузера. На рис. 4.14 показано, как выглядит выплывающий чат до и после достижения порога.



Рис. 4.14 ❖ Размеры выплывающего чата до и после достижения пороговой высоты

Разумеется, всегда есть что улучшить. Было бы неплохо оставить какое-то минимальное расстояние между выплывающим чатом и верхней границей окна. Например, если высота окна превышает порог на 0,5 em, то окно чата можно было бы сделать точно на 0,5 em

короче, чем обычно. Это позволило бы оптимально использовать место для чата и плавнее производить корректировку во время изменения размера. Несложную реализацию оставляем читателю в качестве упражнения.

4.6. Резюме

В этой главе мы показали, как можно применить в функциональных модулях достоинства сторонних моделей, обойдя их недостатки. Мы определили, что такое функциональные модули, сравнили их со сторонними модулями и обсудили их место в нашей архитектуре. Мы продемонстрировали, что наше приложение, как и большинство сайтов, содержит фрактальное повторение паттернов MVC, и показали, как это проявляется в функциональных модулях. Затем мы создали функциональный модуль, отталкиваясь от кода, написанного в главе 3. На первом проходе мы добавили все необходимые файлы, включив в них лишь простейшую реализацию. Затем мы спроектировали API и на втором проходе реализовали его. Наконец, мы добавили несколько методов, часто применяемых в функциональных модулях, и подробно объяснили, как ими пользоваться.

Настало время централизовать бизнес-логику в модели. В следующих главах мы разработаем модель и покажем, как облечь в плоть и кровь логику работы с пользователями, людьми и чатом. Мы воспользуемся событиями jQuery для активации изменений DOM, вместо того чтобы полагаться на ненадежные обратные вызовы. Кроме того, мы имитируем «живой» сеанс работы в чате. Оставайтесь с нами – мы как раз собираемся превратить наше SPA из забавной демонстрации в почти готовое клиентское приложение.

Глава 5

Построение модели

В этой главе:

- ✧ Определение модели и ее места в нашей архитектуре.
- ✧ Связи между модулями Model, Data и Fake.
- ✧ Подготовка файлов для модели.
- ✧ Поддержка сенсорных устройств.
- ✧ Проектирование объекта `people`.
- ✧ Реализация объекта `people` и тестирование API.
- ✧ Добавление в Shell поддержки аутентификации и завершения сеанса.

В этой главе мы продолжаем работу, начатую в главах 3 и 4. Перед тем как приступить к ее чтению, вы должны иметь файлы из проекта, созданного в главе 4, потому что мы будем развивать их. Мы рекомендуем целиком скопировать созданное в главе 4 дерево каталогов со всеми файлами в новый каталог «chapter_5» и там уже вносить изменения.

В этой главе мы спроектируем и реализуем часть модели – объект `people`. Модель предоставляет модуль Shell и функциональным модулям бизнес-логику и данные. Модель не зависит от пользовательского интерфейса и изолирует его от логики и управления данными. Сама модель изолирована от веб-сервера посредством модуля `Data`.

Мы хотим использовать в нашем SPA объект `people` для управления списком людей, включающим как текущего пользователя, так и тех, с кем он чатится. Создав и протестировав модель, мы изменим модуль Shell, так чтобы пользователь мог аутентифицироваться и завершить сеанс. Попутно мы добавим сенсорные элементы управления, чтобы нашим SPA можно было пользоваться на смартфоне или планшете. Но для начала разберемся, что делает модель и какое место она занимает в нашей архитектуре.

5.1. Что такое модель

В главе 3 мы ввели в рассмотрение модуль Shell, который отвечает за задачи, свойственные приложению в целом, например за управление якорем в URI и макет интерфейса. Узкоспециальные задачи Shell по-

ручает тщательно изолированным функциональным модулям, с которыми мы познакомились в главе 4. У каждого такого модуля есть свои представление, контроллер и часть модели, разделяемая с Shell. Общий вид этой архитектуры показан на рис. 5.1¹.

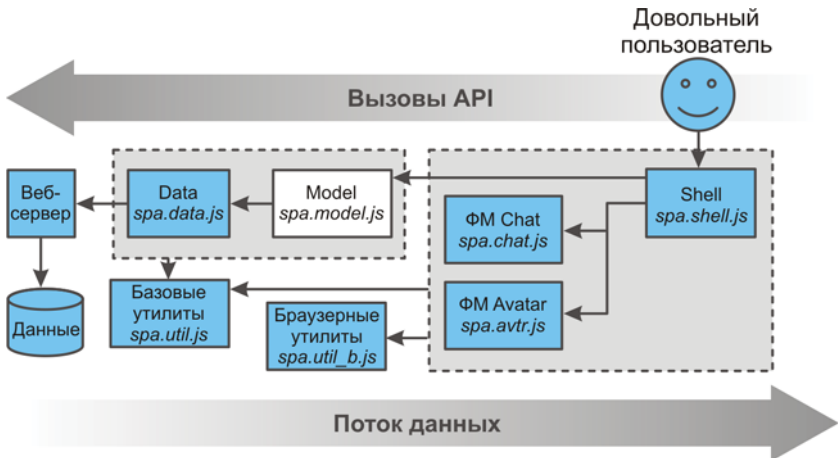


Рис. 5.1 ❖ Место модуля Model в архитектуре SPA

Модель консолидирует всю бизнес-логику и данные в одном пространстве имен. Ни Shell, ни функциональные модули не взаимодействуют с веб-сервером непосредственно, они имеют дело только с модулем Model. Модель и сама изолирована от веб-сервера благодаря модулю Data. Такая изоляция позволяет вести разработку быстрее и качественнее, в чем мы скоро убедимся.

Мы начнем изложение с вопроса о разработке и использовании модели. Эту работу мы завершим в главе 6. Рассмотрим, чего мы собираемся достичь в этих двух главах и что в связи с этим нам потребуется от модели.

5.1.1. Что мы собираемся сделать

Прежде чем переходить к обсуждению модели, полезно будет обратиться к примеру приложения. На рис. 5.2 показано, какую функциональность мы собираемся добавить в наше SPA к концу главы 6.

¹ Группы модулей, пользующиеся общими утилитами, обведены пунктирной линией. Например, модули Chat, Avatar и Shell пользуются «браузерными утилитами» и «базовыми утилитами», а модули Data и Model – только «базовыми утилитами».

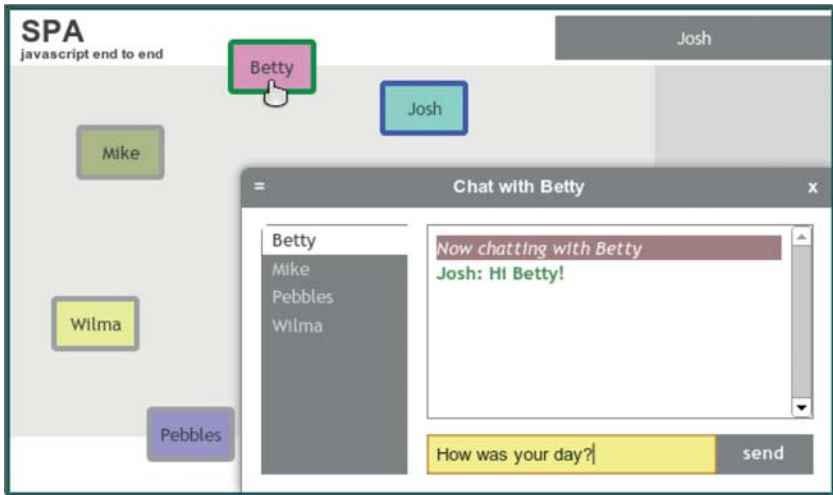


Рис. 5.2 ❖ Как будет выглядеть наше SPA в недалеком будущем

Модуль Shell будет управлять процессом аутентификации – в правом верхнем углу мы видим имя аутентифицированного пользователя. Функциональный модуль Chat будет управлять окном чата, которое находится справа внизу. А функциональный модуль *Avatar* будет отвечать за цветные блоки, представляющие людей. Рассмотрим, какая бизнес-логика и данные потребуются в каждом модуле.

- Модулю Shell нужно представление текущего пользователя, чтобы управлять процессом входа и выхода. Ему понадобятся методы для определения личности текущего пользователя и, возможно, смены пользователя.
- Функциональному модулю Chat также необходима возможность проверить, имеет ли текущий пользователь (в данном случае Josh) право отправлять и получать сообщения. Нужно также уметь определять, с кем общается пользователь, – если он вообще с кем-то общается. Нужно уметь запрашивать список людей в онлайн, чтобы можно было показать его в левой части окна чата. Наконец, нужны методы для отправки сообщений и выбора собеседника.
- Функциональный модуль Avatar также должен уметь опрашивать текущего пользователя (Josh), чтобы узнать, имеет ли он право видеть другие аватары и взаимодействовать с ними. Ему также необходимо идентифицировать текущего пользовате-

ля, чтобы можно было обвести его аватар синей рамкой. Еще он должен знать, с кем сейчас общается пользователь (Betty), чтобы обвести аватар собеседника зеленой рамкой. Наконец, ему необходимы методы для получения подробных сведений об аватарах (например, цвет и положение) всех находящихся в онлайн.

Как видим, потребности наших модулей в бизнес-логике и данных значительно перекрываются. Так, объект, представляющий текущего пользователя, нужен модулям Shell, Chat и Avatar, а список людей в онлайн – модулям Chat и Avatar. На ум приходит несколько способов управиться с перекрытием:

- встроить необходимую логику и данные в каждый функциональный модуль;
- разместить части логики и данных в разных функциональных модулях. Например, можно было бы считать, что Chat – владелец объекта `people`, а Avatar – владелец объекта `chat`. Для обмена информацией можно было бы вызывать методы одного модуля из другого;
- построить централизованную модель, в которой будут консолидированы логика и данные.

Первый вариант – сопровождение параллельных структур данных и методов в разных модулях – настолько чреват ошибками и трудоемок, что даже смешно его рассматривать. Уж лучше сделать увлекательную карьеру в сфере поджаривания бургеров. По мне так хорошо бы с картошечкой.

Второй вариант выглядит получше, но ненамного. Когда логика и данные достигнут определенного – не слишком большого – уровня сложности, количество межмодульных зависимостей возрастет настолько, что получится кошмарный код в виде SPA-getti.

Наш опыт показывает, что последний вариант, использование модели, намного превосходит остальные, причем некоторые его достоинства становятся очевидны не сразу. Посмотрим, как должна выглядеть хорошо написанная модель.

5.1.2. Что делает модель

Модель – это то место, куда Shell и все функциональные модули ходят за данными и бизнес-логикой. Если нам нужно аутентифицировать пользователя, то мы вызываем метод, предоставляемый моделью. Желая получить список людей, мы обращаемся к модели. Нужны сведения об аватаре... ну, в общем, вы поняли. Любые данные

или логика, которые должны совместно использоваться несколькими функциональными модулями или принадлежат приложению в целом, должны быть частью модели. Если вы свободно владеете архитектурой модель–представление–контроллер (MVC), то идея модели не должна вызвать никаких затруднений.

Но из того, что вся бизнес-логика и данные доступны только через модель, вовсе не следует, что реализация должна содержаться в одном (потенциально гигантском) JavaScript-файле. Мы вполне можем воспользоваться пространствами имен для разбиения модели на более податливые части. Например, если в модели есть объекты `people` и `chat`, то логику `people` можно поместить в файл `spa.model.people.js`, а логику `chat` – в файл `spa.model.chat.js`, которые затем консолидировать в главном файле модели `spa.model.js`. Такая техника позволяет сохранить интерфейс, предоставляемый модулю Shell, неизменным вне зависимости от того, сколько файлов входит в состав модели.

5.1.3. Чего модель не делает

Модели не нужен браузер. Это означает, что модель не должна предполагать наличия объекта `document` или таких специфичных для браузера методов, как `document.location`. При использовании паттерна MVC считается хорошим стилем поручать отрисовку представления данных модели модулю Shell и (особенно) функциональным модулям. Такое разделение обязанностей существенно упрощает автоматизацию автономного и регрессионного тестирования. Наша практика свидетельствует о том, что когда дело доходит до взаимодействия с браузером, ценность автоматизированного тестирования резко уменьшается с ростом стоимости реализации. Но, избегая DOM, мы можем протестировать все вплоть до пользовательского интерфейса, не прибегая к запуску браузера.

Автономное и регрессионное тестирование

Коллективы разработчиков должны решить, когда следует инвестировать в автоматизированное тестирование. Автоматизация тестов API модели – всегда хорошее дело, потому что тесты можно изолировать так, что для всех вызовов API будут использоваться одни и те же данные. Автоматизация тестов пользовательского интерфейса обходится гораздо дороже из-за большого числа переменных, которые трудно проконтролировать или предсказать. Например, сложно и дорого имитировать темп нажатия одной кнопки пользователем, или спрогнозировать поток данных через систему при участии пользователя, или узнать, насколько быстро работает сеть. Поэтому веб-страницы часто тестируются вручную с применением некоторых вспомогательных инструментов типа валидаторов HTML и программ проверки ссылок.

В хорошо спроектированном SPA имеются независимые уровни данных, модели и функциональных модулей (представление + контроллер). Мы позаботимся о том, чтобы модули данных (Data) и модели (Model) имели четко определенные API и были изолированы от функциональных модулей. Поэтому для тестирования этих уровней браузер нам не понадобится, и, стало быть, мы сможем без особых затрат применить автоматизированное и регрессионное тестирование с помощью таких сред выполнения JavaScript, как Node.js или написанный на Java каркас Rhino. С другой стороны, по нашему опыту, уровни представления и контроллера лучше тестировать вручную с привлечением реальных людей.

Модель не предоставляет никаких универсальных утилит. Вместо этого мы пользуемся общей служебной библиотекой (spa/js/spa.util.js), которая не нуждается в DOM. Эти утилиты включены в отдельный пакет, потому что могут применяться в разных SPA. С другой стороны, модель часто бывает «заточена» под конкретное SPA.

Модель не взаимодействует напрямую с сервером. Для этого есть отдельный модуль Data, который отвечает за получение всех необходимых модели данных от сервера.

Теперь, когда мы лучше понимаем роль модели в архитектуре, займемся подготовкой файлов, которые понадобятся в этой главе.

5.2. Подготовка файлов модели, и не только

Нам предстоит добавить и модифицировать ряд файлов, чтобы поддерживать работу с моделью. Заодно мы добавим файлы для функционального модуля Avatar, поскольку скоро они понадобятся.

5.2.1. Планируем структуру каталогов и файлов

Мы рекомендуем скопировать всю созданную в главе 4 структуру каталогов в новый каталог «chapter_5» и там вносить изменения. Структура, на которой мы остановились в главе 4, показана в листинге 5.1:

Листинг 5.1 ❖ Структура каталогов и файлов из главы 4

```
spa
+-- css
| +-- spa.chat.css
| +-- spa.css
| '-- spa.shell.css
+-- js
| +-- jq
| | +-- jquery-1.9.1.js
| | '-- jquery.uriAnchor-1.1.3.js
```

```
| +-- spa.js
| +-- spa.chat.js
| +-- spa.model.js
| +-- spa.shell.js
| `-- spa.util.js
|-- spa.html
```

А вот какие изменения мы планируем внести:

- *создать* для Avatar таблицу стилей в пространстве имен;
- *модифицировать* таблицу стилей для Shell с целью поддержать аутентификацию пользователей;
- *включить* подключаемый модуль jQuery, обеспечивающий унификацию ввода с помощью мыши и сенсорного экрана;
- *включить* подключаемый модуль jQuery для поддержки глобальных пользовательских событий;
- *включить* JavaScript-библиотеку для работы с базой данных в браузере;
- *создать* для Avatar JavaScript-модуль в пространстве имен. Это будет заглушка в преддверии главы 6;
- *создать* модуль Data в пространстве имен. Он будет предоставлять интерфейс к «реальным» данным, хранящимся на сервере;
- *создать* модуль Fake в пространстве имен. Он будет предоставлять интерфейс к «подставным» данным, используемым для тестирования;
- *создать* в пространствах имен модули, содержащие общие утилиты для работы с браузером;
- *модифицировать* модуль Shell для поддержки аутентификации пользователей;
- *модифицировать* головной HTML-документ, включив в него новые файлы CSS и JavaScript.

Обновленная структура файлов и каталогов показана на рис. 5.2. Новые и измененные файлы выделены полужирным шрифтом.

Листинг 5.2 ❖ Обновленная структура файлов и каталогов

```
spa
+-- CSS
| +-- spa.avtr.css ← Добавляем таблицу стилей для Avatar.
| +-- spa.chat.css
| +-- spa.css
| `-- spa.shell.css ← Модифицируем таблицу стилей для Shell с целью поддержки
+-- js                               аутентификации.
| +-- jq
| | +-- jquery-1.9.1.js
| | +-- jquery.event.ue-0.3.2.js ← Включаем подключаемый модуль jQuery, обеспечивающий
|                                     унификацию ввода с помощью мыши и сенсорного экрана.
```

```

| | ++- jquery.event.gevent-0.1.9.js ← Включаем подключаемый модуль gevent, необходимый
| | ++- jquery.uriAnchor-1.1.3.js      для поддержки глобальных пользовательских событий.
| | '-- taffydb-2.6.2.js ← Включаем библиотеку TaffyDB для работы с базой дан-
| ++- spa.js                          ных в браузере.
| ++- spa.avtr.js ← Добавляем функциональный модуль Avatar.
| ++- spa.chat.js
| ++- spa.data.js ← Добавляем модуль Data.
| ++- spa.fake.js ← Добавляем модуль Fake.
| ++- spa.model.js
| ++- spa.shell.js
| ++- spa.util_b.js ← Добавляем браузерные утилиты.
| '-- spa.util.js
|-- spa.html ← Модифицируем Shell для поддержки аутентификации.

```

Решив, какие файлы нужно добавить и изменить, откроем текстовый редактор и приступим к делу. Мы будем рассматривать файлы в том порядке, в котором описывали их выше. Если вы прорабатываете материал дома, можете создавать файлы по ходу чтения.

5.2.2. Создание файлов

Первым рассмотрим файл `spa/css/spa.avtr.css`. Создайте его и заполните, как показано в листинге 5.3. Поначалу это будет *заглушка*.

Листинг 5.3 ❖ Наша таблица стилей для Avatar (заглушка) – `spa/css/spa.avtr.css`

```

/*
 * spa.avtr.css
 * Стили для функционального модуля Avatar
 */

```

Следующие три файла – это библиотеки. Загрузите их в каталог `spa/js/jq`.

Файл `spa/js/jq/jquery.event.ue-0.3.2.js` можно скачать со страницы <https://github.com/mmikowski/jquery.event.ue>. Это библиотека унифицированного ввода с помощью мыши и сенсорного экрана.

Файл `spa/js/jq/jquery.event.gevent-0.1.9.js` можно скачать со страницы <https://github.com/mmikowski/jquery.event.gevent>. Он необходим для работы с глобальными пользовательскими событиями.

Файл `spa/js/jq/taffydb-2.6.2.js` нужен для организации базы данных на стороне клиента. Его можно скачать со страницы <https://github.com/typicaljoe/taffydb>. Это не подключаемый модуль jQuery, и, если бы наш проект был масштабнее, мы поместили бы его в отдельный каталог `spa/js/lib`.

Следующие три JavaScript-файла – `spa/js/spa.avtr.js`, `spa/js/spa.data.js` и `spa/js/spa.fake.js` – пока являются заглушками. Их содержание показано в листингах 5.4, 5.5, 5.6. По существу, они идентичны –

в каждом имеется заголовок, за которым следуют параметры JSLint и объявление пространства имен, согласованное с именем файла. Уникальные части выделены **полужирным** шрифтом.

Листинг 5.4 ❖ Функциональный модуль Avatar – spa/js/spa.avtr.js

```
/*
 * spa.avtr.js
 * Функциональный модуль Avatar
 */

/*jslint      browser : true,   continue : true,
   devel  : true,   indent  : 2,     maxerr  : 50,
   newcap : true,   nomen   : true,   plusplus : true,
   regexp : true,   sloppy  : true,   vars    : false,
   white  : true
 */
/*global $, spa */
spa.avtr = (function () { return {}; }());
```

Листинг 5.5 ❖ Модуль Avatar – spa/js/spa.data.js

```
/*
 * spa.data.js
 * Модуль Data
 */

/*jslint      browser : true,   continue : true,
   devel  : true,   indent  : 2,     maxerr  : 50,
   newcap : true,   nomen   : true,   plusplus : true,
   regexp : true,   sloppy  : true,   vars    : false,
   white  : true
 */
/*global $, spa */
spa.data = (function () { return {}; }());
```

Листинг 5.6 ❖ Модуль Fake – spa/js/spa.fake.js

```
/*
 * spa.fake.js
 * Модуль Fake
 */

/*jslint      browser : true,   continue : true,
   devel  : true,   indent  : 2,     maxerr  : 50,
   newcap : true,   nomen   : true,   plusplus : true,
   regexp : true,   sloppy  : true,   vars    : false,
   white  : true
 */
/*global $, spa */
spa.fake = (function () { return {}; }());
```

Напомним, что секции `/*jslint ...*/` и `/*global ...*/` используются, когда мы запускаем JSLint для проверки кода на предмет на-

личия типичных ошибок. В секции `/*jslint ...*/` устанавливаются параметры проверки. Например, `browser : true` для валидатора JSLint означает, что нужно предполагать, что JavaScript работает в контексте браузера, поэтому объект `document` (и многие другие) существует. В секции `/*global $, spa */` мы сообщаем JSLint, что переменные `$` и `spa` определены вне этого модуля. Без этой информации валидатор стал бы жаловаться, что указанные переменные используются раньше, чем определены. Полный перечень параметров JSLint приведен в приложении А.

Далее мы можем добавить файл с браузерными утилитами `spa/js/spa.util_b.js`. Этот модуль предоставляет общеупотребительные функции, работающие только в среде браузера. Иными словами, браузерные утилиты не будут нормально работать с Node.js, тогда как стандартные утилиты (`spa/js/spa.util.js`) будут. На рис. 5.3 изображено место этого модуля в нашей архитектуре.

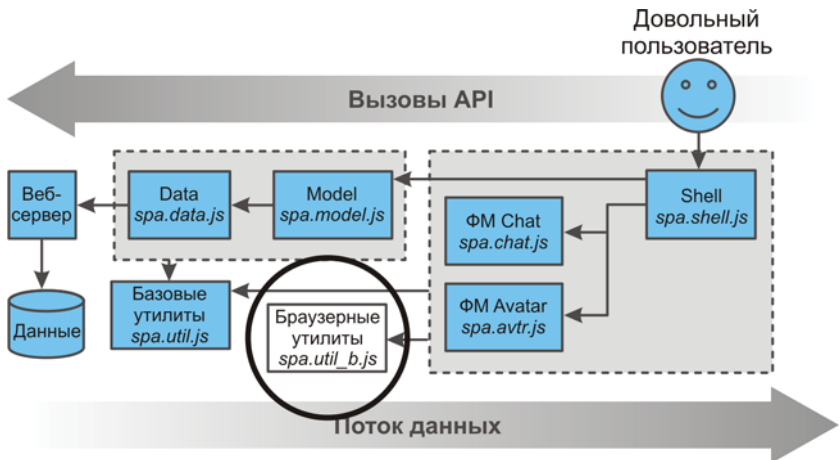


Рис. 5.3 ❖ Модуль браузерных утилит содержит функции, нуждающиеся в наличии браузера

В состав браузерных утилит мы включили функции `encodeHtml` и `decodeHtml`, которые, как следует из названий, применяются для кодирования и декодирования специальных символов HTML, например `&` или `<`¹. В этом же файле имеется функция `getEmSize`, которая вычис-

¹ Эти методы важны для предотвращения атак межсайтовых сценариев, когда мы выводим на страницу данные, полученные от пользователя.

ляет количество пикселей в единице измерения **em**. Включение этих функций в одно место гарантирует, что они реализованы одинаково, и минимизирует общий объем написанного нами кода. Запустите редактор и скопируйте в него код, приведенный в листинге 5.7. Имена методов выделены **полужирным** шрифтом.

Листинг 5.7 ❖ Модуль браузерных утилит – `spa/js/spa.util_b.js`

```
/**
 * spa.util_b.js
 * Браузерные утилиты JavaScript
 *
 * Собрал Michael S. Mikowski
 * Эти функции я создавал и модифицировал, начиная с 1998 года,
 * черпая вдохновение из веб.
 * Лицензия МТИ
 */

/*jslint      browser : true,   continue : true,
   devel  : true,   indent : 2,     maxerr : 50,
   newcap : true,   nomen  : true,   plusplus : true,
   regexp : true,   sloppy  : true,   vars    : false,
   white  : true
 */
/*global $, spa, getComputedStyle */

spa.util_b = (function () {
    'use strict'; //----- Используем прагму strict (будет рассмотрена ниже).
    //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
    var
        configMap = { //----- Сохраняем конфигурационные параметры модуля в configMap.
            regex_encode_html : /[&'"><]/g,
            regex_encode_noamp : /["'><]/g,
            html_encode_map : {
                '&' : '&#38;',
                "'" : '&#34;',
                '"' : '&#39;',
                '>' : '&#62;',
                '<' : '&#60;'
            }
        },

        decodeHtml, encodeHtml, getEmSize; //----- Создаем модифицированную копию конфигурации, кото-
        //----- рая используется для кодирования HTML-компонентов...

    configMap.encode_noamp_map = $.extend(
        {}, configMap.html_encode_map
    );
    delete configMap.encode_noamp_map['&']; //----- ... но удаляем амперсанд.
    //----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

    //----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
```



```

// Начало decodeHtml ←
// Декодирует HTML-компоненты в среде браузера
// См. http://stackoverflow.com/questions/1912501/\unescape-html-entities-in-javascript
//
decodeHtml = function ( str ) {
    return $('<div/>').html(str || '').text();
};
// Конец decodeHtml

// Начало encodeHtml ←
// Однопроходный кодировщик HTML-компонентов, способный
// обработать произвольное число символов.
//
encodeHtml = function ( input_arg_str, exclude_amp ) {
    var
        input_str = String( input_arg_str ),
        regex, lookup_map
        ;

    if ( exclude_amp ) {
        lookup_map = configMap.encode_noamp_map;
        regex = configMap.regex_encode_noamp;
    }
    else {
        lookup_map = configMap.html_encode_map;
        regex = configMap.regex_encode_html;
    }
    return input_str.replace(regex,
        function ( match, name ) {
            return lookup_map[ match ] || '';
        }
    );
};
// Конец encodeHtml

// Начало getEmSize ←
// Возвращает размер em в пикселях
//
getEmSize = function ( elem ) {
    return Number(
        getComputedStyle( elem, '' ).fontSize.match(/\d*\.?\d*/)[0]
    );
};
// Конец getEmSize

// Экспортируем методы
return { ← Экспортируем все открытые методы.
    decodeHtml : decodeHtml,
    encodeHtml : encodeHtml,
    getEmSize : getEmSize
};
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();

```

Метод decodeHtml для преобразования HTML-компонентов, например &;, в отображаемый символ, например &.

Метод encodeHtml для преобразования специальных символов, например &, в представление, пригодное для использования в HTML, например &.

Метод getEmSize для вычисления количества пикселей в единице em.

Осталось рассмотреть только головной HTML-файл. Мы включим в него новые файлы CSS и JavaScript, как показано в листинге 5.8. Отличия от файла в главе 4 показаны **полужирным** шрифтом.

Листинг 5.8 ❖ Обновленный HTML-файл – spa/spa.html

```

<!doctype html>
<!--
    spa.html
    HTML-документ SPA
-->

<html>
<head>
    <!-- поддержка последних стандартов в части отрисовки в ie9+ -->
    <meta http-equiv="Content-Type" content="text/html;
        charset=ISO-8859-1">
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <title>SPA Chapters 5-6</title>
    <!-- сторонние таблицы стилей -->

    <!-- наши таблицы стилей -->
    <link rel="stylesheet" href="css/spa.css" type="text/css"/>
    <link rel="stylesheet" href="css/spa.shell.css" type="text/css"/>
    <link rel="stylesheet" href="css/spa.chat.css" type="text/css"/>
    <link rel="stylesheet" href="css/spa.avtr.css" type="text/css"/>

    <!-- сторонний javascript -->
    <script src="js/jq/taffydb-2.6.2.js" ></script>
    <script src="js/jq/jquery-1.9.1.js" ></script>
    <script src="js/jq/jquery.uriAnchor-1.1.3.js" ></script>
    <script src="js/jq/jquery.event.gevent-0.1.9.js"></script>
    <script src="js/jq/jquery.event.ue-0.3.2.js" ></script>

    <!-- наш javascript -->
    <script src="js/spa.js" ></script>
    <script src="js/spa.util.js" ></script>
    <script src="js/spa.data.js" ></script>
    <script src="js/spa.fake.js" ></script>
    <script src="js/spa.model.js" ></script>
    <script src="js/spa.util_b.js"></script>
    <script src="js/spa.shell.js" ></script>
    <script src="js/spa.chat.js" ></script>
    <script src="js/spa.avtr.js" ></script>
    <script>
        $(function () { spa.initModule( $('#spa') ); });
    </script>

</head>
<body>
    <div id="spa"></div>
</body>
</html>

```

Изменяем название страницы. Извини, Тототшка, мы уже не в Канзасе.

Включаем таблицу стилей для Avatar.

Включаем библиотеку базы данных на стороне клиента.

Включаем библиотеку событий gevent, необходимую для поддержки глобальных пользовательских событий.

Включаем подключаемый модуль для поддержки унифицированного ввода.

Включаем наш модуль Data.

Включаем наш модуль Fake.

Включаем наши браузерные утилиты.

Включаем наш функциональный модуль Avatar.

Итак, все на месте и можно поговорить о добавлении в наше SPA сенсорных элементов управления.

5.2.3. Использование унифицированной библиотеки ввода

В настоящее время смартфонов и планшетов продается во всем мире больше, чем традиционных ноутбуков и настольных ПК. Мы ожидаем, что продажи мобильных устройств будут и дальше расти, как и процентная доля активных устройств, поддерживающих SPA. Вскоре большинство потенциальных пользователей вашего сайта будут пользоваться сенсорным устройством.

Осознавая эту тенденцию, мы включили в эту главу рассмотрение библиотеки унифицированного ввода с помощью мыши и сенсорного экрана — `jquery.event.ue-0.3.2.js`. Эта библиотека, пусть и не идеальная, немало способствует конвергенции обоих интерфейсов ввода в одном приложении; она поддерживает мультисенсорный ввод, масштабирование сведением и разведением пальцев, перетаскивание и долгие нажатия — наряду с более привычными событиями. Мы подробно опишем работу с ней по ходу обновления пользовательского интерфейса в этой и последующих главах.

Мы уже подготовили файлы к внесению изменений. Загрузив в браузер головной HTML-документ (`spr/spr.html`), мы увидим ту же страницу, на которой остановились в главе 4, безо всяких ошибок. Теперь приступим к построению модели.

5.3. Проектирование объекта `people`

В этой главе мы построим одну из частей модели — объект `people` (рис. 5.4), а вся модель будет состоять из двух объектов: `chat` и `people`. Ниже воспроизведен черновик спецификации из главы 4.

```
...
// * chat_model - объект модели чата, который предоставляет методы для
// взаимодействия с нашей системой мгновенного обмена сообщениями.
// * people_model - объект модели пользователя, который предоставляет
// методы для управления списком пользователей, хранящимся в модели.
...
```

Описание объекта `people` — «объект, который предоставляет методы для управления списком пользователей, хранящимся в моде-

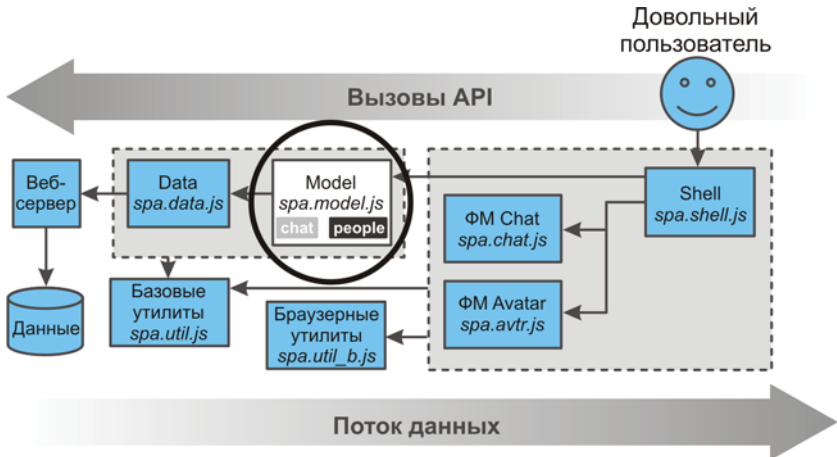


Рис. 5.4 ❖ В этом разделе мы приступаем к проектированию модели и начинаем с объекта `people`

ли» – неплохая отправная точка, но для реализации оно недостаточно детально. Начнем проектирование объекта `people` с объектов, представляющих одного человека в списке.

5.3.1. Проектирование объекта `person`

Мы решили, что объект `people` должен управлять списком людей. Опыт подсказывает, что каждого человека также удобно представлять объектом. Поэтому объект `people` будет управлять множеством объектов `person`. Ниже приведен минимальный, по нашему мнению, набор свойств объекта `person`:

- `id` – серверный идентификатор. Он будет определен для всех объектов, полученных от сервера;
- `cid` – клиентский идентификатор. Он также должен быть определен всегда и обычно совпадает с серверным идентификатором. Но если мы только что создали объект `person` на стороне клиента и на сервер он еще не попал, то серверный идентификатор не определен;
- `name` – имя человека;
- `css_map` – хэш отображаемых свойств. Он потребуется нам для поддержки аватаров.

В табл. 5.1 приведена UML-диаграмма классов для объекта `person`:

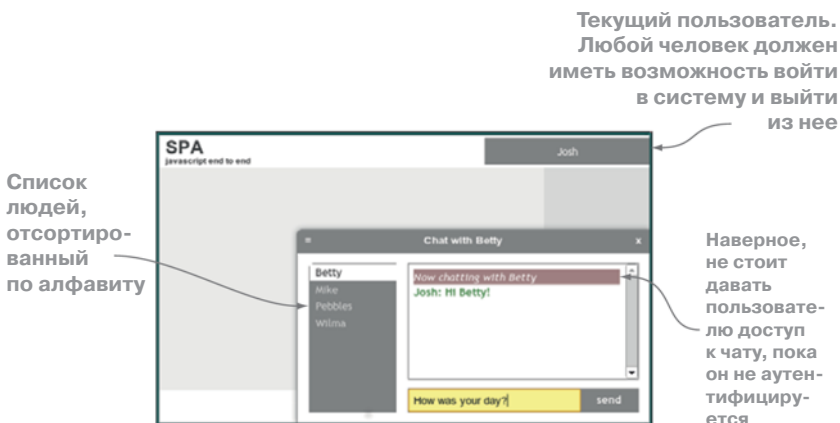
Таблица 5.1. UML-диаграмма классов для объекта *person*

person	
Имя атрибута	Тип атрибута
id	string
cid	string
name	string
css_map	map
Имя метода	Тип возвращаемого значения
get_is_user()	boolean
get_is_anon()	boolean

Можно обойтись и без клиентского идентификатора

Сейчас мы редко заводим отдельное свойство для клиентского идентификатора. Вместо этого используется единое свойство ID, а к идентификаторам, сгенерированным на стороне клиента, добавляется уникальный префикс. Например, клиентский идентификатор может быть равен x23, а серверный иметь вид 50a04142c692d1fd18000003 (особенно если мы работаем с MongoDB). Поскольку сгенерированный сервером идентификатор не может начинаться с x, легко понять, откуда взялся любой ID. Логика программы по большей части не зависит от происхождения ID. Это становится важно только в момент синхронизации с сервером.

Прежде чем думать о том, какие методы должны быть у объекта *person*, решим, какие бывают типы людей, управляемых объектом *people*. На рис. 5.5 показан эскиз экрана, который видят наши пользователи, с примечаниями касательно людей.

**Рис. 5.5 ❖ Эскиз экрана SPA с примечаниями касательно людей**

Похоже, что объект `people` должен различать четыре типа людей.

1. Текущий пользователь.
2. Анонимный пользователь.
3. Человек, с которым беседует текущий пользователь.
4. Прочие люди в онлайн.

В настоящий момент нас интересуют только текущий и анонимный пользователи – о людях в онлайн речь пойдет в следующей главе. Мы хотели бы иметь методы, позволяющие узнать, к какому типу принадлежит человек:

- `get_is_user()` – возвращает `true`, если объект `person` является текущим пользователем;
- `get_is_anon()` – возвращает `true`, если объект `person` является анонимным пользователем.

Решив, как должны выглядеть объекты `person`, разберемся, как ими будет управлять объект `people`.

5.3.2. Проектирование API объекта `people`

API объекта `people` состоит из методов и глобальных пользовательских событий jQuery. Сначала рассмотрим методы.

Методы объекта `people`

Мы хотим, чтобы в нашей модели всегда имелся объект текущего пользователя. Если пользователь еще не аутентифицирован, то этот объект будет представлять *анонимного* пользователя `person`. Разумеется, отсюда следует, что мы должны предложить средства для аутентификации и завершения сеанса. Список людей в левой части окна чата наводит на мысль, что нам понадобится список находящихся в онлайн людей, с которыми мы могли бы поболтать, и что возвращать этот список следует в алфавитном порядке. С учетом этих требований кажется разумным такой состав методов.

- `get_user()` – возвращает объект `person`, представляющий текущего пользователя. Если пользователь еще не аутентифицирован, возвращает объект, представляющий анонимного пользователя.
- `get_db()` – возвращает коллекцию всех объектов `person`, включая текущего пользователя. Желательно, чтобы этот список был отсортирован по алфавиту.
- `get_by_cid(<client_id>)` – возвращает объект `person` с указанным клиентским идентификатором. Хотя того же результата можно достичь, получив коллекцию и найдя в ней объект `person` по его клиентскому идентификатору, мы полагаем, что эта функция

будет использоваться достаточно часто, и специальный метод поможет избежать ошибок и откроет возможность для оптимизации.

- `login(<user_name>)` – аутентифицирует пользователя с указанным именем. Мы не станем вникать в сложности реальной аутентификации, поскольку эта тема выходит за рамки данной книги, а в других местах можно найти сколь угодно примеров. После того как пользователь аутентифицирован, в объекте текущего пользователя происходят изменения, отражающие вновь полученные сведения о личности. Следует также опубликовать событие `sra-login`, в котором в качестве данных фигурирует объект текущего пользователя.
- `logout()` – возвращает объект текущего пользователя в анонимное состояние. Мы должны опубликовать событие `sra-logout`, в котором в качестве данных фигурирует объект ранее аутентифицированного пользователя.

В описаниях обоих методов `login()` и `logout()` говорится, что они публикуют события. В следующем разделе мы обсудим, что это за события и для чего мы их используем.

События объекта people

События используются для асинхронной публикации данных. Например, если список людей изменился, то модель может опубликовать событие `sra-listchange`, содержащее обновленный список¹. Методы функциональных модулей или модуля Shell, заинтересованные в этом событии, могут зарегистрироваться в модели для его получения – часто это называют *подпиской на событие*. Когда происходит событие `sra-listchange`, подписчики уведомляются о нем и получают данные, опубликованные моделью. Например, в модуле Avatar может быть метод, добавляющий новый графический аватар, а в модуле Chat – метод добавления в список людей, показанных в чате. На рис. 5.6 показано, как события доставляются всем подписавшимся функциональным модулям и модулю Shell.

Мы хотели бы, чтобы в состав API объекта `people` входили по крайней мере два события²:

¹ У механизма событий есть и другие названия: push-уведомления и публикация-подписка.

² Мы используем префикс пространства имен (`sra-`) во всех именах событий. Это помогает избежать потенциальных конфликтов со сторонними модулями и библиотеками.

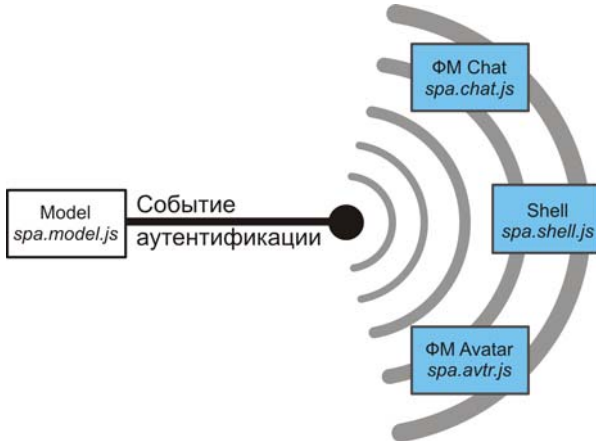


Рис. 5.6 ❖ События распространяются от нашей модели во все стороны и могут быть получены подписавшимися методами функциональных модулей или модуля Shell

- `spa-login` должно публиковаться после завершения аутентификации. Это происходит не мгновенно, потому что для аутентификации обычно нужно обратиться к серверу. Вместе с событием следует передать объект текущего пользователя;
- `spa-logout` должно публиковаться по завершении сеанса. Вместе с событием следует передать объект ранее аутентифицированного пользователя.

События зачастую являются предпочтительным способом асинхронного распространения данных. В классических реализациях на JavaScript применяются обратные вызовы, что приводит к мешанине кода, который трудно отлаживать и сохранять модульным. События позволяют модулям использовать одни и те же данные, оставаясь при этом независимыми. Поэтому всегда, когда нужно асинхронно раздать данные из модели, мы стараемся использовать события.

Поскольку мы уже используем jQuery, будет разумно в качестве механизма публикации взять глобальные пользовательские события jQuery. Для этой цели мы написали подключаемый модуль¹. Этот механизм демонстрирует неплохую производительность, а его ин-

¹ До версии 1.9.0 этот механизм поддерживался самой jQuery. Но, разумеется, они убрали его как раз перед запуском книги в печать – наверное, чтобы нам было не так скучно.

терфейс такой же, как у других событий jQuery. Можно подписаться на глобальное пользовательское событие в любой коллекции jQuery; когда оно произойдет, будет вызвана указанная функция. Вместе с событием часто передаются данные. Например, вместе с событием `spa-login` можно передать только что обновленный объект пользователя. Если из документа удаляется какой-то элемент, то все подписки на него также автоматически удаляются. Эти идеи продемонстрированы в листинге 5.9. Мы можем открыть файл `spa/spa.html` в браузере, перейти на консоль JavaScript и посмотреть, что происходит.

Листинг 5.9 ❖ Использование глобальных пользовательских событий jQuery

```
$( 'body' ).append( '<div id="spa-chat-list-box"/>' );
var $listbox = $( '#spa-chat-list-box' );
$listbox.css({
    position: 'absolute', 'z-index' : 3,
    top : 50, left : 50, width : 50, height : 50,
    border : '2px solid black', background : '#fff'
});
var onListChange = function ( event, update_map ) {
    $( this ).html( update_map.list_text );
    alert( 'onListChange ran' );
};

$.gevent.subscribe(
    $listbox,
    'spa-listchange',
    onListChange
);

$.gevent.publish(
    'spa-listchange',
    [ { list_text : 'the list is here' } ]
);

$listbox.remove();
$.gevent.publish( 'spa-listchange', [ {} ] );
```

Добавляем <div> на страницу в конец body.

Создаем коллекцию jQuery \$listbox. Стилизуем ее, чтобы было лучше видно.

Определяем функцию, которую планируем использовать для обработки глобального пользовательского события `spa-listchange`. Функция ожидает получить в качестве аргументов объект события и хэш со сведениями о том, как был обновлен список пользователей. Включаем в обработчик `alert`, чтобы знать, когда он вызывается.

Подписываемся на событие `spa-listchange` коллекции jQuery \$listbox, указывая в качестве обработчика функцию `onListChange`. Когда это событие произойдет, функция будет вызвана, причем первым ее аргументом станет объект события, а последующими – аргументы, опубликованные в составе события. Значением `this` в функции `onListChange` будет элемент DOM, с которым ассоциирована \$listbox.

При возникновении этого события вызывается подписавшаяся функция `onListChange`. Должно появиться окно `alert`, которое мы можем закрыть.

Когда мы удаляем из DOM элементы коллекции \$listbox, подписка на событие `onListChange` перестает быть действительной и удаляется.

Функция `onListChange`, привязанная к \$listbox, не вызывается, окно `alert` не появляется.

Если вы уже знакомы с обработкой событий в jQuery, то, наверное, все изложенное для вас не новость – и это хорошо. Если нет, тоже не переживайте. Просто радуйтесь, что это поведение полностью совместимо со всеми другими событиями jQuery. К тому же оно достаточно развитое, отлично протестировано и пользуется тем же кодом, что и внутренние методы jQuery. *Зачем учить два механизма, если до-*

статочно одного? Это сильный аргумент в пользу глобальных пользовательских событий jQuery, а заодно против применения других «каркасов», которые приносят с собой избыточные и слегка отличающиеся механизмы событий.

5.3.3. Документирование API объекта people

Теперь сведем все наши рассуждения к лаконичному формату и поместим их в модель Model для справки. Первая попытка показана в листинге 5.10.

Листинг 5.10 ❖ API объекта people

```
// API объекта people
// -----
// Объект people доступен по имени spa.model.people.
// Объект people предоставляет методы и события для управления
// коллекцией объектов person. Ниже перечислены его открытые методы:
// * get_user() – возвращает объект person, представляющий текущего
//   пользователя. Если пользователь не аутентифицирован, возвращает
//   объект person, представляющий анонимного пользователя.
// * get_db() – возвращает базу данных TaffyDB, содержащую все
//   объекты person, в том числе текущего пользователя, – в
//   отсортированном виде.
// * get_by_cid( <client_id> ) – возвращает объект person,
//   представляющий пользователя с указанным уникальным идентификатором.
// * login( <user_name> ) – аутентифицируется от имени пользователя
//   с указанным именем. В объект, представляющий текущего пользователя,
//   вносятся изменения, отражающие новый статус.
// * logout() – возвращает объект текущего пользователя в анонимное
//   состояние.
//
//
// Объект публикует следующие глобальные пользовательские события
// jQuery:
// * 'spa-login' публикуется по завершении аутентификации.
//   В качестве данных передается обновленный объект пользователя.
// * 'spa-logout' публикуется по завершении процедуры выхода из системы.
//   В качестве данных передается прежний объект пользователя.
//
//
// Каждый человек представляется объектом person.
// Объекты person предоставляют следующие методы:
// * get_is_user() – возвращает true, если объект соответствует
//   текущему пользователю
// * get_is_anon() – возвращает true, если объект соответствует
//   анонимному пользователю
//
//
// Объект person имеет следующие атрибуты:
// * cid – строковый клиентский идентификатор. Он всегда определен и
//   отличается от атрибута id, только если данные на стороне клиента
//   еще не синхронизированы с сервером.
```

```
// * id - уникальный идентификатор. Может быть равен undefined, если
// объект еще не синхронизирован с сервером.
// * name - строка, содержащая имя пользователя.
// * css_map - хэш атрибутов, используемый для представления аватара.
//
```

Теперь, когда объект `people` специфицирован, давайте реализуем его и протестируем API. Затем мы подправим модуль `Shell`, включив в него аутентификацию и завершение сеанса с помощью этого API.

5.4. Реализация объекта `people`

Имея проект объекта `people`, мы можем его реализовать. Для этого воспользуемся модулем `Fake`, который будет поставлять модели подставные данные. Это даст нам возможность продолжить работу без сервера и функционального модуля. Модуль `Fake` – основа быстрой разработки, и мы будем подставлять, пока не сможем предоставить.

Еще раз взглянем на нашу архитектуру, чтобы понять, чем `Fake` может помочь при разработке. Архитектура во всей своей полноте показана на рис. 5.7.

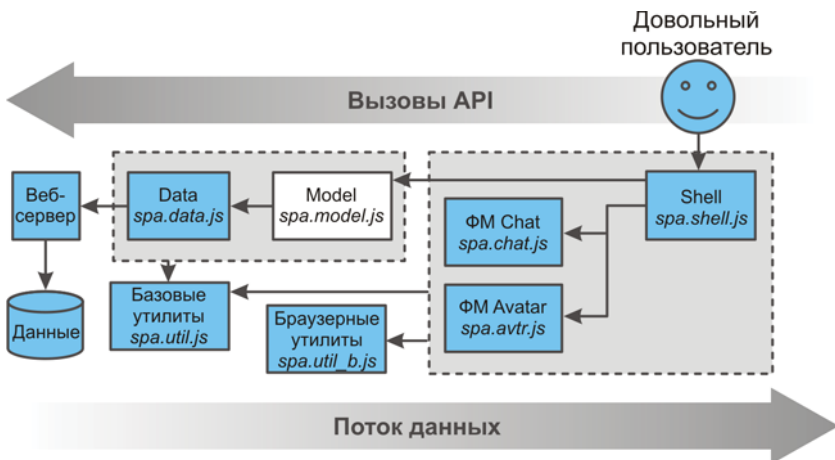


Рис. 5.7 ❖ Модель в архитектуре SPA

Все это замечательно, но построить такую архитектуру за один проход не удастся. Хорошо бы работать, не требуя ни веб-сервера, ни пользовательского интерфейса. На этом этапе мы хотим сосредоточиться на модели и не отвлекаться на другие модули. Для эмуляции

модуля Data и соединения с сервером мы можем воспользоваться модулем Fake, а для непосредственного вызова методов API – консолью JavaScript вместо окна браузера. На рис. 5.8 показано, какие модули необходимы, чтобы можно было вести разработку таким способом.

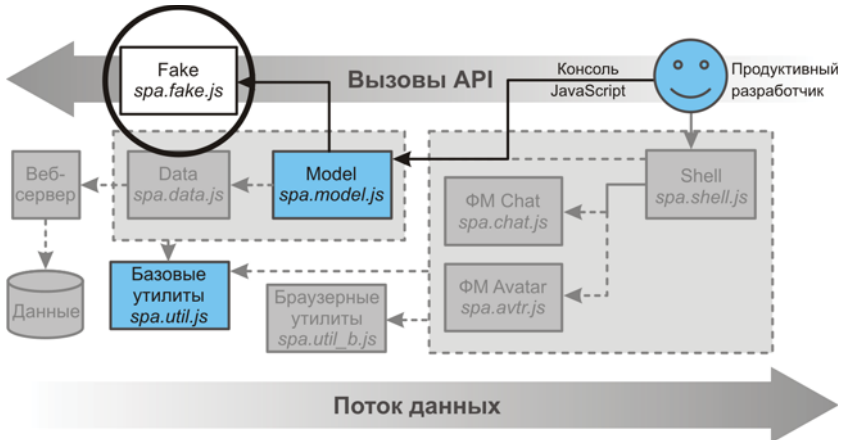


Рис. 5.8 ❖ Во время разработки мы используем подставной модуль данных, который называется Fake

Если отодвинуть в сторону весь неиспользуемый код, то останутся модули, показанные на рис. 5.9.

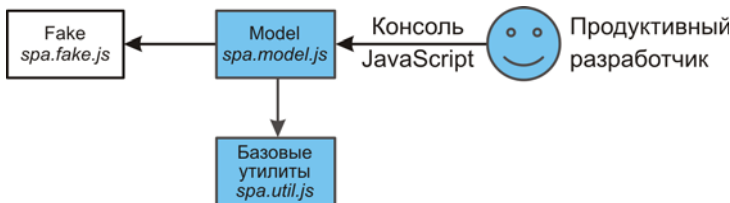


Рис. 5.9 ❖ Это все модули, используемые при разработке и тестировании модели

Применяя модуль Fake и консоль JavaScript, мы сможем сконцентрироваться исключительно на разработке и тестировании модели. Для такого важного модуля, как Model, преимущества подобного подхода особенно велики. В процессе работы мы не должны забывать,

что в этой главе серверная часть эмулируется модулем *Fake*. Определив стратегию разработки, приступим к работе над модулем *Fake*.

5.4.1. Создание подставного списка людей

То, что мы называем «настоящими» данными, обычно приходит браузеру от сервера. Но что, если мы устали, проведя весь день на работе, и сил для работы с «настоящими» данными уже не осталось? Ничего страшного – иногда допустим небольшой подлог. В этом разделе мы честно и открыто расскажем, как подделывать данные. Надеемся, что здесь вы найдете все, что хотели знать о подложных данных, но боялись спросить.

Модуль *Fake* будет предоставлять приложению подставные данные и методы. В модели предусмотрен флаг `isFakeData`, который говорит, что нужно использовать модуль *Fake*, а не «настоящие» данные от веб-сервера и методы модуля *Data*. Это дает возможность вести быструю и сосредоточенную разработку независимо от наличия сервера. Поскольку мы четко описали поведение объектов `person`, то подделать данные не составит особого труда. Сначала напишем метод `spa.fake.getPeopleList`, возвращающий список подставных людей (листинг 5.11).

Листинг 5.11 ❖ Метод модуля *Fake*, возвращающий список подставных людей

```
/*
 * spa.fake.js
 * Модуль Fake
 */
/*jslint      browser : true,   continue : true,
   devel  : true,   indent : 2,       maxerr : 50,
   newcap : true,   nomen  : true,    plusplus : true,
   regexp : true,   sloppy : true,     vars : false,
   white  : true
 */
/*global $, spa */

spa.fake = (function () {
    'use strict';
    var getPeopleList;

    getPeopleList = function () {
        return [
            { name : 'Betty', _id : 'id_01',
              css_map : { top: 20, left: 20,
                          'background-color' : 'rgb( 128, 128, 128)'}
            }
        ]
    }
}
```

```

    },
    { name : 'Mike', _id : 'id_02',
      css_map : { top: 60, left: 20,
        'background-color' : 'rgb( 128, 255, 128)'}
    }
  ],
  { name : 'Pebbles', _id : 'id_03',
    css_map : { top: 100, left: 20,
      'background-color' : 'rgb( 128, 192, 192)'}
  }
],
  { name : 'Wilma', _id : 'id_04',
    css_map : { top: 140, left: 20,
      'background-color' : 'rgb( 192, 128, 128)'}
  }
];
};

return { getPeopleList : getPeopleList };
})();

```

Мы включили в этот модуль прагму `'use strict'`. Если вы серьезно занимаетесь крупномасштабными проектами на JavaScript – а мы знаем, что так оно и есть, – то настоятельно рекомендуем использовать эту прагму *в области видимости функции, определяющей пространство имен*. При работе в строгом режиме более вероятно, что интерпретатор JavaScript возбудит исключение, обнаружив небезопасные действия, например использование необъявленных глобальных переменных. Кроме того, в этом режиме отключаются сбивающие с толку и оказавшиеся неудачными средства языка. Подавите искушение поместить прагму `strict` в глобальную область видимости, потому что в результате может «сломаться» код тех немногих сторонних разработчиков, которые еще не достигли такой степени просветления, как вы. Ну а теперь воспользуемся списком подставных людей в нашей модели.

5.4.2. Начало реализации объекта `people`

Теперь мы приступаем к реализации объекта `people` в модуле `Model`. На этапе инициализации (в методе `spa.model.initModule()`) мы первым делом создаем анонимный объект `person`, применяя тот же конструктор `makePerson`, что и для других таких объектов. Тем самым гарантируется, что у этого объекта будут такие же методы и атрибуты, как и у прочих объектов `person`, какие бы изменения мы ни вносили в конструктор в будущем.

Затем мы используем список подставных людей, полученный от метода `spa.fake.getPeopleList()`, чтобы создать TaffyDB-коллекцию объектов `person`. TaffyDB – это написанное на JavaScript хранилище данных, предназначенное для использования в браузере. У него много возможностей, присущих базе данных, в том числе выборка массива объектов с запрошенными свойствами. Например, имея TaffyDB-коллекцию объектов `person`, названную `people_db`, мы можем отобразить массив людей с именем `Pebbles`:

```
found_list = people_db({ name : 'Pebbles' }).get();
```

Почему нам нравится TaffyDB

Мы любим TaffyDB за то, что она предоставляет развитые средства управления данными в браузере и не пытается прикинуться чем-то большим (например, принести с собой модель событий, в каких-то тонких нюансах отличающуюся от принятой в jQuery). Нам нравятся такие оптимизированные инструменты с четко очерченной функциональностью, как TaffyDB. Если по какой-то причине нам потребуются другие средства управления данными, то мы сможем подменить TaffyDB другой библиотекой (или написать свою собственную), не переделывая приложение целиком. Подробная документация по TaffyDB имеется по адресу <http://www.taffydb.com>.

Наконец, мы экспортируем объект `people`, чтобы можно было протестировать API. Пока что мы предоставляем два метода для взаимодействия с объектами `person`: `spa.model.people.get_db()`, который возвращает TaffyDB-коллекцию людей, и `spa.model.people.get_cid_map()`, который возвращает хэш с клиентскими идентификаторами в качестве ключей. Откройте текстовый редактор и введите код модели, показанный в листинге 5.12. Это только первый проход, так что не расстраивайтесь, если не все поймете.

Листинг 5.12 ❖ Начало построения модели – `spa/js/spa.model.js`

```
/*
 * spa.model.js
 * Модуль Model
 */
/*jslint      browser : true,    continue : true,
   devel    : true,    indent : 2,      maxerr : 50,
   newcap   : true,    nomen   : true,    plusplus : true,
   regexp   : true,    sloppy  : true,    vars     : false,
   white    : true
 */
/*global TAFFY, $, spa */

spa.model = (function () {
    'use strict';
```

```

var
  configMap = { anon_id : 'a0' },
  stateMap = {
    anon_user      : null,
    people_cid_map : {},
    people_db      : TAFFY()
  },
  isFakeData = true,
  personProto, makePerson, people, initModule;

  personProto = {
    get_is_user : function () {
      return this.cid === stateMap.user.cid;
    },
    get_is_anon : function () {
      return this.cid === stateMap.anon_user.cid;
    }
  };

  makePerson = function ( person_map ) {
    var person,
        cid      = person_map.cid,
        css_map  = person_map.css_map,
        id       = person_map.id,
        name     = person_map.name;

    if ( cid === undefined || ! name ) {
      throw 'client id and name required';
    }

    person      = Object.create( personProto );
    person.cid   = cid;
    person.name  = name;
    person.css_map = css_map;

    if ( id ) { person.id = id; }

    stateMap.people_cid_map[ cid ] = person;
    stateMap.people_db.insert( person );
    return person;
  };

  people = {
    get_db : function () { return stateMap.people_db; },
    get_cid_map : function () { return stateMap.people_cid_map; }
  };

  initModule = function () {

```

Резервируем для анонимного пользователя специальный идентификатор.

Заводим в хэше состояния ключ `anon_user` для хранения анонимного объекта `person`.

Заводим в хэше состояния ключ `people_cid_map`, в котором будет храниться хэш объектов `person`, индексированный по клиентскому идентификатору.

Заводим в хэше состояния ключ `people_db`, в котором будет храниться TaffyDB-коллекция объектов `person`. Первоначально коллекция пуста.

Устанавливаем флаг `isFakeData` в `true`. Этот флаг сообщает модели, что нужно использовать подставные данные, объекты и методы из модуля `Fake`, а не настоящие данные, полученные от модуля `Data`.

Создать прототип для объектов `person`. Использование прототипа обычно сокращает потребность в памяти и повышает производительность объектов.

Добавляем метод `makePerson`, который создает объект `person` и сохраняет его в TaffyDB-коллекции. Одновременно не забываем обновить индекс `people_cid_map`.

Используем метод `Object.create` (<prototype>), чтобы создать объект по прототипу, а затем добавляем уникальные для объекта свойства.

Определяем объект `people`.

Добавляем метод `get_db`, который возвращает TaffyDB-коллекцию объектов `person`.

Добавляем метод `get_cid`, который возвращает хэш объектов `person`, индексированный по клиентскому идентификатору.


```

var i, people_list, person_map;

// инициализируем анонимного пользователя
stateMap.anon_user = makePerson({
  cid : configMap.anon_id,
  id : configMap.anon_id,
  name : 'anonymous'
});
stateMap.user = stateMap.anon_user;

if ( isFakeData ) {
  people_list = spa.fake.getPeopleList();
  for ( i = 0; i < people_list.length; i++ ) {
    person_map = people_list[ i ];
    makePerson({
      cid : person_map._id,
      css_map : person_map.css_map,
      id : person_map._id,
      name : person_map.name
    });
  }
}

return {
  initModule : initModule,
  people : people
};
}();

```

Создаем в методе `initModule` анонимный объект `person`, чтобы он имел такие же методы и атрибуты, как любой другой объект `person`, несмотря ни на какие будущие изменения в конструкторе. Это пример «проектирования с учетом качества».

Получаем список людей в онлайн от модуля `Fake` и добавляем их в `TaffyDB`-коллекцию `people_db`.

Разумеется, метод `spa.model.initModule()` пока ниоткуда не вызывается. Исправим это, модифицировав корневой модуль `spa/js/spa.js`, как показано в листинге 5.13.

Листинг 5.13 ❖ Добавляем инициализацию модели в корневой модуль – `spa/js/spa.js`

```

...
var spa = (function () {
  'use strict';
  var initModule = function ( $container ) {
    spa.model.initModule();
    spa.shell.initModule( $container );
  };

  return { initModule: initModule };
}());

```

Добавляем прагму `strict`;

Инициализируем `Model` раньше `Shell`.

Теперь загрузим файл `spa/spa.html` в браузер и убедимся, что страница работает, как и раньше, – если это не так или если на консоли

появляются сообщения об ошибках, значит, мы где-то напортачили и нужно внимательно проверить все сделанное. Хотя на поверхности ничего не изменилось, под капотом код работает по-другому. Открой-те консоль JavaScript в инструментах разработчика в Chrome, чтобы протестировать API объекта `people`. В листинге 5.14 показано, как получить коллекцию людей и исследовать некоторые достоинства TaffyDB. Вводимые нами команды выделены **полужирным** шрифтом, получаемые ответы – *курсивом*.

Листинг 5.14 ❖ Эксперименты с подставными людьми – а это весело

```
// получить коллекцию людей
var peopleDb = spa.model.people.get_db();
```

← Получаем TaffyDB-коллекцию объектов `person`.

```
// получить список всех людей
var peopleList = peopleDb().get();
```

← С помощью метода TaffyDB `get()` получаем из коллекции массив.

```
// показать список людей
peopleList;
```

← Инспектируем список пользователей. Обозначение `>Object` можно раскрыть – щелчок по знаку `>` выводит список свойств.

```
>> [ >Object, >Object, >Object, >Object, >Object ]
```

```
// показать имена всех людей в списке
peopleDb().each(function(person, idx){console.log(person.name)});
```

←

```
>> anonymous
>> Betty
>> Mike
>> Pebbles
>> Wilma
```

← Перебираем все объекты `person` и печатаем имя. Используем метод `each`, предоставляемый TaffyDB-коллекцией. Он принимает в качестве аргумента функцию, которой передается текущий объект `person` и его индекс в коллекции.

```
// получить человека с идентификатором 'id_03':
var person = peopleDb({ cid : 'id_03' }).first();
```

← Фильтруем TaffyDB-коллекцию с помощью метода `peopleDb(<хэш_условий>)`, а затем извлекаем из возвращенного массива первый объект методом `first()`.

```
// напечатать атрибут name
person.name;
```

← Проверяем, что значение атрибута `name` объекта `person` совпадает с ожидаемым.

```
>> "Pebbles"
```

```
// напечатать атрибут css_map
JSON.stringify( person.css_map );
```

← Отображаем еще одно ожидаемое свойство, `css_map`.

```
>> '{"top":100,"left":20,"background-color":"rgb( 128, 192, 192)"}'
```

```
// попробовать унаследованный метод
person.get_is_anon();
```

← Проверяем, что у нашего объекта `person` имеется метод `get_is_anon`, который дает правильный результат – Pebbles не является анонимным пользователем.

```
>> false
```

← Получаем анонимный объект по его идентификатору.

```
// у анонимного пользователя атрибут id должен быть равен 'a0'
person = peopleDb({ id : 'a0' }).first();
```

```
// используем тот же метод
person.get_is_anon();
```

← Проверяем, что у этого объекта имеется метод `get_is_anon` и что он работает, как положено.

```
>> true
```

```
person.name; ← Проверим имя анонимного объекта.  
>> "anonymous"
```

```
// проверим также person_cid_map ...  
var personCidMap = spa.model.people.get_cid_map();
```

```
personCidMap[ 'a0' ].name; ← Тестируем возможность получить объект person  
>> "anonymous" | по клиентскому идентификатору.
```

Тестирование показывает, что мы успешно реализовали часть объекта `people`. В следующем разделе мы закончим эту работу.

5.4.3. Завершение работы над объектом `people`

Мы должны изменить модули `Model` и `Fake`, так чтобы API объекта `people` отвечал написанной ранее спецификации. Сначала займемся модулем `Model`.

Изменение модели

Мы хотим, чтобы объект `people` в полной мере поддерживал концепцию пользователя. Поэтому добавим новые методы.

- `login(<user_name>)` начинает процедуру аутентификации. Мы должны создать новый объект `person` и добавить его в список людей. По завершении процедуры нужно будет сгенерировать событие `spa-login` и опубликовать в нем объект текущего пользователя.
- `logout()` начинает процедуру завершения сеанса. При этом объект пользователя удаляется из списка людей. По завершении процедуры нужно будет сгенерировать событие `spa-logout` и опубликовать в нем объект, представляющий ранее аутентифицированного пользователя, который только что вышел из системы.
- `get_user()` возвращает объект `person`, представляющий текущего пользователя. Если пользователь не аутентифицировался, возвращается анонимный объект `person`. Объект, представляющий текущего пользователя, мы будем хранить в переменной состояния модуля (`stateMap.user`).

Для поддержки этих методов нам понадобятся дополнительные средства.

- Поскольку для отправки сообщений модулю `Fake` и получения ответов мы будем использовать соединение `Socket.IO`, то в методе `login(<user_name>)` потребуется подставной объект `sio`.

- Так как в методе `login(<username>)` мы будем создавать объект `person`, то будет необходим метод `makeCid()` для создания клиентского идентификатора аутентифицировавшегося пользователя. Для хранения порядкового номера, входящего в состав таких идентификаторов, будем использовать переменную состояния модуля `stateMap.cid_serial`.
- Для удаления объекта `person` из списка людей нам понадобится соответствующий метод – `removePerson(<client_id>)`.
- Так как процедура аутентификации асинхронна (она завершается, только когда модуль `Fake` возвращает сообщение `userupdate`), то для ее окончания понадобится метод `completeLogin`.

Внесем в модуль `Model` соответствующие модификации, как показано в листинге 5.15. Изменения выделены **полужирным** шрифтом.

Листинг 5.15 ❖ Завершение работы над объектом модели `people` – `spa/js/spa.model.js`

```
/*
 * spa.model.js
 * Модуль Model
 */
/*jslint      browser : true,   continue : true,
   devel      : true,   indent   : 2,     maxerr    : 50,
   newcap     : true,   nomen    : true,    plusplus  : true,
   regexp     : true,   sloppy   : true,     vars     : false,
   white      : true
 */
/*global TAFFY, $, spa */

spa.model = (function () {
    'use strict';
    var
        configMap = { anon_id : 'a0' },
        stateMap = {
            anon_user      : null,
            cid_serial     : 0,
            people_cid_map : {},
            people_db      : TAFFY(),
            user           : null
        },

        isFakeData = true,

        personProto, makeCid, clearPeopleDb, completeLogin,
        makePerson, removePerson, people, initModule;

    // API объекта people ← Включаем ранее написанную документацию по API.
    // -----
```

```

// Объект people доступен по имени spa.model.people.
// Объект people предоставляет методы и события для управления
// коллекцией объектов person. Ниже перечислены его открытые методы:
// * get_user() – возвращает объект person, представляющий текущего
// пользователя. Если пользователь не аутентифицирован, возвращает
// объект person, представляющий анонимного пользователя.
// * get_db() – возвращает базу данных TaffyDB, содержащую все
// объекты person, в том числе текущего пользователя, –
// в отсортированном виде.
// * get_by_cid( <client_id> ) – возвращает объект person,
// представляющий пользователя с указанным уникальным идентификатором.
// * login( <user_name> ) – аутентифицируется от имени пользователя
// с указанным именем. В объект, представляющий текущего пользователя,
// вносятся изменения, отражающие новый статус.
// * logout() – возвращает объект текущего пользователя в анонимное
// состояние.
//
// Объект публикует следующие глобальные пользовательские события
// jQuery:
// * 'spa-login' публикуется по завершении аутентификации.
// В качестве данных передается обновленный объект пользователя.
// * 'spa-logout' публикуется по завершении процедуры выхода из системы.
// В качестве данных передается прежний объект пользователя.
//
// Каждый человек представляется объектом person.
// Объект person предоставляет следующие методы:
// * get_is_user() – возвращает true, если объект соответствует
// текущему пользователю;
// * get_is_anon() – возвращает true, если объект соответствует
// анонимному пользователю.
//
// Объект person имеет следующие атрибуты:
// * cid – строковый клиентский идентификатор. Он всегда определен и
// отличается от атрибута id, только если данные на стороне клиента
// еще не синхронизированы с сервером.
// * id – уникальный идентификатор. Может быть равен undefined, если
// объект еще не синхронизирован с сервером.
// * name – строка, содержащая имя пользователя.
// * css_map – хэш атрибутов, используемый для представления аватара.
//
personProto = {
  get_is_user : function () {
    return this.cid === stateMap.user.cid;
  },
  get_is_anon : function () {
    return this.cid === stateMap.anon_user.cid;
  }
};
makeCid = function () {

```

Добавляем генератор клиентских идентификаторов. Обычно клиентский идентификатор объект person совпадает с серверным. Но у объектов, которые созданы на стороне клиента и еще не сохранены на сервере, серверного идентификатора нет.

```

    return 'c' + String( stateMap.cid_serial++ );
};

clearPeopleDb = function ( ) {
    var user = stateMap.user;
    stateMap.people_db = TAFFY();
    stateMap.people_cid_map = {};
    if ( user ) {
        stateMap.people_db.insert( user );
        stateMap.people_cid_map[ user.cid ] = user;
    }
};

completeLogin = function ( user_list ) {
    var user_map = user_list[ 0 ];
    delete stateMap.people_cid_map[ user_map.cid ];
    stateMap.user.cid = user_map._id;
    stateMap.user.id = user_map._id;
    stateMap.user.css_map = user_map.css_map;
    stateMap.people_cid_map[ user_map._id ] = stateMap.user;

    // Когда добавится объект chat, здесь нужно будет войти в чат.
    $.gevent.publish( 'spa-login', [ stateMap.user ] );
};

makePerson = function ( person_map ) {
    var person,
        cid = person_map.cid,
        css_map = person_map.css_map,
        id = person_map.id,
        name = person_map.name;

    if ( cid === undefined || ! name ) {
        throw 'client id and name required';
    }

    person = Object.create( personProto );
    person.cid = cid;
    person.name = name;
    person.css_map = css_map;

    if ( id ) { person.id = id; }

    stateMap.people_cid_map[ cid ] = person;

    stateMap.people_db.insert( person );
    return person;
};

removePerson = function ( person ) {

```

Этот метод удаляет все объекты person, кроме анонимного и — если пользователь аутентифицирован — объекта, представляющего текущего пользователя.

Добавляем метод завершения процедуры аутентификации, вызываемый, когда от сервера получены подтверждение и данные о пользователе. Метод обновляет информацию о текущем пользователе, а затем публикует сведения об успешной аутентификации с помощью события spa-login.

Добавляем метод удаления объекта person из списка людей. Включаем несколько проверок во избежание логических противоречий — например, анонимный и текущий пользователи не удаляются.

```

if ( ! person ) { return false; }
// анонимного пользователя удалять нельзя
if ( person.id === configMap.anon_id ) {
    return false;
}

stateMap.people_db({ cid : person.cid }).remove();
if ( person.cid ) {
    delete stateMap.people_cid_map[ person.cid ];
}
return true;
};

people = (function () {
    var get_by_cid, get_db, get_user, login, logout;

    get_by_cid = function ( cid ) {
        return stateMap.people_cid_map[ cid ];
    };

    get_db = function () { return stateMap.people_db; };
    get_user = function () { return stateMap.user; };

    login = function ( name ) {
        var sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();

        stateMap.user = makePerson({
            cid : makeCid(),
            css_map : {top : 25, left : 25, 'background-color': '#8f8'},
            name : name
        });

        sio.on( 'userupdate', completeLogin );

        sio.emit( 'adduser', {
            cid : stateMap.user.cid,
            css_map : stateMap.user.css_map,
            name : stateMap.user.name
        });

        logout = function () {
            var is_removed, user = stateMap.user;
            // Когда добавится объект chat, здесь нужно будет выйти из чата

            is_removed = removePerson( user );
            stateMap.user = stateMap.anon_user;
        };
    };
}());

```

Определяем замыкание people. Оно позволяет раскрывать только те методы, которые мы пожелаем.

Определяем метод get_by_cid в замыкании people. Это вспомогательный – и очень простой – метод.

Определяем метод get_db в замыкании people. Он возвращает TaffyDB-коллекцию объектов person.

Определяем метод get_user в замыкании people. Он возвращает объект person, представляющий текущего пользователя.

Определяем метод login в замыкании people. Ни на какую настоящую аутентификацию мы не претендуем.

Регистрируем обратный вызов, который завершает аутентификацию, когда сервер опубликует сообщение userupdate.

Посылаем серверу сообщение adduser вместе со всеми данными о пользователе. В этом контексте добавление пользователя и аутентификация – одно и то же.

Определяем метод logout в замыкании people. Он публикует событие spa-logout.

```

$.gevent.publish( 'spa-logout', [ user ] );
return is_removed;
};

return { ←———— Экспортируем открытые методы объекта people.
  get_by_cid : get_by_cid,
  get_db     : get_db,
  get_user   : get_user,
  login      : login,
  logout     : logout
};
})();

initModule = function () {
  var i, people_list, person_map;

  // инициализируем анонимного пользователя
  stateMap.anon_user = makePerson({
    cid : configMap.anon_id,
    id  : configMap.anon_id,
    name : 'anonymous'
  });
  stateMap.user = stateMap.anon_user;

  if ( isFakeData ) {
    people_list = spa.fake.getPeopleList();
    for ( i = 0; i < people_list.length; i++ ) {
      person_map = people_list[ i ];
      makePerson({
        cid      : person_map._id,
        css_map  : person_map.css_map,
        id       : person_map._id,
        name     : person_map.name
      });
    }
  }
};

return {
  initModule : initModule,
  people     : people
};
})();

```

Обновив модель, можем перейти к модификации модуля Fake.

Модификация модуля Fake

В модуль Fake необходимо включить подставной объект соединения Socket.IO – `sio`. Для эмуляции тех средств, которыми мы пользу-

емя при аутентификации и завершении сеанса, нам нужны от него следующие возможности.

- Подставной объект `sio` должен уметь регистрировать обратные вызовы для обработки сообщений. Для тестирования аутентификации и завершения сеанса нам нужно поддерживать только обработку сообщения `userupdate`. Для этого мы регистрируем в модели метод `completeLogin`.
- Когда пользователь аутентифицируется, объект `sio` получает сообщение `adduser` от модели вместе с хэшем данных о пользователе. Для имитации ответа сервера мы ждем три секунды, а затем выполняем обратный вызов `userupdate`. Задержка введена сознательно, чтобы обнаружить потенциальные состояния гонки в процессе аутентификации.
- Думать о завершении сеанса при разработке подставного объекта `sio` пока не нужно, потому что в текущей версии этим занимается сама модель.

Изменения в модуле `Fake` выделены **полужирным** шрифтом в листинге 5.16.

Листинг 5.16 ❖ Добавление подставного объекта сокета с задержкой в модуль `Fake` – `spa/js/spa.fake.js`

```
...
spa.fake = (function () {
    'use strict';
    var getPeopleList, fakeIdSerial, makeFakeId, mockSio;

    fakeIdSerial = 5;

    makeFakeId = function () {
        return 'id_' + String( fakeIdSerial++ );
    };

    getPeopleList = function () {
        return [
            { name : 'Betty', _id : 'id_01',
              css_map : { top: 20, left: 20,
                'background-color' : 'rgb( 128, 128, 128)'
              }
            },
            { name : 'Mike', _id : 'id_02',
              css_map : { top: 60, left: 20,
                'background-color' : 'rgb( 128, 255, 128)'
              }
            },
            { name : 'Pebbles', _id : 'id_03',
              css_map : { top: 100, left: 20,
```

Добавляем новые переменные в область
видимости модуля.

Добавляем порядковый номер, входящий
в состав серверного идентификатора.

Добавляем метод для формирования подставного
серверного идентификатора.

```

        'background-color' : 'rgb( 128, 192, 192)';
    },
    { name : 'Wilma', _id : 'id_04',
      css_map : { top: 140, left: 20,
        'background-color' : 'rgb( 192, 128, 128)';
      }
    }
  ];
};

mockSio = (function () {
    var on_sio, emit_sio, callback_map = {};

    on_sio = function ( msg_type, callback ) {
        callback_map[ msg_type ] = callback;
    };

    emit_sio = function ( msg_type, data ) {
        // отвечаем на событие 'adduser' вызовом
        // 'userupdate' через 3 секунды
        //
        if ( msg_type === 'adduser' && callback_map.userupdate ) {
            setTimeout( function () {
                callback_map.userupdate(
                    [{ _id : makeFakeId(),
                      name : data.name,
                      css_map : data.css_map
                    } ]
                );
            }, 3000 );
        }
    };

    return { emit : emit_sio, on : on_sio };
})();

return {
    getPeopleList : getPeopleList,
    mockSio       : mockSio
};
})();

```

Определяем объект-замыкание mockSio. У него есть два открытых метода: on и emit.

Добавляем метод on_sio в замыкание mockSio. Этот метод регистрирует обратный вызов для обработки сообщения указанного типа. Например, предложение on_sio('updateuser', onUpdateuser); регистрирует функцию onUpdateuser в качестве обработчика сообщений типа updateuser. В аргументах этой функции передаются данные сообщения.

Добавляем метод emit_sio в замыкание mockSio. Этот метод имитирует отправку сообщения серверу. На первом проходе мы посылаем только сообщения типа adduser. После получения ждем 3 секунды, чтобы имитировать сетевую задержку, а затем вызываем функцию, зарегистрированную как обработчик события 'updateuser'.

Экспортируем открытые методы подставного объекта mockSio. Изменив при экспорте имена методов: on_sio — на on, а emit_sio — на emit — мы имитируем настоящий объект SocketIO.

Добавляем объект mockSio в открытый интерфейс модуля Fake.

Итак, модули Model и Fake модифицированы, и мы можем протестировать аутентификацию и завершение сеанса.

5.4.4. Тестирование API объекта people

Как и планировалось, изоляция модуля Model позволяет протестировать процедуры аутентификации и завершения сеанса, не тратя вре-

мени и ресурсов на настройку сервера и подготовку пользовательского интерфейса. Более того, мы таким образом повышаем качество продукта, потому что результаты тестирования не искажаются ошибками интерфейса и данных и тесты проводятся на известном наборе данных. Эта методика также дает возможность продолжать работу, не дожидаясь, пока другие группы закончат разработку своих компонентов.

Загрузим файл `spa/spa.html` в браузер и убедимся, что приложение работает, как и раньше. Затем откроем консоль JavaScript и протестируем методы `login`, `logout` и прочие, как показано в листинге 5.17. Вводимые нами команды выделены **полужирным** шрифтом, получаемые ответы – *курсивом*.

Листинг 5.17 ❖ Тестирование аутентификации и завершения сеанса на консоли JavaScript

```
// создать коллекцию jQuery
$t = $('<div/>'); ← Создаем коллекцию jQuery ($t), не присоединенную к документу
                        в браузере. Ниже мы воспользуемся ей для тестирования событий.

// подписать $t на глобальные пользовательские
// события, указав тестовые функции
$.gevent.subscribe( $t, 'spa-login', function () { ← Подписываем коллекцию $t на со-
    console.log( 'Hello!', arguments ); });          бытие spa-login, указав функцию,
                                                    которая печатает на консоли строку
                                                    «Hello!» и список аргументов.

$.gevent.subscribe( $t, 'spa-logout', function () { ← Подписываем коллекцию $t на собы-
    console.log('!Goodbye', arguments ); });          тие spa-logout, указав функцию,
                                                    которая печатает на консоли строку
                                                    «!Goodbye!» и список аргументов.

// получить объект текущего пользователя
var currentUser = spa.model.people.get_user();

// убедиться, что он анонимный ← Убеждаемся, что объект person, представляющий
currentUser.get_is_anon();          текущего пользователя, анонимный.
>> true

// получить коллекцию людей
var peopleDb = spa.model.people.get_db();

// показать имена всех людей ← Проверяем, что список пользователей совпадает
peopleDb().each(function(person, idx){console.log(person.name);}); с ожидаемым.
>> anonymous
>> Betty
>> Mike
>> Pebbles
>> Wilma

// аутентифицировать как 'Alfred'; через 3 секунды получить текущего пользователя
spa.model.people.login( 'Alfred' ); ← Аутентифицируемся как Alfred.
currentUser = spa.model.people.get_user();
// убедиться, что текущий пользователь уже не анонимный
```

```

currentUser.get_is_anon(); ←
>> false

// проинспектировать id и cid текущего пользователя
currentUser.id; ←
>> undefined
currentUser.cid;
>> "c0"

// ждать 3 секунды...
>> Hello! > [jQuery.Event, Object] ←

// снова распечатать коллекцию людей
peopleDb().each(function(person, idx){console.log(person.name);}); ←
>> anonymous
>> Betty
>> Mike
>> Pebbles
>> Wilma
>> Alfred

// завершить сеанс и наблюдать событие
spa.model.people.logout(); ←
>> !Goodbye [jQuery.Event, Object]

// распечатать коллекцию людей и текущего пользователя
peopleDb().each(function(person, idx){console.log(person.name);}); ←
>> anonymous
>> Betty
>> Mike
>> Pebbles
>> Wilma

currentUser = spa.model.people.get_user();
currentUser.get_is_anon(); ←
>> true

```

Проверяем, что объект `person`, представляющий текущего пользователя, больше не анонимный. Хотя сервер еще не ответил, пользователь уже установлен и метод `get_is_anon()` возвращает `false`.

Инспектируем `id` объекта текущего пользователя. Как видим, `Alfred` добавлен на стороне клиента, но его серверный `id` еще не определен. Это означает, что модель еще не ответила на запрос об аутентификации.

Распечатываем все элементы коллекции людей и убеждаемся, что среди них есть «`Alfred`».

Через 3 секунды публикуется событие `spa-login`. При этом вызывает-ся функция, зарегистрированная в качестве обработчика этого события для коллекции `$t`, и мы видим сообщение «`Hello!`» и список аргументов.

Вызываем метод `logout()`. Он выполняет ряд служебных действий и почти сразу публикует событие `spa-logout`. В результате вызывается функция, зарегистрированная в качестве обработчика этого события для коллекции `$t`, и мы видим сообщение «`!Goodbye!`» и список аргументов.

Убеждаемся, что в коллекции людей больше нет пользователя «`Alfred`».

Проверяем, что объект `person` текущего пользователя анонимный.

Результаты тестирования вселяют уверенность. Мы убедились, что объект `people` отвечает поставленным целям. Мы можем аутентифицироваться и завершить сеанс, и модель ведет себя как положено. А поскольку модель не требует ни пользовательского интерфейса, ни сервера, легко написать комплект тестов, который будет проверять соответствие всех методов спецификации. Эти тесты можно прогнать без браузера, используя `jQuery` и `Node.js`. О том, как это делается, см. приложение Б.

Самое время сделать перерыв. В следующем разделе мы модифицируем интерфейс, дав возможность пользователю аутентифицироваться и завершить сеанс.

5.5. Реализация аутентификации и завершения сеанса в Shell

До сих пор разработка модели шла независимо от пользовательского интерфейса, как показано на рис. 5.10:

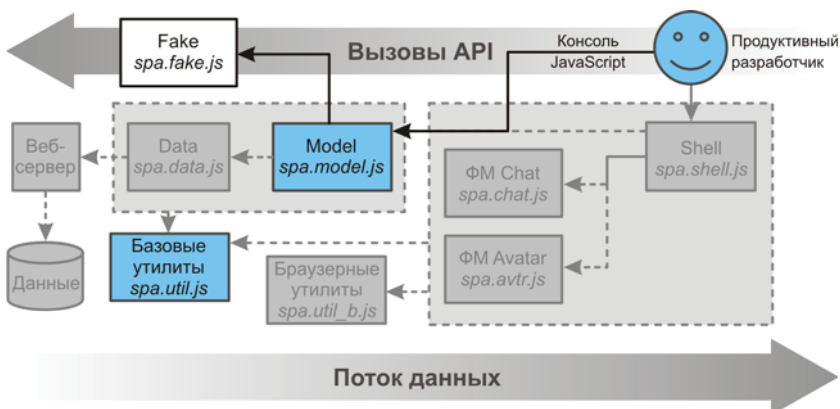


Рис. 5.10 ❖ Тестирование модели с помощью консоли JavaScript

После того как модель тщательно протестирована, мы хотим поддерживать аутентификацию и завершение сеанса в пользовательском интерфейсе, а не на консоли JavaScript. Для этого мы воспользуемся модулем Shell, как показано на рис. 5.11.

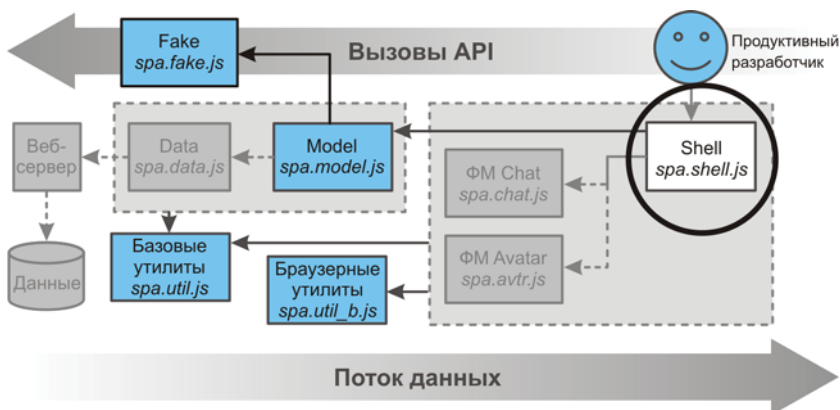


Рис. 5.11 ❖ В этом разделе мы добавим в Shell средства аутентификации с помощью графического интерфейса

Разумеется, перед тем как программировать пользовательский интерфейс, неплохо бы решить, как он должен работать.

5.5.1. Проектирование пользовательского интерфейса аутентификации

Мы хотели бы, чтобы пользовательский интерфейс был простым и знакомым. Следуя распространенному соглашению, мы сделаем так, чтобы процедура аутентификации начиналась щелчком в правой верхней части страницы. Соответствующие шаги показаны на рис. 5.12.

1. Если пользователь еще не аутентифицирован, то в правом верхнем углу («области пользователя») отображается сообщение *Please Sign-in* (Вход в систему). Когда пользователь щелкнет по нему, появится диалоговое окно аутентификации.
2. После того как пользователь заполнит форму и нажмет кнопку **ОК**, начнется процедура аутентификации.
3. Диалоговое окно убирается с экрана, и в области пользователя появляется сообщение *... processing ...* (идет обработка) – в случае нашего модуля Fake этот шаг занимает 3 секунды.
4. После того как аутентификация завершится, в области пользователя появляется имя аутентифицированного пользователя.

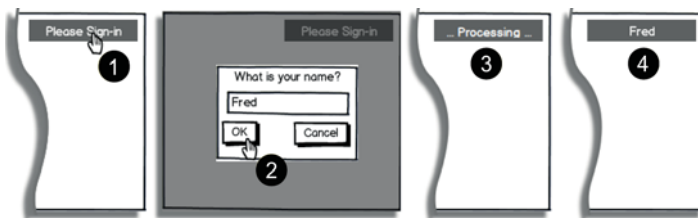


Рис. 5.12 ❖ Процедура аутентификации с точки зрения пользователя

Аутентифицированный пользователь может завершить сеанс, щелкнув мышью в области пользователя, – в результате вновь появится сообщение *Please Sign-in*.

Определившись с поведением пользовательского интерфейса, мы можем реализовать его в модуле Shell.



5.5.2. Модификация JavaScript-кода модуля Shell


Поскольку мы поместили всю бизнес-логику и обработку данных в модель, на долю Shell остаются только отображение и координация.

Ну и пока мы не закрыли капот, легко добавить поддержку сенсорных устройств (планшетов и смартфонов). Изменения, внесенные в Shell, показаны в листинге 5.18 **полужирным** шрифтом.

Листинг 5.18 ❖ Модификация Shell с целью добавления аутентификации – spa/js/spa.shell.js

```
...

spa.shell = (function () {
  'use strict';  Используем прагму strict.
  //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  var
    configMap = {
      anchor_schema_map : {
        chat : { opened : true, closed : true }
      },
      resize_interval : 200,
      main_html : String()
        + '<div class="spa-shell-head">'
          + '<div class="spa-shell-head-logo">' 
            + '<h1>SPA</h1>'
            + '<p>javascript end to end</p>'
          + '</div>'
          + '<div class="spa-shell-head-acct"></div>'
        + '</div>'
        + '<div class="spa-shell-main">'
          + '<div class="spa-shell-main-nav"></div>'
          + '<div class="spa-shell-main-content"></div>'
        + '</div>'
        + '<div class="spa-shell-foot"></div>'
        + '<div class="spa-shell-modal"></div>'
      },
    },
  copyAnchorMap, setJqueryMap, changeAnchorPart,
  onResize,      onHashchange,
  onTapAcct,    onLogin,      onLogout, 
  setChatAnchor, initModule;
  ...
  // Начало метода DOM /setJqueryMap/
  setJqueryMap = function () {
    var $container = stateMap.$container;

    jqueryMap = {
      $container : $container,  Помещаем в наш кэш коллекций jQuery.
      $acct      : $container.find('.spa-shell-head-acct'), 
      $nav       : $container.find('.spa-shell-main-nav')
    };
  };
};
```

Улучшаем внешний вид верхнего колонтиту-
ла и добавляем элемент, содержащий имя
учетной записи.

Объявляем обработчики событий onTapAcct,
onLogin, onLogout.

```
// Конец метода DOM /setJqueryMap/

...
onTapAcct = function ( event ) {
    var acct_text, user_name, user = spa.model.people.get_user();
    if ( user.get_is_anon() ) {
        user_name = prompt( 'Please sign-in' );
        spa.model.people.login( user_name );
        jqueryMap.$acct.text( '... processing ...' );
    }
    else {
        spa.model.people.logout();
    }
    return false;
};

onLogin = function ( event, login_user ) {
    jqueryMap.$acct.text( login_user.name );
};

onLogout = function ( event, logout_user ) {
    jqueryMap.$acct.text( 'Please sign-in' );
};

//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

...
initModule = function ( $container ) {
    ...
    $.gevent.subscribe( $container, 'spa-login', onLogin );
    $.gevent.subscribe( $container, 'spa-logout', onLogout );

    jqueryMap.$acct
        .text( 'Please sign-in' )
        .bind( 'utap', onTapAcct );
};

// Конец открытого метода /initModule/

return { initModule : initModule };
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();
```

Добавляем метод onTapAcct. Если элемента учетной записи касается анонимный (еще не аутентифицировавшийся) пользователь, то мы предлагаем ввести имя, а затем вызываем метод spa.model.people.login(user_name). Если же пользователь уже аутентифицирован, то мы вызываем метод spa.model.people.logout().

Добавляем обработчик события onLogin. Он обновляет область пользователя (в правом верхнем углу), заменяя текст «Please Sign-in» именем пользователя. Имя мы получаем из объекта login_user, который передается в составе события spa-login.

Добавляем обработчик события onLogout. Он восстанавливает в области пользователя текст «Please sign-in».

Подписываем коллекцию jQuery \$container на события spa-login и spa-logout, указывая соответственно обработчики onLogin и onLogout.

Инициализируем текст в области пользователя. Привязываем обработчик события onTapAcct к событию щелчка мышью и касания.

Если вы разобрались в системе публикации-подписки, применяемой в глобальных пользовательских событиях jQuery, то эти изменения понять легко. Теперь внесем изменения в CSS-стили, чтобы область пользователя отображалась правильно. Они выделены в листинге 5.19 **полужирным** шрифтом.

Листинг 5.19 ❖ Добавление стилей области пользователя в таблицу стилей Shell – spa/css/spa.shell.css

```

...

.spa-shell-head-logo {
    top    : 4px;
    left   : 8px;
    height : 32px;
    width  : 128px;
}

.spa-shell-head-logo h1 {
    font : 800 22px/22px Arial, Helvetica, sans-serif;
    margin : 0;
}

.spa-shell-head-logo p {
    font : 800 10px/10px Arial, Helvetica, sans-serif;
    margin : 0;
}

.spa-shell-head-acct {
    top        : 4px;
    right       : 0;
    width       : 210px;
    height      : 32px;
    line-height : 32px;
    background  : #888;
    color       : #fff;
    text-align  : center;
    cursor      : pointer;
    overflow    : hidden;
    text-overflow : ellipsis;
}

...

.spa-shell-main-nav {
    width      : 400px;
    background : #eee;
    z-index    : 1;
}

.spa-shell-main-content {
    left       : 400px;
    right      : 0;
    background : #ddd;
}

...

```

Изменяем класс spa-shellhead-logo с целью немного отодвинуть область логотипа от края.

Создаем производный селектор .spa-shell-head-logo h1, размещая его с отступом. Он стилизует элементы h1 внутри div'a с логотипом.

Удаляем селектор .spa-shell-head-search.

Создаем производный селектор .spa-shell-head-logo p, размещая его с отступом. Он стилизует элементы p внутри div'a с логотипом.

Модифицируем селектор .spa-shell-head-acct, так чтобы сделать текст в области пользователя более заметным.

Модифицируем селектор .spa-shell-main-content, чтобы учесть увеличившуюся ширину примыкающего контейнера с классом .spa-shell-main-nav.

Модифицируем селектор .spa-shell-main-nav – увеличиваем ширину и задаем z-index, так чтобы этот элемент располагался поверх всех элементов в контейнере с классом .spa-shell-main-content.

Закончив с CSS, протестируем изменения.

5.5.4. Тестирование аутентификации и завершения сеанса в пользовательском интерфейсе

После загрузки головного HTML-файла в браузер мы должны увидеть страницу с надписью «Please sign in» в области пользователя – правом верхнем углу окна. После щелчка по ней должно появиться диалоговое окно, показанное на рис. 5.13.

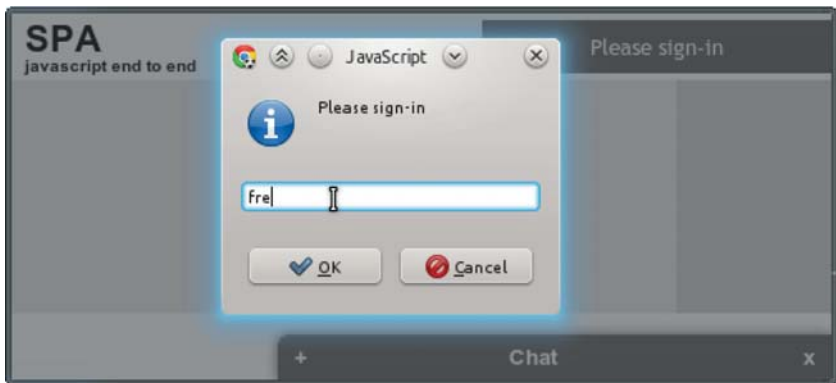


Рис. 5.13 ❖ Диалоговое окно аутентификации на экране

После того как мы введем имя пользователя и нажмем **ОК**, диалоговое окно должно закрыться, и в течение трех секунд в области пользователя должно отображаться сообщение «... processing ...»¹, после чего публикуется событие `spa-login`. Подписанный на это событие обработчик в модуле `Shell` должен вывести имя пользователя в правом верхнем углу окна, как показано на рис. 5.14.

Мы держим пользователя в курсе всего происходящего на протяжении этой процедуры. Это отличительный знак хорошего дизайна – благодаря систематическому предоставлению обратной связи даже относительно медленное приложение может показаться шустрым и отзывчивым.

¹ Перед тем как публиковать этот сайт, стоило бы заменить текст каким-нибудь симпатичным анимированным графическим индикатором хода выполнения. Качественные индикаторы бесплатно предлагаются на многих веб-сайтах.

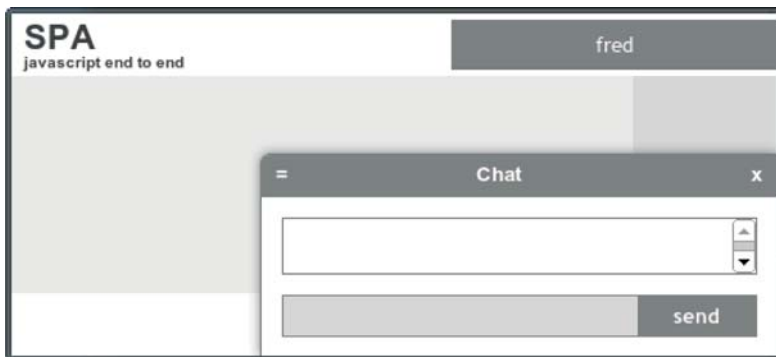


Рис. 5.14 ❖ Вид экрана после завершения аутентификации

5.6. Резюме

В этой главе мы познакомились с моделью и обсудили ее место в нашей архитектуре. Мы описали, что модель должна и чего *не* должна делать. Затем мы подготовили структуру файлов и каталогов для реализации и тестирования модели.

Мы спроектировали, специфицировали, разработали и протестировали одну часть модели – объект `people`. Для предоставления модели контролируемого набора данных мы воспользовались модулем `Fake`, а для тестирования API объекта `people` – консолью JavaScript. Подобная изоляция модели позволила вести разработку быстрее, а тестирование производить в контролируемом окружении. Мы также модифицировали наше SPA, добавив подключаемый модуль, который унифицирует ввод с помощью мыши и сенсорного экрана. В результате с нашим приложением смогут работать и мобильные пользователи.

В последнем разделе главы мы изменили модуль `Shell`, включив средства аутентификации и завершения сеанса. Для этого мы воспользовались API, предоставляемым объектом `people`. Мы также сделали интерфейс удобным для пользователей, обеспечив быструю реакцию SPA на их действия.

В следующей главе мы добавим в модель объект `chat`. В результате мы сможем закончить работу над функциональным модулем `Chat` и построить функциональный модуль `Avatar`. Затем мы займемся подготовкой клиента к работе с настоящим веб-сервером.

Глава 6

Завершение модулей Model и Data

В этой главе:

- ✧ Проектирование второй части модели – объекта `chat`.
- ✧ Реализация объекта `chat` и тестирование его API.
- ✧ Завершение функционального модуля `Chat`.
- ✧ Создание нового функционального модуля `Avatar`.
- ✧ Использование jQuery для привязки к данным.
- ✧ Взаимодействие с сервером с помощью модуля `Data`.

В этой главе мы завершим работу над моделью и функциональными модулями, начатую в главе 5. Перед тем как приступить к ее чтению, вы должны иметь файлы из проекта, созданного в главе 5, потому что мы будем дополнять их. Мы рекомендуем целиком скопировать созданное в главе 5 дерево каталогов со всеми файлами в новый каталог «chapter_6» и там уже вносить изменения.

В этой главе мы спроектируем и реализуем часть модели – объект `chat`. Затем мы закончим пользовательский интерфейс всплывающего чата, включив в него использование API объекта `chat`. Мы также добавим функциональный модуль `Avatar`, который с помощью API объекта `chat` будет отображать список представлений людей, находящихся в онлайн. Мы обсудим, как средствами jQuery реализовать привязку к данным. Наконец, мы довершим клиентскую часть SPA, добавив в нее модуль `Data`.

Начнем с проектирования объекта `chat`.

6.1. Проектирование объекта `chat`

В этой главе мы займемся созданием объекта модели `chat`, показанного на рис. 6.1.

В предыдущей главе мы спроектировали, реализовали и протестировали объект модели `people`, а в этой сделаем то же самое с объектом `chat`. Вернемся к спецификации API из главы 4:

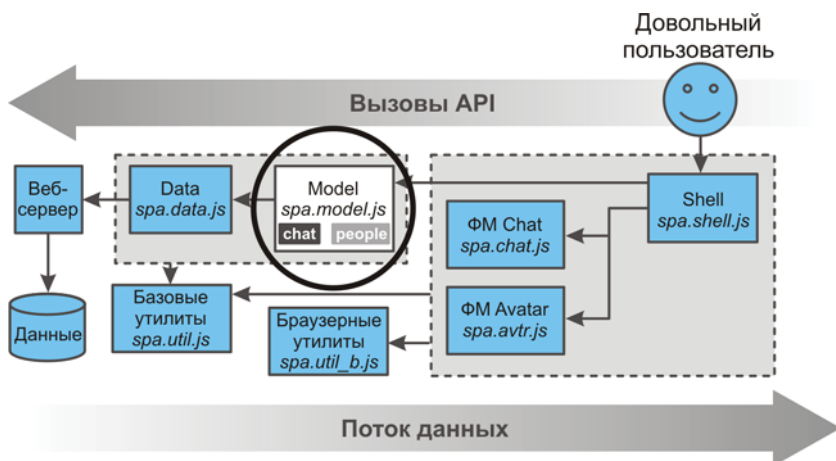


Рис. 6.1 ❖ В этой главе мы будем работать с объектом модели `chat`

```
...
// * chat_model - объект модели chat, который предоставляет методы для
// взаимодействия с нашей системой мгновенного обмена сообщениями.
// * people_model - объект модели people, который предоставляет
// методы для управления списком пользователей, хранящимся в модели.
...
```

Приведенное здесь описание объекта `chat` – «объект, который предоставляет методы для взаимодействия с нашей системой мгновенного обмена сообщениями» – неплохая отправная точка, но для реализации оно слишком общее. Прежде чем проектировать объект `chat`, проанализируем, чего мы от него хотим.

6.1.1. Проектирование методов и событий

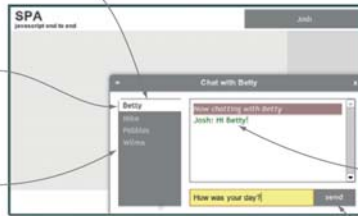
Мы знаем, что объект `chat` должен предоставлять средства для мгновенного обмена сообщениями, но нужно точно определить, что это за средства. Рассмотрим рис. 6.2, на котором показан эскиз SPA с комментариями относительно интерфейса нашего чата.

По опыту мы знаем, что, вероятно, понадобится инициализировать комнату чата. Мы также ожидаем, что у пользователя должны быть возможность изменить собеседника и возможность отправлять тому сообщения. А из обсуждения аватаров мы заключаем, что пользователь может изменить сведения о своем аватаре. Но интерфейс чата определяется не только пользователем; чат могут посещать и покидать другие люди, которые также посылают и принимают сообщения

Вероятно, понадобится
возможность войти
в чат

Сейчас собеседни-
ком является Betty.
У пользователя
должна быть возмож-
ность в любой момент
сменить собеседника

Пользователь должен
видеть актуальный
список людей в онлайн



Пользователь должен
получать уведомления
о поступающих
сообщениях

Должна быть
возможность
посылать сообщения

Рис. 6.2 ❖ Эскиз нашего SPA – акцент на чате

и изменяют сведения о своих аватарах. Проведенный анализ позволяет составить список функций, которые должны присутствовать в API объекта `chat`:

- методы для входа в чат и выхода из чата;
- метод для смены собеседника;
- метод для отправки сообщений другим людям;
- метод, который извещает сервер о том, что пользователь изменил свой аватар;
- событие, публикуемое, когда собеседник по какой-то причине изменился, например если он покинул чат или пользователь выбрал другого собеседника;
- событие, публикуемое, когда нужно по какой-то причине обновить панель сообщений, например если пользователь отправил или получил сообщение;
- событие, публикуемое, когда по какой-то причине изменился список людей в онлайн, например если пользователь вошел в чат, вышел из него или переместил свой аватар.

В API нашего объекта `chat` используются два канала коммуникации. Первый – классический механизм возврата значения из метода. Это *синхронный* канал – последовательность передачи данных заранее известна. Объект `chat` может вызывать внешние методы и получать от них информацию в виде возвращаемых значений. И наоборот, любой код может вызывать открытые методы объекта `chat` и точно так же получать от них информацию.

Второй канал коммуникации – механизм событий. Этот канал *асинхронный* – события могут происходить в любое время и не зависят от того, что в данный момент делает объект `chat`. Объект `chat` может получать события (например, сообщения от сервера) и публи-

ковать события – то и другое может приводить к обновлению пользовательского интерфейса.

Сначала рассмотрим, какие синхронные методы будет предоставлять объект `chat`.

Проектирование методов чата

Как было сказано в главе 5, метод – это раскрываемая внешнему миру функция, например `spa.model.chat.get_chatee`, которая служит для синхронного выполнения некоторого действия и возврата результата. С учетом наших требований представляется разумным такой список методов:

- `join()` – войти в чат. Если пользователь анонимный, то этот метод должен просто вернуть `false` и больше ничего не делать;
- `get_chatee()` – вернуть объект `person`, представляющий пользователя, с которым мы беседуем. Если такого нет, вернуть `null`;
- `set_chatee(<person_id>)` – установить в качестве собеседника объект `person`, представляющий пользователя с уникальным идентификатором `person_id`. Этот метод должен опубликовать событие `spa-setchatee`, сопроводив его данными о собеседнике. Если подходящего объекта `person` нет в онлайн, установить в качестве собеседника `null`. Если указанный человек уже является собеседником, вернуть `false`;
- `send_message(<msg_text>)` – отправить собеседнику сообщение. При этом необходимо опубликовать событие `spa-updatechat`, сопроводив его данными о сообщении. Если пользователь анонимный или собеседник равен `null`, то этот метод должен вернуть `false` и больше ничего не делать;
- `update_avatar(<update_avatar_map>)` – обновить сведения об аватаре для объекта `person`. Аргумент (`update_avatar_map`) должен содержать свойства `person_id` и `css_map`.

Похоже, эти методы удовлетворяют нашим требованиям. Теперь более подробно рассмотрим события, которые должен публиковать объект `chat`.

Проектирование событий чата

Как отмечалось выше, события применяются для асинхронной публикации данных. Например, получив сообщение, объект `chat` должен уведомить подписанные коллекции jQuery об изменении и предоставить данные, необходимые для обновления внешнего представления.

Мы ожидаем, что список людей в онлайн и собеседник будут изменяться часто. Не всегда эти изменения обусловлены действиями

пользователя – например, собеседник может в любой момент измениться, потому что другой пользователь послал сообщение. Ниже перечислены события, с помощью которых эти изменения доводятся до сведения функциональных модулей.

- **spa-listchange** публикуется при изменении списка людей в он-лайне. В качестве данных передается сам измененный список.
- **spa-setchatee** публикуется при смене собеседника. В качестве данных передается хэш, содержащий сведения о старом и новом собеседниках.
- **spa-updatechat** публикуется при получении или отправке сообщения. В качестве данных передается хэш, содержащий сведения о сообщении.

Как и в главе 5, в качестве механизма публикации мы возьмем глобальные события jQuery. Определившись с методами и событиями, мы можем приступить к документированию и реализации.

6.1.2. Документирование API объекта chat

Оформим наши планы в виде спецификации API, которую можно поместить в код модели для справки.

Листинг 6.1 ❖ API объекта chat – spa/js/spa.model.js

```
// API объекта chat
// -----
// Объект chat доступен по имени spa.model.chat.
// Объект chat предоставляет методы и события для управления
// сообщениями в чате. Ниже перечислены его открытые методы:
// * join() – войти в чат. Настраивает протокол
// взаимодействия чата с сервером, включая публикаторов
// глобальных пользовательских событий 'spa-listchange' и
// 'spa-updatechat'. Если текущий пользователь анонимный,
// то join() возвращает false и больше ничего не делает.
// * get_chatee() – вернуть объект person, представляющий
// собеседника пользователя. Если такого нет, возвращается
// null.
// * set_chatee( <person_id> ) – установить в качестве
// собеседника объект person с указанным идентификатором
// person_id. Если person_id не входит в список людей, то
// в качестве собеседника устанавливается null. Если
// указанный человек уже является собеседником, то метод
// возвращает false. Публикуется глобальное пользовательское
// событие 'spa-setchatee'.
// * send_msg( <msg_text> ) – отправить сообщение собеседнику.
// Публикуется глобальное пользовательское событие
// 'spa-updatechat'. Если пользователь анонимный или
// собеседник равен null, метод возвращает false и больше
// ничего не делает.
```



```
// * update_avatar( <update_avtr_map> ) - отправить серверу
// хэш update_avtr_map. В результате публикуется событие
// 'spa-listchange', сопровождаемое обновленным списком
// людей и аватаров (атрибут css_map для каждого человека).
// Хэш update_avtr_map должен иметь вид
// { person_id : person_id, css_map : css_map }.
//
// Ниже перечислены глобальные пользовательские события
// jQuery, публикуемые объектом:
// * spa-setchatee - публикуется при установке нового
// собеседника. В качестве данных передается хэш вида
// { old_chatee : <old_chatee_person_object>,
//   new_chatee : <new_chatee_person_object>
// }
// * spa-listchange - публикуется при изменении длины
// списка людей в онлайн (то есть когда кто-то входит в чат
// или выходит из чата) или его содержимого (то есть когда
// изменяется информация о каком-то аватаре).
// Получатель этого события должен запросить обновленные
// данные у метода people_db объекта модели people.
// * spa-updatechat - публикуется при получении или отправке
// сообщения. В качестве данных передается хэш вида
// { dest_id : <chatee_id>,
//   dest_name : <chatee_name>,
//   sender_id : <sender_id>,
//   msg_text : <message_content>
// }
//
```

Теперь, когда объект `chat` специфицирован, давайте реализуем его и протестируем API. Затем мы подправим модуль `Shell` и функциональные модули, предоставив в них новые возможности с помощью API объекта `chat`.

6.2. Реализация объекта `chat`

Имея проект API объекта `chat`, мы можем его реализовать. Как и в главе 5, мы будем пользоваться модулем `Fake` и консолью `JavaScript`, чтобы не зависеть от наличия веб-сервера и пользовательского интерфейса. *Мы должны все время помнить, что в этой главе «сервер» имитируется модулем `Fake`.*

6.2.1. Начинаем с метода `join`

В этом разделе мы создадим объект модели `chat`, который позволит нам:

- аутентифицироваться с помощью метода `spa.model.people.login(<username>);`

- войти в чат с помощью метода `spa.model.chat.join()`;
- зарегистрировать обработчик события `spa-listchange`, вызываемый, когда модель получает от сервера сообщение `listchange` об изменении списка людей в онлайн.

Объект `chat` будет пользоваться объектом `people` для аутентификации и отслеживания списка людей в онлайн. Он не позволит войти в чат анонимному пользователю. В листинге 6.2 **полужирным** шрифтом показаны изменения в коде объекта `chat` в модуле `Model`.

Листинг 6.2 ❖ Начало реализации объекта `chat` – `spa/js/spa.model.js`

```
spa.model = (function () {
  ...
  stateMap = {
    ...
    is_connected : false, ← Флаг stateMap.is_connected определяет,
                           находится ли пользователь в чате.
    ...
  },
  ...
  personProto, makeCid, clearPeopleDb, completeLogin,
  makePerson, removePerson, people, chat, initModule;
  ...
  // API объекта chat
  // -----
  // Объект chat доступен по имени spa.model.chat.
  // Объект chat предоставляет методы и события для управления
  // сообщениями в чате. Ниже перечислены его открытые методы:
  // * join() – войти в чат. Настраивает протокол
  // взаимодействия чата с сервером, включая публикаторов
  // глобальных пользовательских событий 'spa-listchange' и
  // 'spa-updatechat'. Если текущий пользователь анонимный,
  // то join() возвращает false и больше ничего не делает.
  // ...
  // Ниже перечислены глобальные пользовательские события
  // jQuery, публикуемые объектом:
  // ...
  // * spa-listchange – публикуется при изменении длины
  // списка людей в онлайн (то есть когда кто-то входит в чат
  // или выходит из чата) или его содержимого (то есть когда
  // изменяется информация о каком-то аватаре).
  // Получатель этого события должен запросить обновленные
  // данные у метода people_db объекта модели people.
  // ...
  //
  chat = (function () { ← Создаем пространство имен chat.
    var
      _publish_listchange,
      _update_list, _leave_chat, join_chat;

    // Начало внутренних методов
```

```

_update_list = function( arg_list ) {
  var i, person_map, make_person_map,
      people_list = arg_list[ 0 ];

  clearPeopleDb();

  PERSON:
  for ( i = 0; i < people_list.length; i++ ) {
    person_map = people_list[ i ];

    if ( ! person_map.name ) { continue PERSON; }
    // если пользователь определен, обновить
    // css_map и больше ничего не делать
    if ( stateMap.user && stateMap.user.id === person_map._id ) {
      stateMap.user.css_map = person_map.css_map;
      continue PERSON;
    }

    make_person_map = {
      cid      : person_map._id,
      css_map  : person_map.css_map,
      id       : person_map._id,
      name     : person_map.name
    };

    makePerson( make_person_map );
  }

  stateMap.people_db.sort( 'name' );
};

_publish_listchange = function ( arg_list ) {
  _update_list( arg_list );
  $.gevent.publish( 'spa-listchange', [ arg_list ] );
};
// Конец внутренних методов

_leave_chat = function () {
  var sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();

  stateMap.is_connected = false;
  if ( sio ) { sio.emit( 'leavechat' ); }
};

join_chat = function () {
  if ( stateMap.is_connected ) { return false; }

  if ( stateMap.user.get_is_anon() ) {
    console.warn( 'User must be defined before joining chat' );
  }
}

```

Метод `_update_list` обновляет объект `people` после получения нового списка людей.

Метод `_publish_listchange` публикует глобальное пользовательское событие `spa-listchange`, сопровождаемое обновленным списком людей. Мы ожидаем, что этот метод будет вызываться после получения сообщения `listchange` от сервера.

Метод `_leave_chat` отправляет серверу сообщение `leavechat` и очищает переменные состояния.

Метод `join_chat` вызывается, чтобы войти в чат. Он проверяет, находится ли уже пользователь в чате (`stateMap.is_connected`), чтобы не регистрировать обработчик события `listchange` более одного раза.

```

        return false;
    }

    sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();
    sio.on( 'listchange', _publish_listchange );
    stateMap.is_connected = true;
    return true;
};

return { ← Экспортируем все открытые методы объекта chat.
    _leave : _leave_chat,
    join : join_chat
};
}());

initModule = function () {
    // инициализировать анонимный объект person
    stateMap.anon_user = makePerson({
        cid : configMap.anon_id,
        id : configMap.anon_id,
        name : 'anonymous'
    });
    stateMap.user = stateMap.anon_user; ← Удаляем код, который вставляет подставной список
};                                     людей в объект people, поскольку теперь это де-
                                     лается при входе пользователя в чат.

return {
    initModule : initModule,
    chat       : chat, ← Добавляем chat как открытый объект.
    people     : people
};
}());

```

Это первый этап реализации объекта `chat`. Вместо того чтобы добавлять еще какие-то методы, протестируем то, что уже сделано. В следующем разделе мы модифицируем модуль `Fake`, наделив его возможностью имитировать взаимодействие с сервером, необходимое для тестирования.

6.2.2. Модификация модуля `Fake` для поддержки метода `chat.join`

Теперь нам нужно обновить модуль `Fake`, чтобы он мог имитировать ответы сервера, необходимые для тестирования метода `join`. Вот какие изменения мы хотим добавить:

- включить аутентифицированного пользователя в подставной список людей;
- имитировать получение сообщения `listchange` от сервера.

Первый шаг прост: создаем хэш с атрибутами человека и вставляем его в список людей, поддерживаемый модулем Fake. Второй сложнее, поэтому читайте внимательно: объект `chat` регистрирует обработчик сообщения `listchange` от сервера *только после того, как пользователь аутентифицировался и вошел в чат*. Поэтому мы можем добавить закрытую функцию `send_listchange`, которая посылает подставной список людей лишь в случае, когда обработчик зарегистрирован. Изменения выделены в листинге 6.3 **полужирным** шрифтом.

Листинг 6.3 ❖ Модификация модуля Fake для имитации сообщений сервера о входе в чат – `spa/js/spa.fake.js`

```
...
spa.fake = (function () {
  'use strict';
  var peopleList, fakeIdSerial, makeFakeId, mockSio;

  fakeIdSerial = 5;

  makeFakeId = function () {
    return 'id_' + String( fakeIdSerial++ );
  };

  peopleList = [ ← Создаем список хэшей peopleList, в котором хранит-
    { name : 'Betty', _id : 'id_01',
      css_map : { top: 20, left: 20,
                  'background-color' : 'rgb( 128, 128, 128)'}
    },
    { name : 'Mike', _id : 'id_02',
      css_map : { top: 60, left: 20,
                  'background-color' : 'rgb( 128, 255, 128)'}
    },
    { name : 'Pebbles', _id : 'id_03',
      css_map : { top: 100, left: 20,
                  'background-color' : 'rgb( 128, 192, 192)'}
    },
    { name : 'Wilma', _id : 'id_04',
      css_map : { top: 140, left: 20,
                  'background-color' : 'rgb( 192, 128, 128)'}
    }
  ];

  mockSio = (function () {
    var
```

```

on_sio, emit_sio,
send_listchange, listchange_idto,
callback_map = {}];

on_sio = function ( msg_type, callback ) {
    callback_map[ msg_type ] = callback;
};

emit_sio = function ( msg_type, data ) {
    var person_map;

    // отвечаем на событие 'adduser' вызовом
    // 'userupdate' через 3 секунды
    if ( msg_type === 'adduser' && callback_map.userupdate ) {
        setTimeout( function () {
            person_map = {
                _id      : makeFakeId(),
                name      : data.name,
                css_map   : data.css_map
            };
            peopleList.push( person_map );
            callback_map.userupdate([ person_map ]);
        }, 3000 );
    }
};

// Пытаемся воспользоваться обработчиком listchange один ←
// раз в секунду. Прекращаем попытки после первой удачной.
send_listchange = function () {
    listchange_idto = setTimeout( function () {
        if ( callback_map.listchange ) {
            callback_map.listchange([ peopleList ]);
            listchange_idto = undefined;
        }
        else { send_listchange(); }
    }, 1000 );
};

// Мы должны запустить процесс ... ←
send_listchange();

return { emit : emit_sio, on : on_sio };
}());

return { mockSio : mockSio }; ←
}());

```

Изменяем реакцию на сообщение adduser (возникающее после аутентификации пользователя) – помещаем описание пользователя в подставной список людей.

Добавляем функцию send_listchange, которая имитирует получение сообщения listchange от сервера. Один раз в секунду этот метод проверяет, существует ли обработчик сообщения listchange (объект chat регистрирует его только после того, как пользователь аутентифицировался и вошел в чат). Если обработчик найден, метод вызывает его, передавая подставной список peopleList в качестве аргумента, после чего проверка наличия обработчика прекращается.

В этой строке вызывается функция send_listchange.

Удаляем метод getPeopleList, потому что требуемые данные теперь представляются обработчиком сообщения listchange.

Реализовав часть объекта chat, протестируем ее – точно так же мы действовали в случае объекта people в главе 5.

6.2.3. Тестирование метода chat.join

Прежде чем продолжать работу над объектом `chat`, проверим, что уже реализованные средства работают, как и ожидалось. Загрузите файл `spa/spa.html` в браузер, откройте консоль JavaScript и убедитесь, что SPA не содержит ошибок JavaScript. Далее с помощью консоли можно протестировать методы, как показано в листинге 6.4. Вводимые нами команды выделены **полужирным** шрифтом, получаемые ответы — *курсивом*.

Листинг 6.4 ❖ Тестирование метода `spa.model.chat.join()` без сервера и пользовательского интерфейса

```
// создать коллекцию jQuery
var $t = $('<div/>');

// подписать $t на глобальные пользовательские
// события, указав тестовые функции
$.gevent.subscribe( $t, 'spa-login', function () {
    console.log( 'Hello!', arguments ); });

$.gevent.subscribe( $t, 'spa-listchange', function () {

// получить объект текущего пользователя
var currentUser = spa.model.people.get_user();

// убедиться, что он еще не аутентифицирован
currentUser.get_is_anon();
>> true

// попытаться войти в чат без аутентификации
spa.model.chat.join();
>> User must be defined before joining chat

// аутентифицироваться, ждать 3 секунды.
// Пользовательский интерфейс тоже обновится!
spa.model.people.login( 'Fred' );
>> Hello! > [jQuery.Event, Object]

// получить коллекцию людей
var peopleDb = spa.model.people.get_db();

// показать имена всех людей
peopleDb().each(function(person, idx){console.log(person.name); });
```

Создаем коллекцию jQuery (\$t), не присоединенную к документу в браузере. Ниже мы воспользуемся ей для тестирования событий.

Подписываем коллекцию \$t на событие spa-login, указав функцию, которая печатает на консоли строку «Hello!» и список аргументов.

Подписываем коллекцию \$t на событие spa-listchange, указав функцию, которая печатает на консоли строку «*Listchange» и список аргументов.

Получаем объект текущего пользователя от объекта people.

С помощью метода get_is_anon() убеждаемся, что этот пользователь еще не аутентифицирован.

Пытаемся войти в чат без аутентификации. В точном соответствии со спецификацией доступ запрещен.

Аутентифицируемся как Fred. Текст «Please sign-in» в области пользователя в правом верхнем углу окна браузера сначала изменится на «... processing...», а затем на «Fred». По завершении аутентификации публикуется событие spa-login. В ответ на него вызывается зарегистрированный для коллекции \$t обработчик, поэтому мы видим сообщение «Hello!» и список аргументов.

Получаем коллекцию TaffyDB от объекта people.

Проверяем, что в коллекции людей есть только Fred и анонимный пользователь. Это правильно, потому что мы еще не вошли в чат.

```
>> anonymous
>> Fred
```

```
// войти в чат ← Входим в чат.
spa.model.chat.join();
>> true
```

Менее чем через секунду после вызова `join()` должно быть опубликовано событие `spa-listchange`. В ответ на него вызывается зарегистрированный для коллекции `$t` обработчик, поэтому мы видим сообщение «Hello!» и список аргументов.

```
// событие spa-listchange должно произойти почти мгновенно
>> *listchange > [jQuery.Event, Array[1]]
```

Убеждаемся, что возвращен массив аргументов в стиле `Socket.IO`. Первым элементом в этом массиве является обновленный список людей.

```
// снова распечатать список пользователей. Мы видим, что список
// изменился, в нем теперь присутствуют все люди в онлайнe
var peopleDb = spa.model.people.get_db(); ← Получаем обновленный список людей.
peopleDb().each(function(person, idx){console.log(person.name);});
>> Betty
>> Fred ← Убеждаемся, что теперь в списке людей есть вся наша
>> Mike подставная тусовка, а также аутентифицированный поль-
>> Pebbles зователь Fred.
>> Wilma
```

Мы реализовали и протестировали первую часть объекта `chat`, то есть методы для аутентификации, входа в чат и просмотра списка людей в онлайнe. Теперь нужно научить `chat` отправлять и получать сообщения.

6.2.4. Добавление средств работы с сообщениями в объект `chat`

Отправка и получение сообщения – не такая простая вещь, как кажется. Как сказали бы в компании FedEx, нужно наладить *логистику* – управление приемом и передачей сообщений. Нам необходимо:

- поддерживать запись с данными о *собеседнике* – человеке, с которым беседует пользователь;
- отправлять вместе с сообщением дополнительные данные: идентификатор и имя отправителя, идентификатор получателя;
- корректно обрабатывать ситуацию, когда из-за сетевых задержек пользователь, которому мы отправили сообщение, уже успел выйти из чата;
- публиковать глобальные пользовательские события jQuery после получения сообщений от сервера, чтобы наши коллекции jQuery могли подписаться на эти события и обработать их.

Сначала модифицируем модель, как показано в листинге 6.5. Изменения выделены **полужирным** шрифтом.

Листинг 6.5 ❖ Добавление средств работы с сообщениями в модель – spa/js/spa.model.js

```

...
completeLogin = function ( user_list ) {
    ...
    stateMap.people_cid_map[ user_map._id ] = stateMap.user;
    chat.join(); ←
    $.gevent.publish( 'spa-login', [ stateMap.user ] );
};
...
people = (function () {
    ...
    logout = function () {
        var is_removed, user = stateMap.user;
        chat._leave(); ←
        is_removed = removePerson( user );
        stateMap.user = stateMap.anon_user;

        $.gevent.publish( 'spa-logout', [ user ] );
        return is_removed;
    };
    ...
})();

// API объекта chat
// -----
// Объект chat доступен по имени spa.model.chat.
// Объект chat предоставляет методы и события для управления
// сообщениями в чате. Ниже перечислены его открытые методы:
// * join() – войти в чат. Настраивает протокол
// взаимодействия чата с сервером, включая публикаторов
// глобальных пользовательских событий 'spa-listchange' и
// 'spa-updatechat'. Если текущий пользователь анонимный,
// то join() возвращает false и больше ничего не делает.
// * get_chatee() – вернуть объект person, представляющий ←
// собеседника пользователя. Если такого нет, возвращается
// null.
// * set_chatee( <person_id> ) – установить в качестве
// собеседника объект person с указанным идентификатором
// person_id. Если person_id не входит в список людей, то
// в качестве собеседника устанавливается null. Если
// указанный человек уже является собеседником, то метод
// возвращает false. Публикуется глобальное пользовательское
// событие 'spa-setchatee'.
// * send_msg( <msg_text> ) – отправить сообщение собеседнику.
// Публикуется глобальное пользовательское событие
// 'spa-updatechat'. Если пользователь анонимный или

```

Включаем в метод completeLogin вызов chat.join(), чтобы пользователь автоматически входил в чат после завершения аутентификации.

Включаем в метод people._logout вызов chat._leave(), чтобы пользователь автоматически выходил из чата по завершении сеанса.

Добавляем документацию по методам get_chatee(), set_chatee() и send_msg().

```

// собеседник равен null, метод возвращает false и больше
// ничего не делает.
// ...
//
// * spa-setchatee – публикуется при установке нового
// собеседника. В качестве данных передается хэш вида
// { old_chatee : <old_chatee_person_object>,
//   new_chatee : <new_chatee_person_object>
// }
// * spa-listchange – публикуется при изменении длины
// списка людей в онлайн (то есть когда кто-то входит в чат
// или выходит из чата) или его содержимого (то есть когда
// изменяется информация о каком-то аватаре).
// Получатель этого события должен запросить обновленные
// данные у метода people_db объекта модели people.
// * spa-updatechat – публикуется при получении или отправке
// сообщения. В качестве данных передается хэш вида
// { dest_id : <chatee_id>,
//   dest_name : <chatee_name>,
//   sender_id : <sender_id>,
//   msg_text : <message_content>
// }
//
chat = (function () {
  var
    _publish_listchange, _publish_updatechat,
    _update_list, _leave_chat,

    get_chatee, join_chat, send_msg, set_chatee,

    chatee = null;

  // Начало внутренних методов
  _update_list = function( arg_list ) {
    var i, person_map, make_person_map,
        people_list = arg_list[ 0 ],
        is_chatee_online = false;
    clearPeopleDb();

    PERSON:
    for ( i = 0; i < people_list.length; i++ ) {
      person_map = people_list[ i ];

      if ( ! person_map.name ) { continue PERSON; }
      // если пользователь определен, обновить
      // css_map и больше ничего не делать
      if ( stateMap.user && stateMap.user.id === person_map._id ) {

```

Добавляем документацию по событиям
spa-setchatee и spa-updatechat.

←

```

stateMap.user.css_map = person_map.css_map;
continue PERSON;
}

make_person_map = {
  cid      : person_map._id,
  css_map  : person_map.css_map,
  id       : person_map._id,
  name     : person_map.name
};

if ( chatee && chatee.id === make_person_map.id ) {
  is_chatee_online = true;
}
makePerson( make_person_map );
}

stateMap.people_db.sort( 'name' );
// Если собеседник уже не в онлайн, сбросить chatee,
// в результате чего возникнет глобальное событие 'spa-setchatee'
if ( chatee && ! is_chatee_online ) { set_chatee(''); }
};

_publish_listchange = function ( arg_list ) {
  _update_list( arg_list );
  $.gevent.publish( 'spa-listchange', [ arg_list ] );
};

_publish_updatechat = function ( arg_list ) {
  var msg_map = arg_list[ 0 ];

  if ( ! chatee ) { set_chatee( msg_map.sender_id ); }
  else if ( msg_map.sender_id !== stateMap.user.id
    && msg_map.sender_id !== chatee.id
  ) { set_chatee( msg_map.sender_id ); }

  $.gevent.publish( 'spa-updatechat', [ msg_map ] );
};
// Конец внутренних методов

_leave_chat = function () {
  var sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();
  chatee = null;
  stateMap.is_connected = false;
  if ( sio ) { sio.emit( 'leavechat' ); }
};

get_chatee = function () { return chatee; };

join_chat = function () {

```

Добавляем код, который устанавливает флаг `is_chatee_online` в `true`, если объект `chatee` присутствует в обновленном списке людей.

Добавляем код, который сбрасывает `chatee` в `null`, если этот объект не найден в обновленном списке людей.

Добавляем вспомогательный метод `_publish_updatechat`. Он публикует событие `spa-updatechat`, сопровождая его хэшем с информацией о сообщении.

Добавляем метод `get_chatee`, который возвращает объект `chatee`.

```

var sio;

if ( stateMap.is_connected ) { return false; }

if ( stateMap.user.get_is_anon() ) {
  console.warn( 'User must be defined before joining chat' );
  return false;
}
Регистрируем метод _publish_updatechat в качестве обработчика сообщений updatechat от сервера. В результате при получении каждого сообщения будет публиковаться событие spa-updatechat.

sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();
sio.on( 'listchange', _publish_listchange );
sio.on( 'updatechat', _publish_updatechat );
stateMap.is_connected = true;
return true;
};

send_msg = function ( msg_text ) {
  var msg_map,
  sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();

  if ( ! sio ) { return false; }
  if ( ! ( stateMap.user && chatee ) ) { return false; }

  msg_map = {
    dest_id : chatee.id,
    dest_name : chatee.name,
    sender_id : stateMap.user.id,
    msg_text : msg_text
  };
  Этот код отменяет отправку сообщения, если отсутствует соединение. Отмена производится и тогда, когда не заданы либо текущий пользователь, либо его собеседник.

  Добавляем код для конструирования хэша, содержащего сообщение и связанную с ним информацию.

  // мы опубликовали updatechat, чтобы можно было показать
  // текущий список сообщений
  _publish_updatechat( [ msg_map ] );
  sio.emit( 'updatechat', msg_map );
  return true;
};
Добавляем код для публикации событий spa-updatechat, чтобы пользователь видел свои сообщения в окне чата.

Добавляем метод set_chatee, который меняет объект chatee на указанный. Если новый объект chatee совпадает с текущим, то метод ничего не делает и возвращает false.

set_chatee = function ( person_id ) {
  var new_chatee;
  new_chatee = stateMap.people_cid_map[ person_id ];
  if ( new_chatee ) {
    if ( chatee && chatee.id === new_chatee.id ) {
      return false;
    }
  }
  else {
    new_chatee = null;
  }
  Добавляем код, который публикует событие spa-setchatee, сопровождая его хэшем, содержащим объекты old_chatee и new_chatee.

  $.gevent.publish( 'spa-setchatee',

```

```

    { old_chatee : chatee, new_chatee : new_chatee }
  );
  chatee = new_chatee;
  return true;
};

return {
  _leave      : _leave_chat,
  get_chatee  : get_chatee,
  join        : join_chat,
  send_msg    : send_msg,
  set_chatee  : set_chatee
};
})();

initModule = function () { ...
};

return {
  initModule : initModule,
  chat       : chat,
  people     : people
};
}());

```

Экспортируем открытые методы: `get_chatee`, `send_msg` и `set_chatee`.

Мы закончили второй этап реализации объекта `chat`, добавив средства работы с сообщениями. Как и раньше, мы хотим проверить сделанное и только потом идти дальше. В следующем разделе мы модифицируем модуль `Fake`, добавив имитацию необходимого нам взаимодействия с сервером.

6.2.5. Модификация модуля `Fake` для имитации работы с сообщениями

Теперь нам предстоит модифицировать модуль `Fake`, чтобы он мог имитировать ответы сервера, необходимые для тестирования методов работы с сообщениями.

- Имитировать ответ на исходящее сообщение `updatechat` путем отправки входящего сообщения `updatechat` от текущего собеседника.
- Имитировать входящее сообщение `updatechat` от пользователя `Wilma`.
- Имитировать ответ на исходящее сообщение `leavechat`, которое посылается, когда пользователь завершает сеанс. В этот момент можно отменить регистрацию обработчиков сообщений.

Изменения, внесенные в модуль `Fake`, выделены в листинге 6.6 **полужирным шрифтом**.

Листинг 6.6 ❖ Добавление подставных сообщений в модуль Fake – spa/js/spa.fake.js

```

...
mockSio = (function () {
  var
    on_sio, emit_sio, emit_mock_msg, ← Добавляем объявление функции отправки подставного
    send_listchange, listchange_idto, сообщения emit_mock_msg.
    callback_map = {};

  on_sio = function ( msg_type, callback ) {
    callback_map[ msg_type ] = callback;
  };

  emit_sio = function ( msg_type, data ) {
    var person_map;

    // отвечаем на событие 'adduser' вызовом
    // 'userupdate' через 3 секунды
    if ( msg_type === 'adduser' && callback_map.userupdate ) {
      setTimeout( function () {
        person_map = {
          _id      : makeFakeId(),
          name     : data.name,
          css_map  : data.css_map
        };
        peopleList.push( person_map );
        callback_map.userupdate([ person_map ]);
      }, 3000 );
    }
    // Добавляем код, который отвечает на посланное сообще-
    // ние подставным с двухсекундной задержкой.

    // отвечаем на событие 'updatechat' вызовом 'updatechat' ←
    // с задержкой 2 с. Возвращаем сведения о пользователе.
    if ( msg_type === 'updatechat' && callback_map.updatechat ) {
      setTimeout( function () {
        var user = spa.model.people.get_user();
        callback_map.updatechat([ {
          dest_id   : user.id,
          dest_name : user.name,
          sender_id : data.dest_id,
          msg_text  : 'Thanks for the note, ' + user.name
        } ]);
      }, 2000 );
    }
    // Добавляем код, который стирает ранее зарегистрированные
    // обратные вызовы при получении сообщения leavechat.
    // Это означает, что пользователь завершил сеанс.

    if ( msg_type === 'leavechat' ) { ←
      // восстанавливаем состояние "не аутентифицирован"
      delete callback_map.listchange;
      delete callback_map.updatechat;

      if ( listchange_idto ) {
        clearTimeout( listchange_idto );
      }
    }
  }
}

```

```

        listchange_idto = undefined;
    }
    send_listchange();
}
};

emit_mock_msg = function () {
    setTimeout( function () {
        var user = spa.model.people.get_user();
        if ( callback_map.updatechat ) {
            callback_map.updatechat([
                {
                    dest_id : user.id,
                    dest_name : user.name,
                    sender_id : 'id_04',
                    msg_text : 'Hi there ' + user.name + '! Wilma here.'
                }
            ]);
        }
        else { emit_mock_msg(); }
    }, 8000 );
};

// Пытаемся воспользоваться обработчиком listchange один
// раз в секунду. Прекращаем попытки после первой удачной.
send_listchange = function () {
    listchange_idto = setTimeout( function () {
        if ( callback_map.listchange ) {
            callback_map.listchange([ peopleList ]);
            emit_mock_msg();
            listchange_idto = undefined;
        }
        else { send_listchange(); }
    }, 1000 );
};

// Мы должны запустить процесс ...
send_listchange();

return { emit : emit_sio, on : on_sio };
})();

return { mockSio : mockSio };
})();

```

Добавляем код, который пытается послать подставное сообщение аутентифицированному пользователю каждые 8 секунд. Попытка завершится успехом только после того, как пользователь аутентифицируется и будет зарегистрирован обратный вызов updatechat. В случае успеха эта функция не вызывает себя, так что попытки послать подставное сообщение прекращаются.

Добавляем код, который начинает процесс пробной отправки подставного сообщения после аутентификации пользователя.

Модифицировав объект chat и модуль Fake, мы можем протестировать работу с сообщениями.

6.2.6. Тестирование работы с сообщениями в чате

Протестируем установку собеседника, отправку и получение сообщений. Загрузите файл spa/spa.html в браузер, откройте консоль JavaScript и убедитесь, что SPA не содержит ошибок JavaScript. Далее

можно протестировать новую функциональность, как показано в листинге 6.7. Вводимые нами команды выделены **полужирным** шрифтом, получаемые ответы – *курсивом*.

Листинг 6.7 ❖ Тестирование обмена сообщениями

```
// создать коллекцию jQuery
var $t = $('<div/>');
```

Создаем коллекцию jQuery (\$t), не присоединенную к документу в браузере. Ниже мы воспользуемся ей для тестирования событий.

```
// привязать функции для тестирования глобальных событий
$.gevent.subscribe( $t, 'spa-login', function( event, user ) {
    console.log('Hello!', user.name); });

$.gevent.subscribe( $t, 'spa-updatechat', function( event, chat_map ) {
    console.log( 'Chat message:', chat_map);
});

$.gevent.subscribe( $t, 'spa-setchatee',
    function( event, chatee_map ) {
        console.log( 'Chatee change:', chatee_map);
    });

$.gevent.subscribe( $t, 'spa-listchange',
    function( event, changed_list ) {
        console.log( '*Listchange:', changed_list );
    });

// аутентифицироваться, ждать 3 с
spa.model.people.login( 'Fanny' );
>> Hello! Fanny
>> *Listchange: [Array[5]]
```

Подписываем коллекцию \$t на событие spa-login, указав функцию, которая печатает на консоли строку «Hello!» и список аргументов.

Подписываем коллекцию \$t на событие spa-updatechat, указав функцию, которая печатает на консоли строку «Chat message:» и chat_map.

Подписываем коллекцию \$t на событие spa-setchatee, указав функцию, которая печатает на консоли строку «Chatee change:» и chatee_map.

Подписываем коллекцию \$t на событие spa-listchange, указав функцию, которая печатает на консоли строку «*Listchange:» и changed_list.

Аутентифицируемся как Fanny.

Через три секунды публикуется событие spa-login, что приводит к вызову функции, зарегистрированной для этого события в коллекции \$t.

Публикуется также событие spa-listchange, что приводит к вызову функции, зарегистрированной для него в коллекции \$t.

```
// попытка отправить сообщение, не установив собеседника
spa.model.chat.send_msg( 'Hi Pebbles!' );
>> false
```

Пытаемся послать сообщение, не установив chatee. Это нужно сделать в течение 8 секунд, до получения сообщения от Wilma.

Этот метод возвращает false, потому что получатель еще не установлен.

```
// ждать прихода тестового сообщения в течение 8 секунд
>> Chatee change: Object {old_chatee: null, new_chatee: Object}
>> Chat message: Object {dest_id: "id_5", dest_name: "Fanny",
>> sender_id: "id_04", msg_text: "Hi there Fanny! Wilma here."}
```

Через несколько секунд публикуется событие spa-setchatee, что приводит к вызову функции, зарегистрированной для него в коллекции \$t.

Публикуется событие spa-updatechat, что приводит к вызову функции, зарегистрированной для него в коллекции \$t.

```
// получение сообщения устанавливает собеседника
spa.model.chat.send_msg( 'What is up, tricks?' );
>> Chat message: Object {dest_id: "id_04", dest_name: "Wilma",
```



```

>> sender_id: "id_5", msg_text: "What is up tricks?"}
>> true
// Сделать собеседником Pebbles
spa.model.chat.set_chatee( 'id_03' );
>> Chatee change: Object {old_chatee: Object, new_chatee: Object}
>> true
// Послать сообщение
spa.model.chat.send_msg( 'Hi Pebbles!' )
>> Chat message: Object {dest_id: "id_03", dest_name: "Pebbles",
>> sender_id: "id_5", msg_text: "Hi Pebbles!"}
>> true
>> Chat message: Object {dest_id: "id_5", dest_name: "Fanny",
>> sender_id: "id_03", msg_text: "Thanks for the note, Fanny"}

```

Посылаем сообщение «What is up tricks?» собеседнику. Это последний, кто отправлял сообщения текущему пользователю.

Этот метод в случае успеха возвращает true.

Мы видим ответ на наше сообщение; публикуется событие `spa-updatechat`, что приводит к вызову функции, зарегистрированной для него в коллекции `$t`.

Устанавливаем в качестве собеседника пользователя с идентификатором `id_03`.

Убеждаемся, что метод `set_chatee` вернул true, то есть завершился успешно.

Публикуется событие `spa-setchatee`.

Посылаем сообщение «Hi Pebbles!» нашему текущему собеседнику Pebbles.

Получен еще один автоматический ответ.

Этот метод в случае успеха возвращает true.

Публикуется событие `spa-updatechat`, что приводит к вызову функции, зарегистрированной для него в коллекции `$t`.

Наш объект `chat` почти готов. Осталось только добавить поддержку аватаров. После этого мы обновим пользовательский интерфейс.

6.3. Добавление поддержки аватаров в модель

Средства работы с аватарами добавить сравнительно просто, потому что мы можем опираться на инфраструктуру работы с сообщениями, реализованную в объекте `chat`. Мы решили сделать это главным образом для того, чтобы показать другие применения обмена сообщениями в режиме, близком к реальному времени. А тот факт, что эта штука выигрышно смотрится на конференциях, – просто вишенка на торте. Для начала обновим модель.

6.3.1. Добавление поддержки аватаров в объект `chat`

Для поддержки аватаров в объект `chat` нужно внести сравнительно небольшие изменения. Нам понадобится лишь метод `update_avatar`,

который будет посылать серверу сообщение `updateavatar`, сопровождая его информацией о том, какой аватар изменился и как именно. Мы ожидаем, что при изменении аватара сервер отправит сообщение `listchange`, а код для его обработки уже написан и протестирован.

Изменения в модуле Model выделены в листинге 6.8 **полужирным** шрифтом.

Листинг 6.8 ❖ Добавление поддержки аватаров в модель – `spa/js/spa.model.js`

```
...
// Публикуется глобальное пользовательское событие
// 'spa-updatechat'. Если пользователь анонимный или
// собеседник равен null, метод возвращает false и больше
// ничего не делает.
// * update_avatar( <update_avtr_map> ) – отправить серверу
// хэш update_avtr_map. В результате публикуется событие
// 'spa-listchange', сопровождаемое обновленным списком
// людей и аватаров (атрибут css_map для каждого человека).
// Хэш update_avtr_map должен иметь вид
// { person_id : person_id, css_map : css_map }.
//
// Ниже перечислены глобальные пользовательские события
// jQuery, публикуемые объектом:
...

chat = (function () {
  var
    _publish_listchange, _publish_updatechat,
    _update_list, _leave_chat,

    get_chatee, join_chat, send_msg,
    set_chatee, update_avatar,

    chatee = null;

  ...

  // avatar_update_map должен иметь вид:
  // { person_id : <string>, css_map : {
  //   top : <int>, left : <int>,
  //   'background-color' : <string>
  // } };
  //
  //
  update_avatar = function ( avatar_update_map ) {
    var sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();
    if ( sio ) {
      sio.emit( 'updateavatar', avatar_update_map );
    }
  };
};
```

Добавляем документацию из спецификации API.

Объявляем локальную переменную `update_avatar`.

Добавляем метод `update_avatar`. В нем мы посылаем серверу сообщение `updateavatar`, включая хэш в качестве данных.

```

return {
  _leave      : _leave_chat,
  get_chatee  : get_chatee,
  join        : join_chat,
  send_msg    : send_msg,
  set_chatee  : set_chatee,
  update_avatar : update_avatar ← Добавляем update_avatar в список
};                                     экспортируемых открытых методов.
}());
...

```

Вот мы и реализовали все методы и события, включенные в спецификацию объекта `chat`. В следующем разделе мы модифицируем модуль `Fake` с целью имитации взаимодействия с сервером для поддержки аватаров.

6.3.2. Модификация модуля `Fake` для имитации аватаров

Следующий наш шаг – модифицировать модуль `Fake`, так чтобы он отправлял серверу сообщение `updateavatar`, когда пользователь перетаскивает аватар в новое место или щелкает по нему для изменения цвета. Получив такое сообщение, модуль `Fake` должен:

- имитировать отправку сообщения `updateavatar` серверу;
- имитировать получение от сервера сообщения `listchange` с обновленным списком людей;
- выполнить обработчик, зарегистрированный для сообщения `listchange`, передав ему обновленный список людей.

Эти три шага можно реализовать, как показано в листинге 6.9. Изменения выделены **полужирным** шрифтом.

Листинг 6.9 ❖ Модификация модуля `Fake` для поддержки аватаров – `spa/js/spa.fake.js`

```

...
emit_sio = function ( msg_type, data ) {
  var person_map, i; ← Объявляем переменную цикла i.
...
  if ( msg_type === 'leavechat' ) {
    // восстанавливаем состояние "не аутентифицирован"
    delete callback_map.listchange;
    delete callback_map.updatechat;

    if ( listchange_idto ) {
      clearTimeout( listchange_idto );
      listchange_idto = undefined;
    }
  }
}

```

```

        send_listchange();
    }
};

// имитируем отправку серверу сообщения 'updateavatar' с данными
if ( msg_type === 'updateavatar' && callback_map.listchange ) {
    // имитируем получение сообщения 'listchange'
    for ( i = 0; i < peopleList.length; i++ ) {
        if ( peopleList[ i ]._id === data.person_id ) {
            peopleList[ i ].css_map = data.css_map;
            break;
        }
    }
    // вызываем обработчик события 'listchange'
    callback_map.listchange( peopleList );
};
...

```

Создаем обработчик получения сообщения updateavatar.

Находим объект person с идентификатором, указанным в сообщении updateavatar, и изменяем его свойство css_map.

Выполняем функцию обратного вызова, зарегистрированную для обработки события listchange.

Доработав объект chat и модуль Fake, мы можем протестировать аватары.

6.3.3. Тестирование поддержки аватаров

Это последний аккорд в тестировании модели. Снова загрузите файл spa/spa.html в браузер и убедитесь, что SPA работает, как и раньше. Откройте консоль JavaScript и протестируйте метод update_avatar, как показано в листинге 6.10. Вводимые нами команды выделены **полужирным** шрифтом, получаемые ответы – *курсивом*.

Листинг 6.10 ❖ Тестирование метода update_avatar

```

// создать коллекцию jQuery
var $t = $('<div/>');

// привязать функции для тестирования глобальных событий
$.gevent.subscribe( $t, 'spa-login', function( event, user ) {
    console.log('Hello!', user.name); });
$.gevent.subscribe( $t, 'spa-listchange',
    function( event, changed_list ) {
        console.log( '*Listchange:', changed_list );
    });

// аутентифицироваться, ждать 3 с
spa.model.people.login( 'Jessy' );
>> Hello! Jessy
>> *Listchange: [Array[5]]

// получить пользователя Pebbles

```

Создаем коллекцию jQuery (\$t), не присоединенную к документу в браузере. Ниже мы воспользуемся ей для тестирования событий.

Подписываем коллекцию \$t на событие spa-login, указав функцию, которая выводит на консоль.

Подписываем коллекцию \$t на событие spa-listchange, указав функцию, которая печатает на консоли строку «*Listchange» и changed_list.

Аутентифицируемся как Jessy.

Через три секунды публикуется событие spa-login, что приводит к вызову функции, зарегистрированной для этого события в коллекции \$t.

Публикуется также событие spa-listchange, что приводит к вызову функции, зарегистрированной для него в коллекции \$t.

Получаем объект person с идентификатором id_03 – Pebbles.

```

var person = spa.model.people.get_by_cid( 'id_03' );

// вывести сведения об аватаре ←———— Выводим сведения об аватаре для объекта Pebbles.
JSON.stringify( person.css_map );
>> '{"top":100,"left":20,
>> "background-color":"rgb( 128, 192, 192)"}'

// обновить сведения об аватаре ←———— Вызываем метод update_avatar для изменения
spa.model.chat.update_avatar({           атрибута css_map объекта Pebbles.
  person_id : 'id_03', css_map : {} });
>> *Listchange: [Array[5]] ←———— Убеждаемся, что метод update_avatar публикует событие spa-
                                listchange, в ответ на которое вызывается функция, зареги-
                                стрированная для этого события в коллекции $t.

// снова получить Pebbles
person = spa.model.people.get_by_cid( 'id_03' );

// и проинспектировать ←———— Обновленное свойство css_map для объекта Pebbles.
JSON.stringify( person.css_map );
>> {}

```

Мы закончили реализацию объекта `chat`. Как и в случае объекта `people` из главы 5, тестирование вселяет уверенность, и мы можем пополнить комплект тестов, не требующих ни сервера, ни браузера.

6.3.4. Разработка через тестирование

Фанаты разработки через тестирование (TDD), наверное, смотрят на все это копошение вручную и думают: «Блин, ну почему не засунуть это в комплект тестов и не прогонять его автоматически?» Мы и сами такие же упертые фанаты, поэтому так и поступили. Загляните в приложение Б, и вы узнаете, как с помощью Node.js автоматизировать процесс.

На самом деле при прогоне комплекта тестов обнаружилось несколько проблем. По большей части они относились к самому процессу тестирования, и их мы отложим до упомянутого приложения. Однако есть и две настоящие ошибки, нуждающиеся в исправлении: механизм завершения сеанса неправильно очищает список пользователей, а объект `chatee` некорректно обновляется после вызова метода `spa.model.chat.update_avatar`. В листинге 6.11 обе ошибки исправлены, изменения выделены **полужирным** шрифтом.

Листинг 6.11 ❖ Исправление ошибок при завершении сеанса и обновлении объекта `chatee` – `spa/js/spa.model.js`

```

...
people = (function () {
  ...
  logout = function () {

```

```

var user = stateMap.user; ← Убираем переменную is_removed.

chat._leave();
stateMap.user = stateMap.anon_user;
clearPeopleDb(); ← Очищаем TaffyDB-коллекцию людей при завершении сеанса.

$.gevent.publish( 'spa-logout', [ user ] );
return is_removed;
};
...
})();

chat = (function () {
  var
    ...
    // Начало внутренних методов
    _update_list = function( arg_list ) {
      var i, person_map, make_person_map, person, ← Объявляем объект person.
      people_list = arg_list[ 0 ],
      is_chatee_online = false;

      clearPeopleDb();

      PERSON:
      for ( i = 0; i < people_list.length; i++ ) {
        person_map = people_list[ i ];

        if ( ! person_map.name ) { continue PERSON; }
        // если пользователь определен, обновить
        // css_map и больше ничего не делать
        if ( stateMap.user && stateMap.user.id === person_map.id ) {
          stateMap.user.css_map = person_map.css_map;
          continue PERSON;
        }

        make_person_map = {
          cid      : person_map.id,
          css_map  : person_map.css_map,
          id       : person_map.id,
          name     : person_map.name
        };
        person = makePerson( make_person_map ); ← Присваиваем результат выполнения
                                                makePerson переменной person.

        if ( chatee && chatee.id === make_person_map.id ) {
          is_chatee_online = true;
          chatee = person; ← Если собеседник найден, записываем в chatee
                           новый объект person.
        }
      }

      stateMap.people_db.sort( 'name' );

```

```

// Если собеседник уже не в онлайн, сбросить chatee,
// в результате чего возникнет глобальное событие 'spa-setchatee'
if ( chatee && ! is_chatee_online ) { set_chatee(''); }
};
...
}());
...

```

Можно сделать перерыв. В оставшейся части этой главы мы вернемся к пользовательскому интерфейсу и с помощью API объектов модели `chat` и `people` закончим реализацию функционального модуля. Кроме того, мы напишем функциональный модуль `Avatar`.

6.4. Завершение функционального модуля Chat

В этом разделе мы модифицируем функциональный модуль `Chat`, изображенный на рис. 6.3. Теперь мы можем воспользоваться объектами модели `chat` и `people`, чтобы имитировать поведение пользователя в чате. Вернемся к намеченному ранее пользовательскому интерфейсу модуля `Chat` и решим, как нужно его модифицировать для работы с объектом `chat`. На рис. 6.4 показано, что мы хотим получить в итоге. Составим на основе этого эскиза список изменений, которые желательно внести в модуль `Chat`.

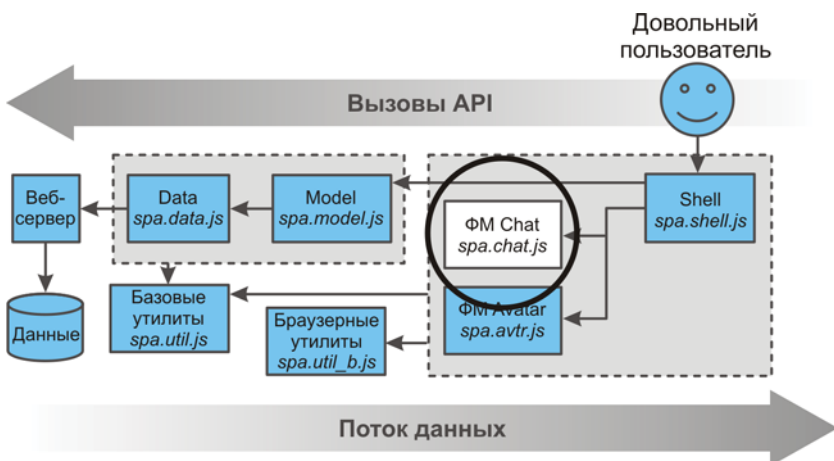


Рис. 6.3 ❖ Место функционального модуля `Chat` в нашей архитектуре SPA

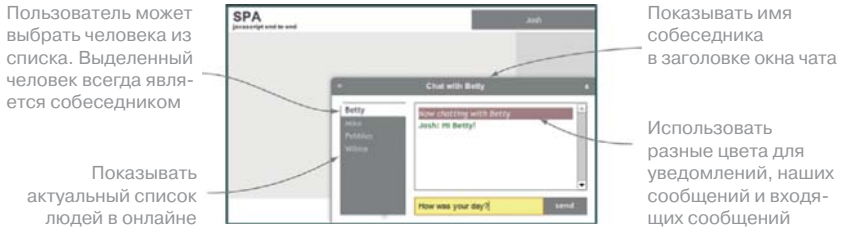


Рис. 6.4 ❖ Каким мы хотим видеть пользовательский интерфейс чата

- Изменить дизайн окна чата, включив в него список людей.
- После аутентификации пользователя выполнить следующие действия: войти в чат, открыть всплывающее окно чата, изменить текст в заголовке окна и вывести список людей в онлайн.
- Обновлять отображаемый список людей в онлайн, когда он изменяется.
- Подсвечивать собеседника в списке людей.
- Дать пользователю возможность посылать сообщения и выбирать собеседника из списка людей в онлайн.
- Отображать сообщения самого пользователя, других людей и системы в области сообщений. Сообщения разных типов должны выглядеть по-разному, а окно должно плавно прокручиваться снизу вверх.
- Добавить в интерфейс поддержку сенсорных элементов управления.
- При завершении сеанса выполнять следующие действия: изменить текст в заголовке окна чата, очистить область сообщений и свернуть всплывающее окно.

Начнем с изменения JavaScript-кода.

6.4.1. Модификация JavaScript-кода модуля Chat

Чтобы реализовать вышеупомянутые изменения, в JavaScript-код модуля Chat нужно внести следующие модификации.

- Включить список людей в HTML-шаблон.
- Написать методы `scrollChat`, `writeChat`, `writeAlert` и `clearChat` для управления областью сообщений.
- Написать обработчики событий ввода `onTapList` и `onSubmitMsg`, чтобы пользователь мог выбирать собеседника из списка и отправлять сообщения. Не забыть поддержать сенсорный ввод.

- Написать метод `onSetchatee` для обработки публикуемого моделью события `spa-setchatee`. Этот метод должен изменить внешний вид собеседника, заменить текст в заголовке окна чата и вывести в область сообщений уведомление от системы.
- Написать метод `onListchange` для обработки публикуемого моделью события `spa-listchange`. Он должен отображать список людей, в котором собеседник подсвечен.
- Написать метод `onUpdatechat` для обработки публикуемого моделью события `spa-updatechat`. Он отображает новые сообщения пользователя, сервера или других людей.
- Написать методы `onLogin` и `onLogout` для обработки публикуемых моделью событий `spa-login` и `spa-logout`. Обработчик `onLogin` должен открыть всплывающее окно чата после успешной аутентификации. Обработчик `onLogout` должен очистить область сообщений, восстановить текст заголовка и закрыть всплывающее окно.
- Подписаться на все публикуемые моделью события и привязать обработчики событий ввода.

Об именах обработчиков событий

Мы уверены, что некоторые читатели недоумевают: «Зачем называть метод `onSetchatee`, а не `onSetChatee`?». Причина есть.

Следуя нашему соглашению, обработчик события должен называться `on<Event>[<Modifier>]`, где часть *Modifier* необязательна. Обычно проблем не возникает, потому что имена большинства событий односложные, например `onTap` или `onTapAvatar`. Это соглашение удобно, потому что, глядя на код, читатель сразу понимает, к какому *событию* относится обработчик.

Но, как всегда бывает с соглашениями, существуют случаи, которые плохо укладываются в схему. Так, имя `onListchange` устроено в строгом соответствии с соглашением: событие называется `listchange`, а *не* `listChange`. Поэтому `onListchange` правильно, а `onListChange` – нет. То же самое относится к именам `onSetchatee` и `onUpdatechat`.

Внесем в JavaScript-файл изменения, выделенные в листинге 6.12 **полужирным** шрифтом.

Листинг 6.12 ❖ Модификация JavaScript-кода модуля Chat – `spa/js/spa.chat.js`

```
...
/*global $, spa */
spa.chat = (function () {
  'use strict';
  //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
```

Удаляем `getComputedStyle` из списка глобальных символов. Эта функция использовалась в методе `getEmSize`, который перенесен в модуль браузерных утилит.

Добавляем прагму `use strict`.

```

var
  configMap = {
    main_html : String()
      + '<div class="spa-chat">'
        + '<div class="spa-chat-head">'
          + '<div class="spa-chat-head-toggle">x</div>'
          + '<div class="spa-chat-head-title">'
            + 'Chat'
          + '</div>'
        + '</div>'
        + '<div class="spa-chat-closer">x</div>'
        + '<div class="spa-chat-size">'
          + '<div class="spa-chat-list">'
            + '<div class="spa-chat-list-box"></div>'
          + '</div>'
          + '<div class="spa-chat-msg">'
            + '<div class="spa-chat-msg-log"></div>'
            + '<div class="spa-chat-msg-in">'
              + '<form class="spa-chat-msg-form">'
                + '<input type="text"/>'
                + '<input type="submit" style="display:none"/>'
                + '<div class="spa-chat-msg-send">'
                  + 'send'
                + '</div>'
              + '</form>'
            + '</div>'
          + '</div>'
        + '</div>'
      + '</div>',
    ...
    slider_closed_em : 2,
    slider_opened_title : 'Tap to close',
    slider_closed_title : 'Tap to open',
    slider_opened_min_em : 10,
    ...
  },
  ...
  jQueryMap, setPxSizes, scrollChat,
  writeChat, writeAlert, clearChat,
  setSliderPosition,
  onTapToggle, onSubmitMsg, onTapList,
  onSetchatee, onUpdatechat, onListchange,
  onLogin, onLogout,
  configModule, initModule,
  removeSlider, handleResize;
//----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

//----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

```

Включаем в шаблон чата список людей и другие улучшения.

Изменяем click на tap, чтобы было понятно обладателям сенсорных устройств.

Объявляем новые методы для обработки событий модели и ввода.

Удаляем метод `getEmSize`, который отныне будет находиться среди браузерных утилит (`spa.util_b.js`).

```
//----- НАЧАЛО МЕТОДОВ DOM -----
// Начало метода DOM /setJqueryMap/
setJqueryMap = function () {
    var
        $append_target = stateMap.$append_target,
        $slider         = $append_target.find( '.spa-chat' );

    jqueryMap = {
        $slider : $slider,
        $head   : $slider.find( '.spa-chat-head' ),
        $toggle  : $slider.find( '.spa-chat-head-toggle' ),
        $title   : $slider.find( '.spa-chat-head-title' ),
        $sizer   : $slider.find( '.spa-chat-sizer' ),
        $list_box : $slider.find( '.spa-chat-list-box' ),
        $msg_log  : $slider.find( '.spa-chat-msg-log' ),
        $msg_in   : $slider.find( '.spa-chat-msg-in' ),
        $input    : $slider.find( '.spa-chat-msg-in input[type=text]' ),
        $send     : $slider.find( '.spa-chat-msg-send' ),
        $form     : $slider.find( '.spa-chat-msg-form' ),
        $window   : $(window)
    };
};
// Конец метода DOM /setJqueryMap/

// Начало метода DOM /setPxSizes/
setPxSizes = function () {
    var px_per_em, window_height_em, opened_height_em;

    px_per_em = spa.util_b.getEmSize( jqueryMap.$slider.get(0) );
    window_height_em = Math.floor(
        ( jqueryMap.$window.height() / px_per_em ) + 0.5
    );
    ...
}
...

// Начало открытого метода /setSliderPosition/
...
setSliderPosition = function ( position_type, callback ) {
    var
        height_px, animate_time, slider_title, toggle_text;

    // состояние 'opened' для анонимного пользователя запрещено,
    // поэтому мы просто возвращаем false; Shell изменит этот
    // uri и попытает еще раз.
    if ( position_type === 'opened'
        && configMap.people_model.get_user().get_is_anon()
    ){ return false; }
```

Включаем в кэш коллекций jQuery новые элементы окна чата.

Берем метод getEmSize из модуля браузерных утилит.

Получаем коллекцию jQuery для элемента window из кэша jQuery.

Добавляем код, запрещающий открывать выпадающий чат анонимному пользователю. Обратный вызов Shell соответственно подправит URI.

```

// вернуть true, если окно чата уже находится в требуемом состоянии
if ( stateMap.position_type === position_type ){
    if ( position_type === 'opened' ) {
        jqueryMap.$input.focus();
    }
    return true;
}

// подготовить параметры анимации
switch ( position_type ){
    case 'opened' :
        ...
        jqueryMap.$input.focus();
        break;
    ...
}
...
};
// Конец открытого метода /setSliderPosition/

// Начало закрытых методов DOM для управления областью сообщений чата
scrollChat = function() {
    var $msg_log = jqueryMap.$msg_log;
    $msg_log.animate(
        { scrollTop : $msg_log.prop( 'scrollHeight' )
          - $msg_log.height()
        },
        150
    );
};

writeChat = function ( person_name, text, is_user ) {
    var msg_class = is_user
        ? 'spa-chat-msg-log-me' : 'spa-chat-msg-log-msg';

    jqueryMap.$msg_log.append(
        '<div class="' + msg_class + '">'
        + spa.util_b.encodeHtml(person_name) + ': '
        + spa.util_b.encodeHtml(text) + '</div>'
    );

    scrollChat();

    writeAlert = function ( alert_text ) {
        jqueryMap.$msg_log.append(
            '<div class="spa-chat-msg-log-alert">'
            + spa.util_b.encodeHtml(alert_text)
            + '</div>'
        );
    };
}

```

Добавляем код, который передает фокус окну ввода после открытия выплывающего чата.

Отсюда начинаются методы DOM для манипуляции областью сообщений.

Метод scrollChat реализует плавную прокрутку области сообщений при добавлении в нее нового текста.

Метод writeChat добавляет новое сообщение в область сообщений. Если автором является пользователь, используем специальный стиль. Не забываем кодировать отображаемый HTML-код.

Метод writeAlert добавляет уведомление от системы в область сообщений. Не забываем кодировать отображаемый HTML-код.

```

    scrollChat();
};

clearChat = function () { jqueryMap.$msg_log.empty(); };
// Конец закрытых методов DOM для управления областью сообщений чата
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
onTapToggle = function ( event ) {
    ...
};
onSubmitMsg = function ( event ) {
    var msg_text = jqueryMap.$input.val();
    if ( msg_text.trim() === '' ) { return false; }
    configMap.chat_model.send_msg( msg_text );
    jqueryMap.$input.focus();
    jqueryMap.$send.addClass( 'spa-x-select' );
    setTimeout(
        function () { jqueryMap.$send.removeClass( 'spa-x-select' ); },
        250
    );
    return false;
};
onTapList = function ( event ) {
    var $stapped = $( event.elem_target ), chatee_id;
    if ( ! $stapped.hasClass( 'spa-chat-list-name' ) ) { return false; }

    chatee_id = $stapped.attr( 'data-id' );
    if ( ! chatee_id ) { return false; }

    configMap.chat_model.set_chatee( chatee_id );
    return false;
};
onSetchatee = function ( event, arg_map ) {
    var
        new_chatee = arg_map.new_chatee,
        old_chatee = arg_map.old_chatee;

    jqueryMap.$input.focus();
    if ( ! new_chatee ) {
        if ( old_chatee ) {
            writeAlert( old_chatee.name + ' has left the chat' );
        }
        else {
            writeAlert( 'Your friend has left the chat' );
        }
    }
}

```

Здесь заканчиваются методы DOM для манипуляции областью сообщений.

Метод clearChat очищает область сообщений.

Располагаем обработчики пользовательских событий в начале этой секции, а обработчики событий модели – в конце.

Переименовываем обработчик onClickToggle в onTapToggle.

Метод onSubmitMsg обрабатывает событие нажатия на кнопку отправки сообщения пользователем. Сама отправка производится методом model.chat.send_msg.

Метод onTapList обрабатывает событие щелчка или касания имени человека в списке. Установка собеседника производится методом model.chat.set_chatee.

Метод onSetchatee обрабатывает публикуемое моделью событие spa-setchatee. Он выделяет нового собеседника и снимает выделение со старого. Кроме того, он изменяет текст в заголовке окна чата и уведомляет пользователя о смене собеседника.

```

    }
    jqueryMap.$title.text( 'Chat' );
    return false;
}

jqueryMap.$list_box
    .find( '.spa-chat-list-name' )
    .removeClass( 'spa-x-select' )
    .end()
    .find( '[data-id=' + arg_map.new_chatee.id + ']' )
    .addClass( 'spa-x-select' );

writeAlert( 'Now chatting with ' + arg_map.new_chatee.name );
jqueryMap.$title.text( 'Chat with ' + arg_map.new_chatee.name );
return true;
};

onListchange = function ( event ) {
    var
        vlist_html = String(),
        people_db = configMap.people_model.get_db(),
        chatee = configMap.chat_model.get_chatee();

    people_db().each( function ( person, idx ) {
        var select_class = '';
        if ( person.get_is_anon() || person.get_is_user() ) { return true; }

        if ( chatee && chatee.id === person.id ) {
            select_class= 'spa-x-select';
        }
        list_html
            += '<div class="spa-chat-list-name'
            + select_class + '" data-id="' + person.id + '">'
            + spa.util_b.encodeHtml( person.name ) + '</div>';
    });

    if ( ! list_html ) {
        list_html = String()
            + '<div class="spa-chat-list-note">'
            + 'To chat alone is the fate of all great souls...<br><br>'
            + 'No one is online'
            + '</div>';
        clearChat();
    }
    // jqueryMap.$list_box.html( list_html );
    jqueryMap.$list_box.html( list_html );
};

onUpdatechat = function ( event, msg_map ) {

```

Метод onListchange обрабатывает публикуемое моделью событие spa-listchange. Он получает текущую коллекцию людей, отображает их список на экране, и если собеседник определен, то подсвечивает его.

Метод onUpdatechat обрабатывает публикуемое событие spa-updatechat. Он обновляет область сообщений. Если сообщение отправлено самим пользователем, то поле ввода очищается и ему заново передается фокус. Кроме того, метод делает отправителя сообщения собеседником.

```

var
    is_user,
    sender_id = msg_map.sender_id,
    msg_text = msg_map.msg_text,
    chatee = configMap.chat_model.get_chatee() || {},
    sender = configMap.people_model.get_by_cid( sender_id );

if ( ! sender ) {
    writeAlert( msg_text );
    return false;
}

is_user = sender.get_is_user();

if ( ! ( is_user || sender_id === chatee.id ) ) {
    configMap.chat_model.set_chatee( sender_id );
}

writeChat( sender.name, msg_text, is_user );

if ( is_user ) {
    jqueryMap.$input.val( '' );
    jqueryMap.$input.focus();
}
};

onLogin = function ( event, login_user ) {
    configMap.set_chat_anchor( 'opened' );
};

onLogout = function ( event, logout_user ) {
    configMap.set_chat_anchor( 'closed' );
    jqueryMap.$title.text( 'Chat' );
    clearChat();
};

//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----
...
initModule = function ( $append_target ) {
    var $list_box;

    // загрузить html чата и кэш jquery
    stateMap.$append_target = $append_target;
    $append_target.append( configMap.main_html );
    setJqueryMap();
    setPxSizes();

    // установить начальный заголовок и состояние окна чата
    jqueryMap.$toggle.prop( 'title', configMap.slider_closed_title );

```

Метод onLogin обрабатывает публикуемое моделью событие spa-login. Он открывает окно чата.

Метод onLogout обрабатывает публикуемое моделью событие spa-logout. Он очищает область сообщений, восстанавливает текст по умолчанию в заголовке чата и закрывает выплывающее окно.

Изменяем метод initModule, так чтобы он добавлял модифицированный шаблон выплывающего чата в конец контейнера, указанного в вызывающей программе.

```

stateMap.position_type = 'closed';

// Подписать $list_box на глобальные события jQuery ←
$list_box = jqueryMap.$list_box;
$.gevent.subscribe( $list_box, 'spa-listchange', onListchange );
$.gevent.subscribe( $list_box, 'spa-setchatee', onSetchatee );
$.gevent.subscribe( $list_box, 'spa-updatechat', onUpdatechat );
$.gevent.subscribe( $list_box, 'spa-login', onLogin );
$.gevent.subscribe( $list_box, 'spa-logout', onLogout );

// привязать обработчики событий ввода ←
jqueryMap.$head.bind( 'utap', onTapToggle );
jqueryMap.$list_box.bind( 'utap', onTapList );
jqueryMap.$send.bind( 'utap', onSubmitMsg );
jqueryMap.$form.bind( 'submit', onSubmitMsg );
};
// Конец открытого метода /initModule/
...

```

Сначала подписываемся на все события, публикуемые моделью.

Затем привязываем обработчики событий ввода. Если бы привязка осуществлялась до подписки, то могло бы возникнуть состояние гонки.

Вы и система шаблонов

В нашем SPA для генерации HTML-разметки используется простая конкатенация строк – и нас это вполне устраивает. Но рано или поздно возникает потребность в более развитом механизме генерации HTML. Тогда имеет смысл подумать о системе шаблонов.

Любая система шаблонов преобразует данные в отображаемые элементы. Грубо такие системы можно классифицировать по языку, которым разработчик пользуется для управления генерацией. *Внедряемые* системы позволяют внедрять базовый язык – в нашем случае JavaScript – прямо в шаблон. *Инструментальные* системы предоставляют предметно-ориентированный язык (DSL), не зависящий от базового.

Мы не рекомендуем использовать внедряемые системы, потому что таким образом очень легко смешать бизнес-логику с логикой представления. Самой популярной из внедряемых систем, наверное, является метод `template` из библиотеки *underscore.js*, но есть и много иных.

Мы обратили внимание, что в других языках инструментальные системы со временем набирают популярность. Возможно, это объясняется тем, что они поощряют четкое разделение бизнес-логики и представления. Для SPA существует немало хороших инструментальных систем шаблонов. На момент написания этой книги к числу популярных и хорошо протестированных систем такого рода можно было отнести *Handlebars*, *Dust* и *Mustache*. Мы полагаем, что все они достойны вашего внимания.

Разобравшись с JavaScript, займемся доработкой таблиц стилей.

6.4.2. Модификация таблиц стилей

Приведем таблицы стилей в соответствие с обновленным интерфейсом. Прежде всего нужно изменить корневую таблицу, так чтобы протвратить выделение текста в большинстве элементов. Это позволит

избежать вызывающих раздражение эффектов, особенно заметных на сенсорных устройствах. Модифицированная таблица стилей показана в листинге 6.13, изменения выделены **полужирным** шрифтом.

Листинг 6.13 ❖ Модификация корневой таблицы стилей – spa/css/spa.css

```
...
/** Начало установки начальных значений */
...
h1,h2,h3,h4,h5,h6,p { margin-bottom : 6pt; }
ol,ul,dl { list-style-position : inside;}

* {
  -webkit-user-select : none;
  -khtml-user-select : none;
  -moz-user-select : -moz-none;
  -o-user-select : none;
  -ms-user-select : none;
  user-select : none;

  -webkit-user-drag : none;
  -moz-user-drag : none;
  user-drag : none;

  -webkit-tap-highlight-color : transparent;
  -webkit-touch-callout : none;
}

input, textarea, .spa-x-user-select {
  -webkit-user-select : text;
  -khtml-user-select : text;
  -moz-user-select : text;
  -o-user-select : text;
  -ms-user-select : text;
  user-select : text;
}
/** Конец установки начальных значений */
...
```

Добавляем селектор, предотвращающий выделение текста для всех элементов. Мы с нетерпением ждем, когда наконец можно будет избавиться от зависящих от поставщика префиксов: -moz, -khtml, -webkit и им подобных. Без них изменений было бы в шесть раз меньше!

Добавляем селектор, делающий исключение для полей ввода, текстовых областей и любых элементов с классом spa-x-user-select.

Теперь модифицируем таблицу стилей Chat. Основные изменения таковы:

- стилизовать список людей в онлайнe, так чтобы он отображался в левой части окна чата;
- увеличить ширину окна чата с учетом списка людей;
- стилизовать область сообщений;
- удалить все селекторы вида `spa-chat-box*` и `spa-chat-msgs*`;
- добавить стили для сообщений от пользователя, от собеседника и от системы.

Эти изменения выделены в листинге 6.14 **полужирным** шрифтом.

Листинг 6.14 ❖ Модификация таблицы стилей Chat – spa/css/spa.chat.css

```

...
.spa-chat {
    ...
    right : 0;
    width : 32em;
    height : 2em;
    ...
}
...
.spa-chat-sizer {
    position : absolute;
    top : 2em;
    left : 0;
    right : 0;
}

.spa-chat-list {
    position : absolute;
    top : 0;
    left : 0;
    bottom : 0;
    width : 10em;
}

.spa-chat-msg {
    position : absolute;
    top : 0;
    left : 0;
    bottom : 0;
    right : 0;
}

.spa-chat-msg-log,
.spa-chat-list-box {
    position : absolute;
    top : 1em;
    overflow-x : hidden;
}

.spa-chat-msg-log {
    left : 0em;
    right : 1em;
    bottom : 4em;
    padding : 0.5em;
    border : thin solid #888;
    overflow-y : scroll;
}

```

Делаем окно чата на 10 em шире, чтобы поместился список людей.

Создаем класс для стилизации контейнера списка людей; должен располагаться слева и занимать треть ширины окна.

Создаем класс для стилизации контейнера сообщений; должен располагаться слева и занимать две трети ширины окна.

Создаем общие правила для стилизации контейнеров списка людей и списка сообщений.

Добавляем правила стилизации контейнера списка сообщений.

```
.spa-chat-msg-log-msg {  
  background-color : #eee;  
}  
  
.spa-chat-msg-log-me {  
  font-weight : 800;  
  color      : #484;  
}  
  
.spa-chat-msg-log-alert {  
  font-style : italic;  
  background : #a88;  
  color      : #fff;  
}  
  
.spa-chat-list-box {  
  left      : 1em;  
  right     : 1em;  
  bottom    : 1em;  
  overflow-y : auto;  
  border-width : thin 0 thin thin;  
  border-style : solid;  
  border-color : #888;  
  background-color : #888;  
  color      : #ddd;  
  border-radius : 0.5em 0 0 0;  
}  
  
.spa-chat-list-name, .spa-chat-list-note {  
  width : 100%;  
  padding : 0.1em 0.5em;  
}  
  
.spa-chat-list-name {  
  cursor : pointer;  
}  
  
.spa-chat-list-name:hover {  
  background-color : #aaa;  
  color            : #888;  
}  
  
.spa-chat-list-name.spa-x-select {  
  background-color : #fff;  
  color            : #444;  
}  
  
.spa-chat-msg-in {  
  position : absolute;
```

Создаем класс для стилизации обычных сообщений.

Создаем класс для стилизации сообщений, отправленных пользователем.

Создаем класс для стилизации уведомлений от системы.

Добавляем правила для стилизации контейнера списка людей.

Создаем общие правила для стилизации имени участника и единственного уведомления в списке людей.

Добавляем правила для стилизации имени участника в списке людей.

Создаем класс для стилизации области ввода.

```

height    : 2em;
left      : 0em;
right     : 1em;
bottom    : 1em;
border    : thin solid #888;
background : #888;
}

```

```

.spa-chat-msg-in input[type=text] {
  position : absolute;
  width    : 75%;
  height   : 100%;
  line-height : 100%;
  padding  : 0 0.5em;
  border   : 0;
  background : #ddd;
  color    : #666;
}

```

Создаем селектор для стилизации поля ввода внутри области ввода.

```

.spa-chat-msg-in input[type=text]:focus {
  background : #fff8;
  color      : #222;
}

```

Создаем зависимый селектор, который изменяет цвет фона поля ввода на желтый, когда оно находится в фокусе.

```

.spa-chat-msg-send {
  position : absolute;
  top      : 0;
  right    : 0;
  width    : 25%;
  height   : 100%;
  line-height : 1.9em;
  text-align : center;
  color    : #fff;
  font-weight : 800;
  cursor   : pointer;
}

```

Создаем класс для стилизации кнопки отправки.

```

.spa-chat-msg-send:hover,
.spa-chat-msg-send.spa-x-select {
  background : #444;
  color      : #ff0;
}

```

```

.spa-chat-head:hover .spa-chat-head-toggle {
  background : #aaa;
}

```

Имея таблицу стилей, посмотрим, как работает обновленный пользовательский интерфейс чата.

6.4.3. Тестирование пользовательского интерфейса чата

Загрузив файл `spa/spa.html` в браузер, мы должны увидеть страницу, в правом верхнем углу которой находится надпись «Please sign in». Щелкнув по ней, мы, как и раньше, сможем аутентифицироваться. В области пользователя в течение трех секунд будет видно сообщение «... processing ...», а потом оно сменится именем пользователя. В этот момент выплывает окно чата, содержащее интерфейс, показанный на рис. 6.5.

Спустя несколько секунд мы получим первое сообщение от Wilma. Можем ответить на него, а затем выбрать Pebbles и отправить ей сообщение. Окно чата будет выглядеть, как показано на рис. 6.6.

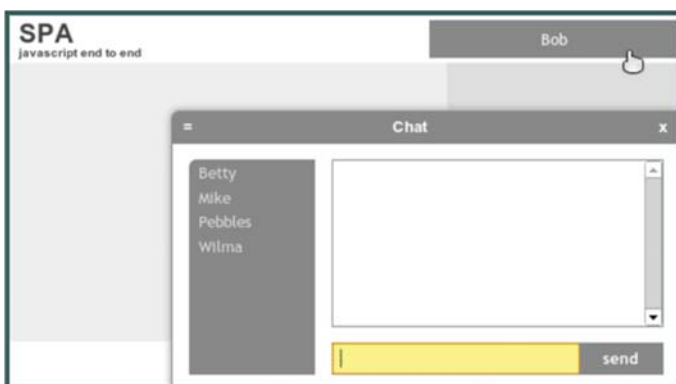


Рис. 6.5 ❖ Обновленный интерфейс чата после аутентификации

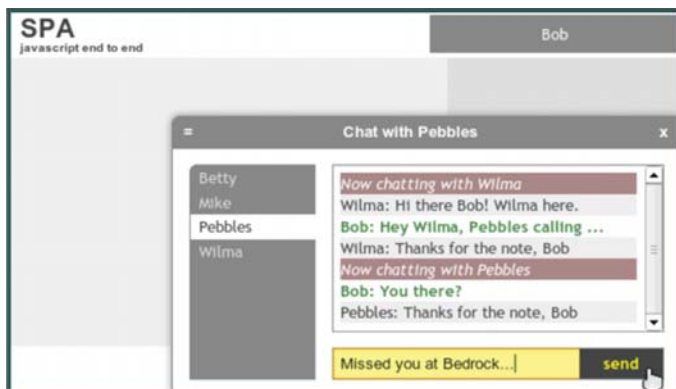


Рис. 6.6 ❖ Окно чата после нескольких операций

С помощью объектов модели `chat` и `people` мы реализовали все запланированные возможности функционального модуля Chat. Теперь хотелось бы добавить функциональный модуль Avatar.

6.5. Разработка функционального модуля Avatar

В этом разделе мы разработаем функциональный модуль Avatar, показанный на рис. 6.7.

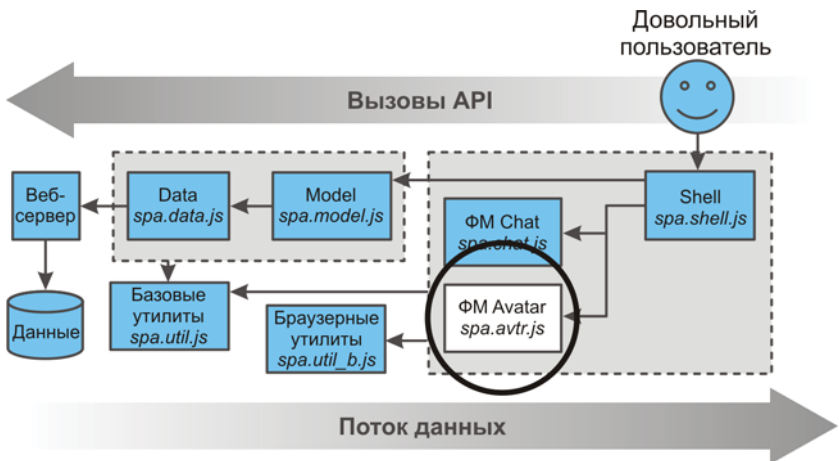


Рис. 6.7 ❖ Место функционального модуля Avatar в нашей архитектуре SPA

Объект `chat` уже предоставляет средства для управления сведениями об аватаре. Нам нужно только определиться с несколькими деталями. Вернемся к пользовательскому интерфейсу модуля Avatar, изображенному на рис. 6.8.

У каждого участника чата имеется аватар, который представлен прямоугольником с жирной рамкой и отображаемым в центре именем. Аватар текущего пользователя должен быть обведен синей рамкой, аватар его собеседника – зеленой. Щелчок по аватару (или касание) должен изменять цвет. В случае долгого касания или длительного удержания мыши внешний вид аватара должен измениться, после чего его можно перетащить в другое место.

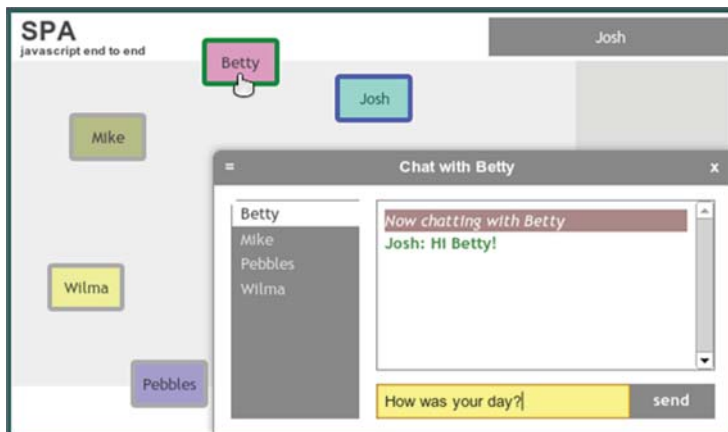


Рис. 6.8 ❖ Предполагаемое представление аватаров

При разработке функционального модуля Avatar мы будем следовать стандартной процедуре:

- создать JavaScript-файл, в котором определено изолированное пространство имен;
- создать таблицу стилей, в которой имена классов начинаются префиксом, совпадающим с именем пространства имен;
- включить новый JavaScript-файл и таблицу стилей в головной HTML-документ;
- добавить в Shell конфигурирование и инициализацию нового модуля.

6.5.1. JavaScript-код модуля Avatar

При создании функционального модуля Avatar мы первым делом напишем JavaScript-код. Поскольку в этом модуле будут использоваться многие события, встречавшиеся нам в модуле Chat, можно скопировать файл `spa/js/spa.chat.js` в `spa/js/spa.avtr.js`, а затем внести необходимые изменения. В листинге 6.15 показан файл свежеспеченного функционального модуля. Поскольку он очень похож на файл модуля Chat, подробное обсуждение мы опустим. Но наиболее интересные места снабжены аннотациями.

Листинг 6.15 ❖ JavaScript-код модуля Avatar – `spa/js/spa.avtr.js`

```
/*
 * spa.avtr.js
 * Функциональный модуль Avatar
```

```

*/
/*jslint      browser : true,    continue : true,
   devel    : true,    indent : 2,      maxerr : 50,
   newcap   : true,    nomen  : true,    plusplus : true,
   regexp  : true,    sloppy  : true,    vars    : false,
   white    : true
*/
/*global $, spa */

spa.avtr = (function () {
  'use strict'; ← Добавляем прагму use strict.
  //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  var
    configMap = {
      chat_model : null, ← Объявляем в конфигурационных свойствах
      people_model : null,  объекты модели people и chat.
      settable_map : {
        chat_model : true,
        people_model : true
      }
    },

    stateMap = { ← Объявляем свойства состояния, которые позволяют отслеживать
      drag_map : null,      перетаскиваемый аватар на протяжении обработки разных событий.
      $drag_target : null,
      drag_bg_color : undefined
    },

    jqueryMap = {},

    getRandRgb,
    setJqueryMap,
    updateAvatar,
    onTapNav,    onHeldstartNav,
    onHeldmoveNav, onHeldendNav,
    onSetchatee, onListchange,
    onLogout,
    configModule, initModule;
  //----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

  //----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
  getRandRgb = function () { ← Служебный метод для генерации случайного цвета в формате RGB.
    var i, rgb_list = [];
    for ( i = 0; i < 3; i++ ){
      rgb_list.push( Math.floor( Math.random() * 128 ) + 128 );
    }
    return 'rgb(' + rgb_list.join(',') + ')';
  };
  //----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

```



```
//----- НАЧАЛО МЕТОДОВ DOM -----
setJqueryMap = function ( $container ) {
    jqueryMap = { $container : $container };
};

updateAvatar = function ( $target ){
    var css_map, person_id;

    css_map = {
        top : parseInt( $target.css( 'top' ), 10 ),
        left : parseInt( $target.css( 'left' ), 10 ),
        'background-color' : $target.css('background-color')
    };
    person_id = $target.attr( 'data-id' );

    configMap.chat_model.update_avatar({
        person_id : person_id, css_map : css_map
    });
};
//----- КОНЕЦ МЕТОДОВ DOM -----

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
onTapNav = function ( event ){
    var css_map,
        $target = $( event.elem_target ).closest('.spa-avtr-box');

    if ( $target.length === 0 ){ return false; }
    $target.css({ 'background-color' : getRandRgb() });
    updateAvatar( $target );
};

onHeldstartNav = function ( event ){
    var offset_target_map, offset_nav_map,
        $target = $( event.elem_target ).closest('.spa-avtr-box');

    if ( $target.length === 0 ){ return false; }

    stateMap.$drag_target = $target;
    offset_target_map = $target.offset();
    offset_nav_map = jqueryMap.$container.offset();

    offset_target_map.top -= offset_nav_map.top;
    offset_target_map.left -= offset_nav_map.left;

    stateMap.drag_map = offset_target_map;
    stateMap.drag_bg_color = $target.css( 'background-color' );

    $target
        .addClass('spa-x-is-drag')
```

Метод `updateAvatar` для чтения значений `css` из переданного аватара `$target` и последующего вызова метода `model.chat.update_avatar`.

Метод `onTapNav` – обработчик события, которое возникает, когда пользователь щелкает мышью или касается области навигации. Метод реагирует на событие, только если элемент в точке касания – аватар, иначе событие игнорируется.

Метод `onHeldstartNav` – обработчик события, которое возникает, когда пользователь начинает перетаскивание в области навигации.

```
.css('background-color','');
};
```

```
onHeldmoveNav = function ( event ){
    var drag_map = stateMap.drag_map;
    if ( ! drag_map ){ return false; }
```

```
    drag_map.top += event.px_delta_y;
    drag_map.left += event.px_delta_x;
```

```
    stateMap.$drag_target.css({
        top : drag_map.top, left : drag_map.left
    });
};
```

```
onHeldendNav = function ( event ) {
    var $drag_target = stateMap.$drag_target;
    if ( ! $drag_target ){ return false; }
```

```
    $drag_target
        .removeClass('spa-x-is-drag')
        .css('background-color',stateMap.drag_bg_color);
```

```
    stateMap.drag_bg_color = undefined;
    stateMap.$drag_target = null;
    stateMap.drag_map = null;
    updateAvatar( $drag_target );
};
```

```
onSetchatee = function ( event, arg_map ) {
```

```
    var
        $nav      = $(this),
        new_chatee = arg_map.new_chatee,
        old_chatee = arg_map.old_chatee;
```

```
// Это для выделения аватара пользователя
// в области навигации.
// См. new_chatee.name, old_chatee.name и т. д.
```

```
// здесь снимаем выделение с аватара old_chatee
if ( old_chatee ){
    $nav
        .find( '.spa-avtr-box[data-id=' + old_chatee.cid + ']' )
        .removeClass( 'spa-x-is-chatee' );
}
}
```

```
// выделяем аватар new_chatee
if ( new_chatee ){
    $nav
```

Метод onHeldmoveNav – обработчик события, которое возникает, когда пользователь перетаскивает аватар. Вызывается часто, поэтому вычисления должны быть сведены к минимуму.

Метод onHeldendNav – обработчик события, которое возникает, когда пользователь отпускает аватар после перетаскивания. Метод восстанавливает исходный цвет аватара, после чего вызывает метод updateAvatar, который читает сведения об аватаре и вызывает метод model.chat.update_avatar(<update_map>).

Метод onSetchatee – обработчик события spa-setchatee, публикуемого моделью. Здесь устанавливается зеленый цвет рамки вокруг аватара.

```

        .find( '.spa-avtr-box[data-id=' + new_chatee.cid + ']' )
        .addClass( 'spa-x-is-chatee' );
    }
};

onlistchange = function ( event ) {
    var
        $nav      = $(this),
        people_db = configMap.people_model.get_db(),
        user       = configMap.people_model.get_user(),
        chatee     = configMap.chat_model.get_chatee() || {},
        $box;

    $nav.empty();
    // если пользователь не аутентифицирован, не отрисовывать
    if ( user.get_is_anon() ){ return false; }

    people_db().each( function ( person, idx ){
        var class_list;
        if ( person.get_is_anon() ){ return true; }
        class_list = [ 'spa-avtr-box' ];

        if ( person.id === chatee.id ){
            class_list.push( 'spa-x-is-chatee' );
        }
        if ( person.get_is_user() ){
            class_list.push( 'spa-x-is-user' );
        }

        $box = $('<div/>')
            .addClass( class_list.join(' ') )
            .css( person.css_map )
            .attr( 'data-id', String( person.id ) )
            .prop( 'title', spa.util_b.encodeHtml( person.name ) )
            .text( person.name )
            .appendTo( $nav );
    });
};

onLogout = function () {
    jqueryMap.$container.empty();
};
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----

// Начало открытого метода /configModule/
// Пример: spa.avtr.configModule({...});
// Назначение: сконфигурировать модуль до инициализации, задав
// значения, которые не будут изменяться во время сеанса.
// Действие:
// Внутренняя структура, в которой хранятся конфигурационные параметры

```

Метод onListchange – обработчик события spa-listchange, публикуемого моделью. Здесь перерисовываются аватары.

Метод onLogout – обработчик события spa-logout, публикуемого моделью. Здесь удаляются все аватары.

```

// (configMap). Обновляется в соответствии с переданными аргументами.
// Больше никаких действий не предпринимается.
// Возвращает: ничего
// Исключения: объект ошибки JavaScript и трассировка стека в случае
//              недопустимых или недостающих аргументов
//
configModule = function ( input_map ) {
    spa.util.setConfigMap({
        input_map      : input_map,
        settable_map   : configMap.settable_map,
        config_map     : configMap
    });
    return true;
};
// Конец открытого метода /configModule/

// Начало открытого метода /initModule/
// Пример: spa.avtr.initModule( $container );
// Назначение: требует, чтобы модуль начал предоставлять
//              свою функциональность
// Аргументы: $container – какой контейнер использовать
// Действие: предоставляет пользователям чата интерфейс для
//           работы с аватарами
// Возвращает: ничего
// Исключения: нет
//
initModule = function ( $container ) {
    setJqueryMap( $container );

    // подписаться на глобальные события модели ← Сначала подписываемся на события,
    $.gevent.subscribe( $container, 'spa-setchatee', onSetchatee );      публикуемые моделью.
    $.gevent.subscribe( $container, 'spa-listchange', onListchange );
    $.gevent.subscribe( $container, 'spa-logout', onLogout );

    // привязать обработчики действий пользователя ← Затем привязываем обработчики со-
    $container                                          бытий браузера. Если бы привязка
        .bind( 'utap', onTapNav )
        .bind( 'uheldstart', onHeldstartNav )
        .bind( 'uheldmove', onHeldmoveNav )
        .bind( 'uheldend', onHeldendNav );

    return true;
};
// Конец открытого метода/initModule/

// возвращаем открытые методы
return {
    configModule : configModule,
    initModule   : initModule
};
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
}();

```

После завершения работы над JavaScript-частью модуля можно приступить к созданию таблицы стилей.

6.5.2. Создание таблицы стилей для модуля Avatar

Модуль Avatar рисует прямоугольники, которые графически представляют пользователей. Мы можем определить единственный класс (**spa-avtr-box**) для стилизации прямоугольников, а затем добавить модификации для подсветки пользователя (**spa-x-is-user**), подсветки собеседника (**spa-x-is-chatee**) и подсветки прямоугольника в процессе перетаскивания (**spa-x-is-drag**). Все эти селекторы показаны в листинге 6.16:

Листинг 6.16 ❖ Таблица стилей для модуля Avatar – `spa/css/spa.avtr.css`

```
/*
 * spa.avtr.css
 * Стили функционального модуля Avatar
 */

.spa-avtr-box { ← Создаем класс для стилизации аватаров.
    position      : absolute;
    width         : 62px;
    padding       : 0 4px;
    height        : 40px;
    line-height   : 32px;
    border        : 4px solid #aaa;
    cursor        : pointer;
    text-align     : left;
    overflow      : hidden;
    text-overflow : ellipsis; ← Добавляем правило text-overflow: ellipsis, чтобы
    border-radius  : 4px;      элегантно обрезать длинный текст. Чтобы это работало, не-
    text-align     : center;   обходимо также включить правило overflow: hidden.
}

.spa-avtr-box.spa-x-is-user { ← Создаем производный селектор для стилизации аватара,
    border-color   : #44f;      представляющего пользователя.
}

.spa-avtr-box.spa-x-is-chatee { ← Создаем производный селектор для стилизации аватара,
    border-color   : #080;      представляющего собеседника.
}

.spa-avtr-box.spa-x-is-drag { ← Создаем производный селектор для стилизации
    cursor         : move;      перетаскиваемого аватара.
    color          : #fff;
    background-color : #000;
    border-color    : #800;
}
```

С файлами модуля все. Осталось еще модифицировать Shell и головной HTML-документ.

6.5.3. Модификация модуля Shell и головного HTML-документа

Чтобы воспользоваться только что созданным функциональным модулем, мы должны сконфигурировать и инициализировать его в модуле Shell, как показано в листинге 6.17.

Листинг 6.17 ❖ Конфигурирование и инициализация Avatar в модуле Shell –spa/js/spa.shell.js

```
...
initModule = function ( $container ) {
    ...
    // сконфигурировать и инициализировать функциональные модули
    spa.chat.configModule({
        set_chat_anchor : setChatAnchor,
        chat_model      : spa.model.chat,
        people_model    : spa.model.people
    });
    spa.chat.initModule( jqueryMap.$container );

    spa.avtr.configModule({ ←————— Сначала конфигурируем функциональный модуль...
        chat_model      : spa.model.chat,
        people_model    : spa.model.people
    });
    spa.avtr.initModule( jqueryMap.$nav ); ←———— ... затем инициализируем его.

    // Обработать события изменения якоря в URI.
    ...
};
...
```

И последний шаг создания функционального модуля – включить файлы, содержащие JavaScript-код и таблицу стилей, в головной HTML-документ. Мы уже сделали это в главе 5, но для полноты картины приведем документ еще раз, выделив изменения **полужирным** шрифтом.

Листинг 6.18 ❖ Включение поддержки аватаров в головной HTML-документ – spa/spa.html

```
...
<!-- наши таблицы стилей -->
<link rel="stylesheet" href="css/spa.css" type="text/css"/>
<link rel="stylesheet" href="css/spa.shell.css" type="text/css"/>
<link rel="stylesheet" href="css/spa.chat.css" type="text/css"/>
<link rel="stylesheet" href="css/spa.avtr.css" type="text/css"/>
...
<!-- наш javascript-код -->
...
```

```
<script src="js/spa.shell.js" ></script>
<script src="js/spa.chat.js" ></script>
<script src="js/spa.avtr.js" ></script>
...
```

Создание модуля Avatar и интеграция его с приложением закончены. Теперь займемся тестированием.

6.5.4. Тестирование функционального модуля Avatar

Загрузив файл `spa/spa.html` в браузер, мы должны увидеть страницу, в правом верхнем углу которой находится надпись «Please sign in». Щелкнув по ней, мы, как и раньше, сможем аутентифицироваться. После открытия окна чата мы увидим интерфейс, показанный на рис. 6.9.

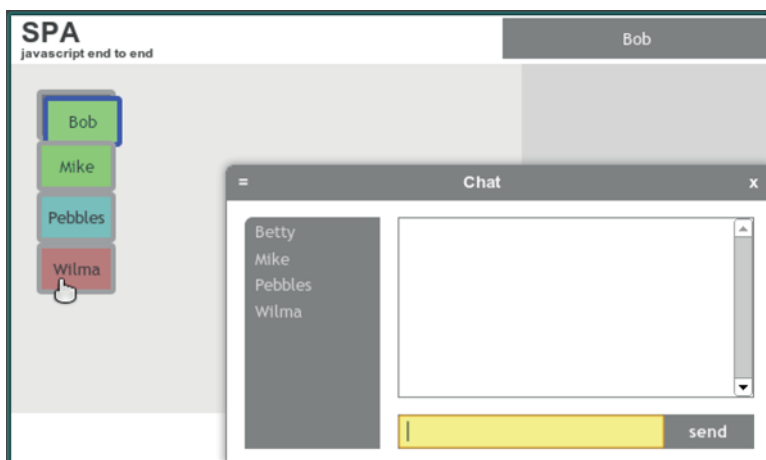


Рис. 6.9 ❖ Аватары сразу после аутентификации

Теперь можно потаскать аватары по экрану (сначала они все находятся в левом верхнем углу), буксируя их пальцем или мышью с нажатой кнопкой. В момент касания аватар должен изменить цвет. Раставив аватары в разные стороны, мы увидим картину, изображенную на рис. 6.10. У аватара пользователя рамка синяя, у аватара его собеседника – зеленая, а в процессе перетаскивания любой аватар раскрашен в черно-бело-красные цвета.

Мы реализовали все, что собирались сделать в начале этой главы. Теперь поговорим о том, как мы организовали модный сейчас аспект – привязку к данным.

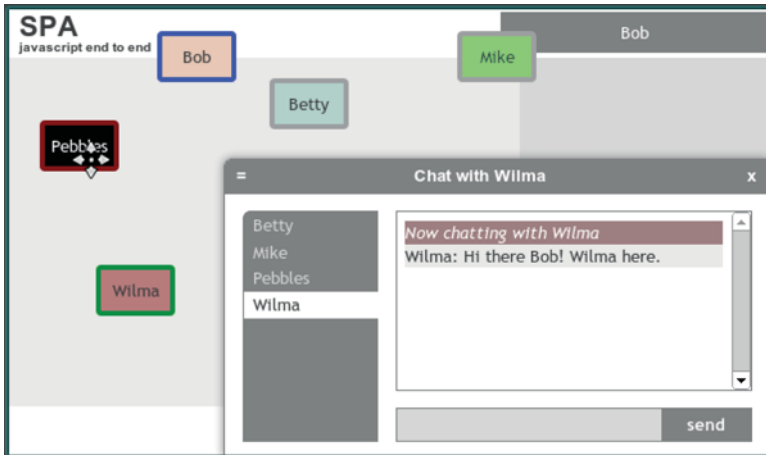


Рис. 6.10 ❖ Аватары после растаскивания

6.6. Привязка к данным и jQuery

Привязка к данным — это механизм, который гарантирует, что любое изменение в модели данных тут же отразится в интерфейсе. И наоборот, когда пользователь что-то изменяет в интерфейсе, модель данных соответственно обновляется. Ничего нового здесь нет — если вам доводилось работать над пользовательскими интерфейсами, то привязку к данным вы воспринимали как нечто само собой разумеющееся.

В этой главе мы реализовали привязку к данным с помощью методов, предоставляемых jQuery. Когда в нашем SPA изменяется модель данных, мы публикуем глобальные пользовательские события jQuery. Коллекции jQuery подписываются на интересующие их события, в результате чего в момент возникновения события вызывается функция, которая обновляет представление на экране. А когда пользователь модифицирует данные на экране, срабатывают обработчики событий, которые вызывают методы обновления модели. Все просто и позволяет достаточно гибко определять, как и когда обновлять данные и представление. Механизм привязки к данным в jQuery не скрыт под покровом тайны — и это хорошо.

Бойтесь «каркасов» SPA, дары приносящих

Некоторые библиотеки-каркасы для построения SPA сулят «автоматическую двустороннюю привязку к данным», что, конечно, звучит за-

манчиво. Но опыт научил нас относиться к таким обещаниям с осторожностью – несмотря на впечатляющие «демонстрашки», включенные в дистрибутив.

- Чтобы добиться тех же результатов, что и поднаторевший демонстратор, необходимо изучить язык библиотеки – ее API и терминологию. Это может потребовать значительных усилий.
- У автора библиотеки часто имеется свое представление о том, как нужно структурировать SPA. Если наше SPA не отвечает этому представлению, то подгонка его под предлагаемую схему может обойтись дорого.
- Библиотека может оказаться громоздкой, изобилуя ошибками и добавляя еще один уровень сложности, который может «сломаться».
- Механизм привязки к данным, реализованный в библиотеке, часто не удовлетворяет нашим требованиям к SPA.

Остановимся на последнем моменте подробнее. Допустим, мы хотим дать пользователю возможность редактировать строку таблицы, а по завершении принять или отвергнуть всю строку (во втором случае должны быть восстановлены предыдущие значения). После завершения редактирования всех строк мы хотели бы, чтобы пользователь подтвердил или отменил всю операцию целиком. И только в случае подтверждения мы готовы сохранить таблицу на сервере.

Шансы на то, что библиотека изначально поддерживает такое – вполне разумное – взаимодействие, малы. Поэтому нам придется переопределить какие-то методы, чтобы обойти поведение по умолчанию. Если мы вынуждены будем делать это неоднократно, то получится *больше* кода, *больше* уровней, *больше* файлов и *больше* сложности, чем если бы мы написали эту чертову штуку самостоятельно.

Предприняв – из самых лучших побуждений – несколько попыток, мы пришли к выводу, что приближаться к библиотекам-каркасам следует с опаской. Мы обнаружили, что они способны увеличить сложность SPA, не сделав разработку ни качественнее, ни быстрее, ни понятнее. Это вовсе не означает, что каркасами не следует пользоваться вовсе, – у них есть своя ниша. Но наше демонстрационное SPA (и немало промышленных тоже) отлично работает, используя только jQuery, несколько подключаемых модулей и немногие специализированные инструменты типа TaffyDb. Зачастую чем проще, тем лучше.

Теперь давайте завершим клиентскую часть SPA, добавив модуль Data и внося несколько косметических улучшений.

6.7. Разработка модуля Data

В этом разделе мы разработаем модуль Data, показанный на рис. 6.11.

Это будет подготовкой к использованию на стороне клиента «настоящих» данных и сервисов, поставляемых сервером, а не нашим модулем Fake. По окончании этого раздела приложение *не будет* работать, потому что необходимая серверная часть еще не готова. Ей мы займемся в главах 7 и 8.

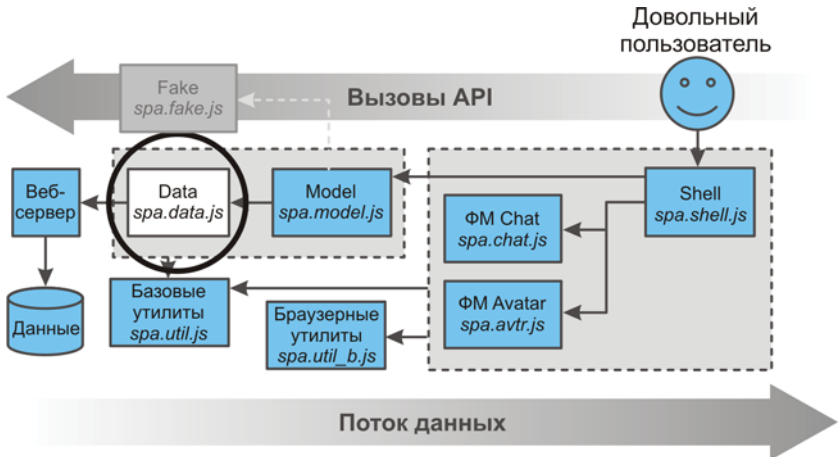


Рис. 6.11 ❖ Место модуля Data в нашей архитектуре SPA

Нам нужно добавить библиотеку Socket.IO, поскольку именно она реализует механизм транспорта сообщений. Изменение выделено **полужирным** шрифтом в листинге 6.19.

Листинг 6.19 ❖ Включение библиотеки Socket.IO в головной HTML-документ – spa/spa.html

```
...
<!-- сторонний javascript-код -->
<script src="socket.io/socket.io.js" ></script>
<script src="js/jq/taffydb-2.6.2.js" ></script>
...
```

Модуль Data должен быть инициализирован раньше, чем Model и Shell; это показано в листинге 6.20.

Листинг 6.20 ❖ Инициализация модуля Data в корневом модуле – spa/js/spa.js

```
...
var spa = (function () {
  'use strict';
  var initModule = function ( $container ) {
    spa.data.initModule(); ← Модуль Data должен быть инициализирован раньше,
    spa.model.initModule();   чем Model и Shell
    spa.shell.initModule( $container );
  };

  return { initModule: initModule };
})();
```

Далее мы обновим модуль Data, как показано в листинге 6.21. В нашей архитектуре этот модуль управляет *всеми* соединениями с сервером, и *все* данные, циркулирующие между клиентом и сервером, проходят через него. Возможно, прямо сейчас вы не до конца поймете все, что делает этот модуль, но беспокоиться нет причин – в следующей главе мы в подробностях рассмотрим библиотеку Socket.IO. Изменения выделены **полужирным** шрифтом.

Листинг 6.21 ❖ Модификация модуля Data – spa/js/spa.data.js

```
...
/*global $, io, spa */
spa.data = (function () {
  'use strict';
  var
    stateMap = { sio : null },
    makeSio, getSio, initModule;

  makeSio = function () {
    var socket = io.connect( '/chat' );
```

Создаем соединение через сокет, указывая пространство имен /chat.

```

    return {
      emit : function ( event_name, data ) {
        socket.emit( event_name, data );
      },
      on : function ( event_name, callback ) {
        socket.on( event_name, function () {
          callback( arguments );
        });
      }
    };
  };

  getSio = function () {
    if ( ! stateMap.sio ) { stateMap.sio = makeSio(); }
    return stateMap.sio;
  };

  initModule = function () {
    // Метод initModule пока ничего не делает, но мы все равно хотим, чтобы он присутствовал и чтобы корневой модуль (spa/js/spa.js) вызывал его до инициализации модулей Model и Shell.
  };

  return {
    getSio : getSio,
    initModule : initModule
  };
})();
```

Возвращаем открытые методы объекта sio.

Метод emit посылает серверу данные, ассоциированные с указанным именем события.

Метод on регистрирует обратный вызов для обработки события с указанным именем. Обработчику будут переданы все данные, полученные от сервера в составе события.

Метод getSio пытается вернуть корректный объект sio.

Экспортируем все открытые методы.

Последний шаг в рамках подготовки к работе с данными о сервере – прекратить использование фиктивных данных в модели. Соответствующее изменение показано в листинге 6.22.

Листинг 6.22 ❖ Модификация модели для работы с «настоящими» данными – spa/js/spa.model.js

```

...
spa.model = (function () {
  'use strict';
  var
    configMap = { anon_id : 'a0' },
    stateMap = {
      ...
    },
    isFakeData = false,
  ...

```

Теперь, загрузив файл `spa/spa.html` в браузер, мы обнаружим, что наше SPA перестало работать, а на консоли появились сообщения об ошибках. Если мы захотим продолжить разработку без сервера, достаточно будет «повернуть выключатель», то есть вновь присвоить переменной `isFakeData` значение `true`¹. Теперь мы готовы добавить в SPA серверную часть.

6.8. Резюме

В этой главе мы завершили работу над моделью. Мы методично спроектировали, специфицировали, разработали и протестировали объект `chat`. Для ускорения разработки мы, как и в главе 5, использовали подставные данные из модуля `Fake`. Затем мы модифицировали функциональный модуль `Chat`, воспользовавшись API объектов модели `chat` и `people`. Мы также написали функциональный модуль `Avatar`, в котором используются те же самые API. Мы обсудили вопрос о привязке к данным средствами `jQuery`. Наконец, мы добавили модуль `Data`, который будет обмениваться данными с сервером `Node.js` с помощью библиотеки `Socket.IO`. В главе 8 мы настроим сервер на работу с модулем `Data`. А в следующей главе познакомимся с `Node.js`.

¹ Браузер может сообщить, что не находит библиотеку `Socket.IO`, но это не страшно.

Сервер SPA

Когда пользователь бродит по традиционному веб-сайту, сервер тратит массу ресурсов на генерацию полных страниц и отправку их браузеру – одну за другой. Сервер SPA работает совершенно по-другому. Большая часть бизнес-логики, а также HTML-шаблоны и логика представления перемещены на сторону клиента. Сервер продолжает играть важную роль, но, очистившись от всего лишнего, он концентрируется на предоставлении таких сервисов, как постоянное хранение данных, проверка данных (валидация), аутентификация пользователей и синхронизация данных.

Исторически сложилось, что веб-разработчик вынужден был тратить изрядное время на преобразование данных из одного формата в другой – работа, напоминающая перебрасывание мусора лопатой из одной гигантской кучи в другую, и почти столь же продуктивная. Веб-разработчик должен был освоить многочисленные языки и инструментальные средства. Типичной была ситуация, когда для создания традиционного сайта требовалось отлично разбираться в SQL, Apache2, mod_rewrite, mod_perl2, Perl, DBI, HTML, CSS и JavaScript. Изучение всех этих языков обходится дорого, а необходимость постоянного переключения с одного на другой раздражает. Хуже того, если вдруг понадобится перенести какую-то логику из одной части приложения в другую, – придется переписывать ее на другом языке. В части III этой книги мы рассмотрим следующие темы:

- основы Node.js и MongoDB;
- как перестать транжирить ресурсы сервера на преобразование данных и начать использовать формат JSON во всех компонентах SPA;
- как, используя один-единственный язык JavaScript, построить серверное HTTP-приложение и организовать взаимодействие с базой данных;
- проблемы, возникающие при развертывании SPA, и их решение.

Во всех компонентах нашего стека мы используем JSON и JavaScript. Это устраняет накладные расходы на преобразование форматов данных. И существенно уменьшает количество языков и сред разработки, которыми необходимо овладеть. А на выходе получается продукт лучшего качества, который значительно дешевле обходится в разработке, распространении и сопровождении.

Глава 7

Веб-сервер

В этой главе:

- ✧ Роль веб-сервера в поддержке SPA.
- ✧ Node.js – программирование веб-сервера на JavaScript.
- ✧ ПО промежуточного уровня Connect.
- ✧ Каркас Express.
- ✧ Настройка Express для поддержки архитектуры SPA.
- ✧ Маршрутизация и CRUD.
- ✧ Обмен сообщениями с помощью Socket.IO и почему нам это интересно.

В этой главе мы обсудим, что должен делать сервер для поддержки SPA. Здесь же дается неплохое введение в Node.js. Если, прочитав эту главу, вы ощутите непреодолимое желание написать промышленное приложение с применением Node.js, мы рекомендуем книгу «Node.js in Action» (издательство Manning, 2013).

7.1. Роль сервера

SPA перемещает значительную часть традиционных функций веб-сервера на сторону клиента – браузера. Но все равно необходимо, чтобы «инь» сервера и «ян» клиента соответствовали друг другу. В некоторых вопросах – например, в обеспечении безопасности – без веб-сервера просто не обойтись, и существуют задачи, с которыми сервер справляется лучше клиента. Наиболее типичные зоны ответственности веб-сервера при построении SPA: *аутентификация и авторизация*, а также *валидация, синхронизация и хранение данных*.

7.1.1. Аутентификация и авторизация

Задача *аутентификации* – убедиться, что некто действительно является тем, за кого себя выдает. Для этого необходим сервер, потому что мы ни в коем случае не должны полагаться исключительно на данные, предоставляемые клиентом. Если бы аутентификация производилась целиком на стороне клиента, то «злойбный хакер» мог бы разобраться в нашем механизме аутентификации, подделать регистрационные

данные пользователя и украсть его учетную запись. Процесс аутентификации часто начинается с ввода имени и пароля.

Но все чаще разработчики прибегают к сторонним службам аутентификации, которые предоставляют, например, Facebook и Yahoo. В таком случае пользователь должен предоставить регистрационные данные (как правило, имя и пароль), заведенные в сторонней службе. Если, например, мы пользуемся аутентификацией через Facebook, то пользователь должен будет ввести имя и пароль своей учетной записи в Facebook, которые передаются серверу Facebook. Затем сторонний сервер свяжется с нашим и подтвердит подлинность пользователя. Для пользователя преимущество такой схемы состоит в том, что ему достаточно запомнить только одну пару имя–пароль. Для разработчика – в том, что большую часть скучных деталей реализации он перепоручает другой стороне, а заодно получает возможность предложить свой продукт всем пользователям, имеющим на «другой стороне» учетную запись.

Цель *авторизации* – гарантировать, что доступ к данным получают только те люди и системы, которые имеют на это право. Реализовать это можно, связав с каждым пользователем набор разрешений, так чтобы после аутентификации можно было узнать, что пользователю разрешено видеть. Авторизацию необходимо производить на стороне сервера, чтобы клиенту никогда не посылались данные, которые он видеть не должен. В противном случае наш «злойбный хакер» сможет разобраться в коде приложения и получить доступ к секретной информации, не предназначенной для его глаз. Побочный эффект авторизации состоит в том, что раз сервер посылает клиенту только разрешенные данные, то общий объем передаваемых по сети данных сокращается, что в принципе может заметно ускорить время выполнения операции.

7.1.2. Валидация

Валидация – это процедура контроля качества, задача которой – не допустить сохранения неточных или неправдоподобных данных. Валидация помогает предотвратить сохранение и распространение ошибочных данных в другие системы и доставку их пользователям. Например, авиакомпания может проверить, что при заказе билета на самолет указана дата в будущем, на которую еще есть свободные места. Не будь такой проверки, компания продавала бы больше билетов, чем есть мест, билеты на несуществующие места или на рейсы, которые уже давно улетели. Важно производить валидацию как на стороне

клиента, так и на сервере: клиентская валидация ускоряет получение ответа, но и без серверной не обойтись, потому что никаким данным, поступившим от клиента, доверять нельзя. Если сервер примет некорректные данные, то возможны самые разные проблемы, например:

- из-за ошибки в программе валидация на клиентской стороне SPA может вообще оказаться пропущенной или быть произведена неправильно;
- на некоторых клиентах валидация может не проводиться, а у одного и того же серверного приложения могут быть самые разные клиенты;
- данные, которые когда-то были корректными, в момент отправки перестали быть таковыми (например, уже после того как пользователь нажал кнопку «Отправить», место было зарезервировано кем-то еще);
- «злой хакер» снова влез и пытается перехватить управление нашим сайтом или нарушить его работу, подложив в наше хранилище заведомо некорректные данные.

Классический пример неправильно реализованной валидации на сервере – атаки путем внедрения SQL, которым подвергались многие именитые организации, которым надо было бы быть поосмотрительнее. Мы ведь не хотим присоединиться к их компании, правда?

7.1.3. Сохранение и синхронизация данных

Хотя SPA и может сохранять данные на стороне клиента, эти данные недолговечны, их легко изменить или удалить без ведома SPA. В большинстве случаев клиентское хранилище следует использовать лишь как временное, а постоянно хранить данные на стороне сервера.

Бывает также, что данные нужно синхронизировать между несколькими клиентами, например информацию о том, находится ли человек в онлайн, должны получать все, кто видит этого человека в списке. Простейший способ решить эту задачу – поручить клиенту отправлять информацию о состоянии сервера, который затем может разослать ее всем аутентифицированным клиентам. Иногда требуется синхронизировать транзитные данные. Например, сервер чата передает сообщения аутентифицированным клиентам; сам он эти данные не хранит, но обязан доставить сообщения тем, кому они предназначены, и никому более.

7.2. Node.js

Node.js – это платформа, на которой языком управления служит JavaScript. При использовании в качестве HTTP-сервера она идеологически аналогична Twisted, Tornado или `mod_perl`. Есть и другие популярные платформы для построения веб-серверов, которые состоят из двух компонентов: собственно HTTP-сервера и контейнера приложений. К таковым можно отнести Apache/PHP, Passenger/Ruby и Tomcat/Java.

Когда HTTP-сервер и приложение пишутся вместе, легко решить некоторые задачи, требующие больших усилий на платформах с разделением HTTP и приложения.

Например, желая записывать журналы в базу данных, организованную в памяти, мы можем не думать о том, где заканчивается HTTP-сервер и начинается сервер приложения.

7.2.1. Почему именно Node.js?

Мы выбрали в качестве серверной платформы Node.js, потому что она обладает рядом возможностей, очень удобных при построении современных SPA:

- сервер и есть приложение. А значит, нет нужды забивать голову настройкой и вопросами интерфейса с отдельным сервером приложений. Все управление сосредоточено в одном месте, в одном процессе;
- язык серверных приложений – JavaScript, и, следовательно, снижается когнитивная нагрузка из-за того, что серверное приложение пишется на одном языке, а клиентское SPA – на другом. Это также означает, что один и тот же код можно использовать на стороне клиента и сервера, что дает массу преимуществ. Например, и в SPA, и на сервере можно работать с общей библиотекой валидации данных;
- Node.js – неблокирующий, событийно-управляемый сервер. По существу, это означает, что единственный экземпляр Node.js, работающий на скромном оборудовании, способен обслуживать десятки и сотни тысяч одновременно открытых соединений – ситуация, характерная для обмена сообщениями в реальном времени. Это свойство часто бывает весьма желательно при разработке современных SPA;
- Node.js работает быстро, отлично поддерживается, а количество модулей для него (и их авторов) стремительно растет.

Node.js обрабатывает сетевые запросы иначе, чем большинство других серверных платформ. Как правило, HTTP-сервер организует пул процессов или потоков, готовых обслуживать входящие запросы. А у Node.js имеется единственная очередь событий, в которую ставятся все входящие запросы. Более того, даже обработка одного запроса разбивается на части, представленные различными событиями в главной очереди. На практике это означает, что Node.js не ждет завершения долгой обработки одного события, перед тем как перейти к следующему. Если какой-то запрос к базе данных занимает много времени, то Node.js сразу приступает к обработке других событий. Когда запрос завершится, в очередь будет помещено событие, чтобы управляющая программа могла воспользоваться результатами.

Не тратя больше слов, посмотрим, как с помощью Node.js создать серверное приложение.

7.2.2. Приложение «Hello World» для Node.js

Зайдите на сайт Node.js (<http://nodejs.org/#download>), скачайте дистрибутив и установите его. Есть много способов это сделать, но если вы не на короткой ноге с командной строкой, то, наверное, самый простой – воспользоваться установщиком, имеющимся в операционной системе.

Вместе с Node.js устанавливается менеджер пакетов для него – Node Package Manager, или `npm`. Это аналог `CPAN` для Perl, `gem` для Ruby или `pip` для Python. Он загружает и устанавливает пакеты, попутно разрешая зависимости. Это гораздо проще, чем делать все вручную. Установив Node.js и `npm`, давайте построим свой первый сервер. На сайте Node.js (<http://nodejs.org>) имеется пример простого веб-сервера Node, им и воспользуемся. Создайте каталог `webapp` и зайдите в него. Затем создайте в нем файл `app.js`, поместив туда код, показанный в листинге 7.1.

Листинг 7.1 ❖ Создание простого серверного приложения для Node – `webapp/app.js`

```
/*
 * app.js - Hello World
 */
/*jslint
    node : true, continue : true,
    devel : true, indent : 2,  maxerr : 50,
    newcap : true, nomen : true, plusplus : true,
    regexp : true, sloppy : true, vars : false,
    white : true
*/
```

```

/*global */

var http, server;

http = require( 'http' );
server = http.createServer( function ( request, response ) {
    response.writeHead( 200, { 'Content-Type': 'text/plain' } );
    response.end( 'Hello World' );
}).listen( 3000 );

console.log( 'Прослушивается порт %d', server.address().port );

```

Откройте терминал, перейдите в каталог, где сохранили файл `app.js`, и запустите сервер командой

```
node app.js
```

Должно появиться сообщение Прослушивается порт 3000. Открыв браузер (на том же самом компьютере) и введя адрес `http://localhost:3000`, мы увидим строку `Hello World`. Класс, действительно просто! Весь сервер занял всего семь строк кода. Не знаю, как вы, а я был в восторге, когда сумел написать и запустить серверное веб-приложение за несколько минут. А теперь поговорим о том, что этот код означает.

В начале находится стандартный заголовок с параметрами JSLint. Это позволяет проверять серверный JavaScript-код точно так же, как клиентский:

```

/*
 * app.js - Hello World
 */
/*jslint      node    : true, continue : true,
   devel      : true, indent  : 2,   maxerr   : 50,
   newcap     : true, nomen   : true, plusplus : true,
   regexp     : true, sloppy  : true, vars     : false,
   white      : true
 */
/*global */

```

В следующей строке объявляются переменные в области видимости модуля:

```
var http, server;
```

Следующая строка говорит Node.js, что в это серверное приложение нужно включить модуль `http`. Это аналог HTML-тега `script`, с помощью которого мы включаем JavaScript-файлы для браузера. Модуль `http` является частью ядра Node.js и служит для создания HTTP-сервера. Мы сохраняем ссылку на этот модуль в переменной `http`:

```
http = require( 'http' );
```

Далее мы вызываем метод `http.createServer`, который создает HTTP-сервер. Мы передаем ему анонимную функцию, которая будет вызываться всякий раз, как сервер Node.js получит событие запроса. Этой функции в качестве аргументов передаются два объекта: `request` и `response`.

Объект `request` представляет отправленный клиентом HTTP-запрос:

```
server = http.createServer( function ( request, response ) {
```

Внутри анонимной функции мы начинаем формировать ответ на HTTP-запрос. В следующей строке с помощью объекта `response` создается HTTP-заголовок. Мы говорим, что код ответа равен 200, то есть успешное завершение, и предоставляем анонимный объект, у которого есть свойство `Content-Type` со значением `text/plain`. Это служит браузеру указанием на тип содержимого в ответе:

```
response.writeHead( 200, { 'Content-Type': 'text/plain' } );
```

В следующей строке метод `response.end` отправляет строку `'Hello World'` клиенту и уведомляет Node.js о том, что работа с ответом завершена:

```
response.end( 'Hello World' );
```

Затем мы закрываем анонимную функцию и вызов метода `createServer`. После этого в программе следует вызов метода `listen` объекта `http`. Этот метод говорит объекту `http`, что тот должен прослушивать порт 3000:

```
}).listen( 3000 );
```

Напоследок мы выводим на консоль сообщение о том, что серверное приложение запущено. Номер прослушиваемого порта мы получаем из атрибута объекта `server`:

```
console.log( 'Прослушивается порт %d', server.address().port );
```

В данном случае мы с помощью Node.js создали очень простой сервер. Имеет смысл потратить некоторое время на более близкое знакомство с объектами `request` и `response`, которые передаются анонимной функции в методе `http.createServer`. Начнем с распечатки содержимого объекта `request`. Новая строка выделена в листинге 7.2 **полужирным шрифтом**.

Листинг 7.2 ❖ Добавление простого протоколирования в серверное приложение для Node – webapp/app.js

```

/*
 * app.js – простое протоколирование
 */
...

var http, server;

http = require( 'http' );
server = http.createServer( function ( request, response ) {
  console.log( request );
  response.writeHead( 200, { 'Content-Type': 'text/plain' } );
  response.end( 'Hello World' );
}).listen( 3000 );

console.log( 'Прослушивается порт %d', server.address().port );

```

Перезапустив веб-приложение, мы увидим распечатку содержимого объекта (листинг 7.3) в том же окне терминала, в котором работает приложение. Пока не стоит ломать голову над структурой этого объекта; в свое время мы расскажем о том, что необходимо знать.

Листинг 7.3 ❖ Объект request

```

{ output: [],
  outputEncodings: [],
  writable: true,
  _last: false,
  chunkedEncoding: false,
  shouldKeepAlive: true,
  useChunkedEncodingByDefault: true,
  sendDate: true,
  _hasBody: true,
  _trailer: '',
  finished: false,
  ... // и еще примерно 100 строк кода

```

Отметим несколько особо интересных свойств объекта `request`.

- `ondata` – метод, вызываемый, когда сервер начинает принимать данные от клиента, например при установке переменных `POST`. Этот способ получения данных от клиента принципиально отличается от применяемого в большинстве систем. Мы абстрагируемся от деталей и получаем весь список параметров из некоторой переменной.
- `headers` – все заголовки, присутствующие в запросе.

- `url` – адрес запрошенной страницы, не включающий адреса узла. Например, для `http://www.singlepagewebapp.com/test` `url` будет равно `/test`.
- `method` – метод отправки запроса: GET или POST.

Вооружившись знаниями об этих атрибутах, мы можем приступить к написанию рудиментарного маршрутизатора. Изменения в листинге 7.4 выделены **полужирным** шрифтом.

Листинг 7.4 ❖ Добавление простого маршрутизатора в серверное приложение для Node – `webapp/app.js`

```
/*
 * app.js - простая маршрутизация
 */
...

var http, server;

http = require( 'http' );
server = http.createServer( function ( request, response ) {
    var response_text = request.url === '/test' ← Проверям URL запрошенной страницы.
    ? 'вы запросили страницу test'
    : 'Hello World';
    response.writeHead( 200, { 'Content-Type': 'text/plain' } );
    response.end( response_text );
}).listen( 3000 );

console.log( 'Прослушивается порт %d', server.address().port );
```

Действуя таким образом, мы могли бы написать свой собственный маршрутизатор, и для простых приложений такой подход был бы даже оправдан. Но мы ожидаем от нашего серверного приложения гораздо большего, поэтому воспользуемся каркасом, который разработало и протестировало сообщество Node.js. Первым каркасом, который мы рассмотрим, будет Connect.

7.2.3. Установка и использование Connect

Connect – это расширяемый каркас *промежуточного уровня*, который наделяет веб-сервер Node.js такими средствами, как простая аутентификация, управление сессиями, обслуживание статических файлов и обработка форм. Это не единственный каркас подобного рода, зато простой и относительно стандартный. Connect позволяет вставлять функциональность *промежуточного уровня* между получением запроса и отправкой окончательного ответа. В общем виде функция промежуточного уровня принимает входящий запрос, производит

над ним какие-то действия, а затем передает результат следующей функции промежуточного уровня или завершает формирование ответа, вызывая метод `response.end`.

Познакомиться с Connect и паттерном промежуточного уровня проще всего на практике. Войдите в каталог `webapp` и установите Connect, для чего введите следующую команду:

```
npm install connect
```

В результате появится каталог `node_modules`, в который будет установлен каркас Connect. В каталоге `node_modules` находятся все модули, составляющие приложение Node.js. В него по умолчанию устанавливаются модули `npm`, и наши собственные модули должны располагаться там же. Модифицируем наше серверное приложение, как показано в листинге 7.5. Изменения выделены **полужирным** шрифтом.

Листинг 7.5 ❖ Модификация серверного приложения для Node с целью использования Connect – `webapp/app.js`

```
/*
 * app.js - простой сервер на основе Connect
 */
...
var
  connectHello, server,
  http    = require( 'http' ),
  connect = require( 'connect' ),
  app     = connect(),
  bodyText = 'Hello Connect';

connectHello = function ( request, response, next ) {
  response.setHeader( 'content-length', bodyText.length );
  response.end( bodyText );
};

app.use( connectHello );
server = http.createServer( app );

server.listen( 3000 );
console.log( 'Прослушивается порт %d', server.address().port );
```

Этот сервер на основе Connect устроен так же, как наш первый сервер из предыдущего раздела. Мы определяем функцию `connectHello`, а затем сообщаем объекту Connect `app`, что это будет единственная функция промежуточного уровня. Поскольку `connectHello` вызывает метод `response.end`, то она завершает формирование ответа сервера. Отправляясь от этого примера, расширим промежуточный уровень.

7.2.4. Добавление промежуточного уровня Connect

Предположим, что мы хотим протоколировать все обращения к странице. Для этого воспользуемся встроенной в Connect функцией промежуточного уровня `connect.logger()`. В листинге 7.6 показано, как добавить ее в наш сервер.

Листинг 7.6 ❖ Добавление протоколирования в серверное приложение для Node на основе Connect – `webapp/app.js`

```
/*
 * app.js - простой сервер на основе Connect с протоколированием
 */
...
var
  connectHello, server,
  http    = require( 'http' ),
  connect = require( 'connect' ),

  app      = connect(),
  bodyText = 'Hello Connect';

connectHello = function ( request, response, next ) {
  response.setHeader( 'content-length', bodyText.length );
  response.end( bodyText );
};

app
  .use( connect.logger() )
  .use( connectHello );
server = http.createServer( app );

server.listen( 3000 );
console.log( 'Прослушивается порт %d', server.address().port );
```

Мы добавили лишь вызов `connect.logger()` перед нашей функцией промежуточного уровня `connectHello`. Теперь всякий раз, как клиент отправляет HTTP-запрос нашему приложению, первой из функций промежуточного уровня вызывается `connect.logger()`, которая выводит информацию на консоль. А *следующей* будет наша собственная функция промежуточного уровня, которая, как и раньше, отправляет клиенту строку `Hello Connect` и завершает формирование ответа. Если мы перейдем в браузере по адресу `http://localhost:3000`, то на консоли увидим примерно такую картину:

```
Listening on port 3000
127.0.0.1 - - [Wed, 01 May 2013 19:27:12 GMT] "GET / HTTP/1.1" 200 \
13 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.31 \
(KHTML, like Gecko) Chrome/26.0.1410.63 Safari/537.31"
```

Но хотя Connect находится на более высоком уровне абстракции, чем Node.js, нам хотелось бы еще большего. Время познакомиться с Express.

7.2.5. Установка и использование Express

Express – это нетребовательный к ресурсам веб-каркас, построенный по образцу Sinatra, облегченного веб-каркаса для Ruby. В SPA нам не понадобится вся функциональность, которую предлагает Express, однако же его возможности шире, чем у Connect, – на самом деле он построен поверх Connect.

Проверьте, что текущим каталогом является webapp, и установите Express. Но вместо прямого указания модуля в командной строке, как в случае Connect, мы воспользуемся файлом манифеста, который называется `package.json`, чтобы сообщить npm, какие версии каких модулей необходимы нашему приложению для корректной работы. Это удобно, когда мы хотим установить приложение на удаленный сервер или когда кто-то другой скачивает и устанавливает наше приложение на свою машину. В листинге 7.7 показан файл `package.json` для установки Express.

Листинг 7.7 ❖ Создание манифеста для установки с помощью npm – `webapp/package.json`

```
{
  "name"    : "SPA",
  "version" : "0.0.3",
  "private" : true,
  "dependencies" : {
    "express" : "3.2.x"
  }
}
```

Атрибут `name` задает имя приложения, оно может быть произвольным. Атрибут `version` содержит номер версии приложения и должен иметь вид `<major>.<minor>.<patch>`, где `major` – основной номер, `minor` – дополнительный номер, а `patch` – номер последнего исправления. Если атрибут `private` равен `true`, то npm не будет публиковать приложение. Наконец, в атрибуте `dependencies` описываются модули вместе с номерами версий, которые npm должен установить. В данном случае нам нужен только модуль `express`. Давайте сначала удалим существующий каталог `webapp/node_modules`, а затем с помощью npm установим Express:

```
npm install
```

Если при добавлении нового модуля командой `npm` задать флаг `-save`, то модуль будет автоматически добавлен в файл `package.json`. Это удобно во время разработки. Обратите также внимание на указанный нами номер версии Express – “3.2.x”. Это означает, что нам нужна версия 3.2 с последними исправлениями. Рекомендуется задавать версию именно так, потому что исправления редко нарушают API, а лишь устраняют ошибки или направлены на поддержание обратной совместимости.

Теперь изменим файл `app.js`, так чтобы воспользоваться Express. Добавим немного формализма, включив прагму `'use strict'` и разграничители секций. Изменения выделены в листинге 7.8 **полужирным** шрифтом.

Листинг 7.8 ❖ Серверное приложение для Node на основе Express – `webapp/app.js`

```
/*
 * app.js - простой сервер на основе Express
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
    http    = require( 'http' ),
    express = require( 'express' ),

    app     = express(),
    server  = http.createServer( app );
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.get( '/', function ( request, response ) {
    response.send( 'Hello Express' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----

// ----- НАЧАЛО ЗАПУСКА СЕРВЕРА -----
server.listen( 3000 );
console.log(
    'Express-сервер прослушивает порт %d в режиме %s',
    server.address().port, app.settings.env
);
// ----- КОНЕЦ ЗАПУСКА СЕРВЕРА -----
```

При взгляде на этот пример не сразу понятно, почему с Express проще работать, поэтому проанализируем код. Сначала мы загружаем модули `express` и `http`:

```
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  http    = require( 'http' ),
  express = require( 'express' ),

  app     = express(),
  server  = http.createServer( app );
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
```

Затем мы создаем объект **app**, вызывая функцию **express**. У этого объекта имеются методы для установки маршрутов и других свойств приложения. Мы также создаем объект **server**, который представляет HTTP-сервер и понадобится нам позже:

```
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  http    = require( 'http' ),
  express = require( 'express' ),

  app     = express(),
  server  = http.createServer( app );
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
```

Далее мы определяем маршруты для приложения с помощью метода **app.get**:

```
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.get( '/', function ( request, response ) {
  response.send( 'Hello Express' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
```

Благодаря богатому набору методов типа **get** Express упрощает маршрутизацию в Node.js. Первый аргумент **app.get** – это образец, с которым сопоставляется URL запроса. Например, если браузер на машине разработчика отправляет запрос на адрес **http://localhost:3000** или **http://localhost:3000/**, то методом запроса будет **GET**, а строка запроса будет содержать **'/'**, что отвечает образцу. Второй аргумент – функция обратного вызова, выполняемая при обнаружении соответствия с образцом. В качестве аргументов ей передаются объекты **request** и **response**. Параметры строки запроса можно получить из свойства **request.params**.

В третьей – и последней – секции мы запускаем сервер и выводим на консоль сообщение:

```
// ----- НАЧАЛО ЗАПУСКА СЕРВЕРА -----
server.listen( 3000 );
console.log(
  'Express-сервер прослушивает порт %d в режиме %s',
  server.address().port, app.settings.env
);
```

Теперь, имея работающее Express-приложение, добавим промежуточный уровень.

7.2.6. Добавление промежуточного уровня в Express-приложение

Поскольку Express построен поверх Connect, мы можем вызывать цепочки функций промежуточного уровня, используя уже знакомый синтаксис. В листинге 7.9 показано, как добавить в приложение функцию протоколирования запросов.

Листинг 7.9 ❖ Добавление протоколирования в приложение на основе Express – webapp/app.js

```
/*
 * app.js – простой сервер на основе Express с протоколированием
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.use( express.logger() );
app.get( '/', function ( request, response ) {
  response.send( 'Hello Express' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
```

Express предоставляет все методы Connect, имеющие отношение к промежуточному уровню, поэтому явно включать Connect нет нужды. Если запустить последнюю программу, то на консоль будет выводиться информация обо всех запросах, как при использовании метода `connect.logger` в предыдущем разделе.

Мы можем организовать все функции промежуточного уровня, воспользовавшись методом Express `app.configure`, как показано в листинге 7.10. Изменения **выделены** полужирным шрифтом.

Листинг 7.10 ❖ Применение `configure` для организации функций промежуточного уровня в приложении на основе Express – webapp/app.js

```
/*
 * app.js – сервер на основе Express с промежуточным уровнем
 */
...
```

```
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
  app.use( express.logger() );
  app.use( express.bodyParser() );
  app.use( express.methodOverride() );
});
app.get( '/', function ( request, response ) {
  response.send( 'Hello Express' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

Мы добавили в конфигурацию еще два метода промежуточного уровня: `bodyParser` и `methodOverride`. Метод `bodyParser` декодирует формы, в дальнейшем мы будем активно им пользоваться. Метод `configure` также позволяет изменять конфигурацию в зависимости от окружения Node.js, в котором работает приложение.

7.2.7. Окружения в Express

Express поддерживает идею смены конфигурации в зависимости от параметров окружения. Приведем несколько примеров окружения: `development`, `testing`, `staging`, `production`. Express определяет, какое окружение используется, читая переменную окружения `NODE_ENV`, и реагирует, соответственно настраивая конфигурацию. В системе Windows запускать серверное приложение следовало бы так:

```
SET NODE_ENV=production node app.js
```

А в Mac и Linux – так:

```
NODE_ENV=production node app.js
```

Уверены, что пользователи других систем сами разберутся, что следует делать.

Имя окружения, в котором запускается серверное Express-приложение, может быть любым. Если переменная `NODE_ENV` вообще не задана, то по умолчанию подразумевается имя `development`.

Изменим приложение, так чтобы оно подстраивалось к заданному окружению. Методы промежуточного уровня `bodyParser` и `methodOverride` мы хотим использовать в любом окружении. Но в окружении `development` мы хотим еще, чтобы приложение протоколировало HTTP-запросы и подробные сведения об ошибках. А в окружении `production` – только краткие сведения об ошибках. Изменения выделены в листинге 7.11 **полужирным** шрифтом.

Листинг 7.11 ❖ Поддержка разных окружений в Express – webapp/app.js

```
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
  app.use( express.bodyParser() );
  app.use( express.methodOverride() );
});

app.configure( 'development', function () {
  app.use( express.logger() );
  app.use( express.errorHandler({
    dumpExceptions : true,
    showStack : true
  }) );
});

app.configure( 'production', function () {
  app.use( express.errorHandler() );
});

app.get( '/', function ( request, response ) {
  response.send( 'Hello Express' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

Методы `bodyParser` и `methodOverride` присутствуют в любом окружении.

В окружении разработки добавляем методы `logger` и `errorHandler`, причем последний настраиваем так, чтобы выводились исключения и трассировка стека.

В производственном окружении добавляем метод `errorHandler` с параметрами по умолчанию.

Чтобы протестировать эти конфигурации, мы можем запустить приложение в режиме разработки (`node app.js`) и загрузить страницу в браузер. На консоли Node.js мы увидим протокол. Теперь остановим сервер и запустим его в производственном режиме (`NODE_ENV=production node app.js`). Если перезагрузить страницу в браузере, никаких сообщений на консоли не появится.

Разобравшись в общих чертах, как работают Node.js, Connect и Express, перейдем к более сложным методам маршрутизации.

7.2.8. Обслуживание статических файлов с помощью Express

Как вы, наверное, и предполагали, для обслуживания статических файлов в Express-приложении необходимо добавить дополнительные функции промежуточного уровня и перенаправление. Скопируйте содержимое каталога `sra` из главы 6 в подкаталог `public`, как показано в листинге 7.12.

Листинг 7.12 ❖ Добавление каталога `public`, содержащего статические файлы

```
webapp
+-- app.js
+-- node_modules/...
```

```
+-- package.json
`-- public # сюда копируется содержимое 'spa'
    +-- css/...
    +-- js/...
    `-- spa.html
```

Теперь можно модифицировать приложение, так чтобы оно обслуживало статические файлы. Изменения выделены в листинге 7.13 **полужирным** шрифтом.

Листинг 7.13 ❖ Обслуживание статических файлов в Express – webapp/app.js

```
/*
 * app.js - Express-сервер, обслуживающий статические файлы
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
  app.use( express.bodyParser() );
  app.use( express.methodOverride() );
  app.use( express.static( __dirname + '/public' ) );
  app.use( app.router );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...

app.configure( 'development', function () {
  app.use( express.logger() );
  app.use( express.errorHandler({
    dumpExceptions : true,
    showStack : true
  }) );
});

app.configure( 'production', function () {
  app.use( express.errorHandler() );
});

app.get( '/', function ( request, response ) {
  response.redirect( '/spa.html' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...

```

Определяем корневой каталог статических файлов как <текущий_каталог>/public.

Добавляем функцию маршрутизации после функции обслуживания статических файлов.

Перенаправляем запросы к корневому каталогу на наш головной HTML-файл /spa.html.

Теперь, если запустить приложение (`node app.js`) и перейти в браузере по адресу `http://localhost:3000`, мы увидим наше SPA в том виде, в каком оставили его в главе 6. Правда, аутентифицироваться мы пока не можем, потому что серверная часть еще не готова.

Составив представление о промежуточном уровне Express, перейдем к вопросу о маршрутизации – она понадобится нам для организации веб-служб.

7.3. Более сложная маршрутизация

До сих пор наше приложение предоставляло только маршрут к корню и возвращало текст браузеру. В этом разделе мы сделаем следующее:

- воспользуемся каркасом Express, чтобы задать CRUD-маршруты для управления объектами, представляющими пользователей;
- установим свойства ответа, в частности тип содержимого, для всех CRUD-маршрутов;
- сделаем код обобщенным, чтобы он мог работать для любого CRUD-маршрута;
- поместим логику маршрутизации в отдельный модуль.

7.3.1. CRUD-маршруты для управления пользователями

Операции *CRUD* (*Create, Read, Update, Delete*)¹ – это основные операции, которые чаще всего требуются для организации постоянного хранения данных. Если вы слышите о CRUD впервые или хотите освежить свои знания, обратитесь к Википедии, где есть подробная статья на эту тему. В веб-приложениях для реализации CRUD часто употребляется архитектурный стиль *REST* (*Representational State Transfer* – передача представимых состояний). В REST строго определена семантика глаголов *GET*, *POST*, *PUT*, *PATCH* и *DELETE*. Если вы знаете и любите REST, не колеблясь пользуйтесь им; это абсолютно допустимый способ обмена данными между распределенными системами, и в Node.js даже есть немало модулей для его поддержки.

Мы реализовали маршруты для основных CRUD-операций над нашими объектами пользователей и решили не прибегать к REST в этом примере. Тому есть несколько причин. Во-первых, во многих браузерах поддерживаются не все используемые в REST глаголы, так что *PUT*, *PATCH* и *DELETE* приходится реализовывать путем передачи дополнительных параметров или заголовка в запросе, отправляемом методом *POST*. Это означает, что разработчику не так-то просто узнать, каким методом был отправлен запрос, он вынужден копаться в заголовках. Кроме того, REST не идеально отображается на CRUD, хотя глаголы REST очень похожи на операции CRUD. Наконец, браузер может непрошено вмешиваться в обработку кодов состояния. Например, вместо того чтобы передать клиентскому SPA код состояния 302,

¹ Создать, читать, обновить, удалить. – Прим. перев.

браузер может перехватить его и попытаться сделать то, что считает «правильным», то есть переадресовать пользователя на другой ресурс. Не всегда это именно то поведение, которое нам нужно.

Начнем с построения списка всех пользователей.

Определение маршрута для получения списка пользователей

Для получения списка пользователей можно определить простой маршрут. Отметим, что свойству `contentType` объекта ответа следует присвоить значение `json`. Тогда по HTTP-заголовкам браузер поймет, что ответ содержит данные в формате JSON. Изменения выделены в листинге 7.14 **полужирным** шрифтом.

Листинг 7.14 ❖ Маршрут для получения списка пользователей – `webapp/app.js`

```
/*
 * app.js - Express-сервер с более сложными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...
// приведенная ниже конфигурация целиком относится к маршрутам
app.get( '/', function ( request, response ) {
    response.redirect( ' /spa.html ' );
});

app.get( '/user/list', function ( request, response ) {
    response.contentType( 'json' );
    response.send({ title: 'user list' });
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

В маршруте для списка пользователей ожидается, что запрос отправлен методом GET. Это нормально, коль скоро мы собираемся читать данные. В следующем маршруте мы укажем метод POST, позволяющий отправить серверу большой объем данных.

Определение маршрута для создания объекта пользователя

Определяя маршрут для создания объекта пользователя, мы должны иметь в виду, что клиент отправит данные методом POST. Express предлагает метод `app.post` для маршрутизации POST-запросов, отвечающих заданному образцу. Добавим его в наше приложение, как показано в листинге 7.15.

Листинг 7.15 ❖ Маршрут для создания объекта пользователя – `webapp/app.js`

```
/*
 * app.js - Express-сервер с более сложными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...
app.get( '/user/list', function ( request, response ) {
    response.contentType( 'json' );
    response.send({ title: 'user list' });
});

app.post( '/user/create', function ( request, response ) {
    response.contentType( 'json' );
    response.send({ title: 'user created' });
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

Мы пока ничего не делаем с присланными данными, этим мы займемся в следующей главе. Перейдя в браузере по адресу `http://localhost:3000/user/create`, мы увидим ошибку 404 и сообщение `Cannot GET /user/create`. Это объясняется тем, что браузер посылает GET-запрос, а этот маршрут предназначен только для обработки POST-запросов. Но мы можем воспользоваться для создания пользователя командной строкой:

```
curl http://localhost:3000/user/create -d {}
```

И сервер должен в ответ прислать:

```
{"title":"User created"}
```

Программы curl и wget

Те, кто работает в операционной системе Mac или Linux, могут воспользоваться для тестирования нашего API программой `curl` и вообще забыть о браузере. Для проверки только что созданного маршрута путем отправки POST-запроса на URL `user/create` нужно выполнить команду:

```
curl http://localhost:3000/user/create -d {}
{"title":"User created"}
```

Флаг `-d` означает, что нужно послать данные, а пустой литеральный объект – что данных нет. Использование `curl` вместо браузера может заметно ускорить разработку. Чтобы больше узнать о возможностях `curl`, выполните команду `curl -h`.

Аналогичные результаты можно получить с помощью программы `wget`:

```
wget http://localhost:3000/user/create --post-data='{}' -O -
```

Чтобы узнать о возможностях `wget`, выполните команду `wget -h`.

Имея маршрут для создания объекта пользователя, мы можем создать и маршрут для чтения такого объекта.

Определение маршрута для чтения объекта пользователя

Маршрут для чтения объекта пользователя похож на маршрут для его создания, только используется метод GET и в URL передается дополнительный аргумент: идентификатор пользователя. В определении маршрута указывается имя параметра, отделенное от пути двоеточием, как показано в листинге 7.16.

Листинг 7.16 ❖ Маршрут для чтения объекта пользователя – webapp/app.js

```
/*
 * app.js - Express-сервер с более сложными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...

app.post( '/user/create', function ( request, response ) {
    response.contentType( 'json' );
    response.send({ title: 'user created' });
});

app.get( '/user/read/:id', function ( request, response ) {
    response.contentType( 'json' );
    response.send({
        title: 'user with id ' + request.params.id + ' found'
    });
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

Параметр `:id`, указанный в конце маршрута, доступен через объект `request.params`. Получить идентификатор пользователя из URL-адреса, сопоставленного с маршрутом `/user/read/:id`, можно в виде значения `request.params['id']` или `request.params.id`. Если был запрошен URL `http://localhost:3000/user/read/12`, то `request.params.id` будет равно 12. Попробуйте сами, а заодно имейте в виду, что для этого маршрута конкретное значение `id` неважно – лишь бы оно было синтаксически допустимым. Несколько примеров приведено в табл. 7.1.

Таблица 7.1. Маршруты и результаты

Попробуйте задать в браузере	Печатается на консоли Node.js
/user/read/19	{"title": "User with id 19 found"}
/user/read/spa	{"title": "User with id spa found"}
/user/read/	Cannot GET /user/read/
/user/read/?	Cannot GET /user/read/?

То, что сопоставление производится с любым маршрутом, это хорошо, но что, если наш идентификатор обязательно должен быть числом? Мы не хотим, чтобы маршрутизатор производил сопоставление с путем, в котором идентификатор нечисловой. В Express имеется механизм, позволяющий принимать только маршруты, содержащие числовые идентификаторы; нужно лишь указать в определении маршрута регулярное выражение `[(0-9)]+`, как показано в листинге 7.17.

Листинг 7.17 ❖ Ограничение на маршрут: допускаются только числовые идентификаторы – `webapp/app.js`

```

/*
 * app.js - Express-сервер с более сложными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...

app.get( '/user/read:id([0-9]+)', function ( request, response ) {
  response.contentType( 'json' );
  response.send({
    title: 'user with id ' + request.params.id + ' found'
  });
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...

```

В табл. 7.2 показано, как работает маршрут, принимающий только числовые идентификаторы.

Таблица 7.2. Маршруты и результаты

Попробуйте задать в браузере	Печатается на консоли Node.js
/user/read/19	{"title": "User with id 19 found"}
/user/read/spa	Cannot GET /user/read/spa

Определение маршрутов для обновления и удаления пользователя

Маршруты для обновления и удаления пользователя выглядят так же, как для чтения пользователя, но в следующей главе мы увидим,

что действия над объектом пользователя при этом совершенно иные. В листинге 7.18 мы добавили маршруты для обновления и удаления. Изменения выделены **полужирным** шрифтом.

Листинг 7.18 ❖ Определение маршрутов для CRUD – webapp/app.js

```
/*
 * app.js – Express-сервер с более сложными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...

app.get( '/user/read/:id([0-9]+)', function ( request, response ) {
    response.contentType( 'json' );
    response.send({
        title: 'user with id ' + request.params.id + ' found'
    });
});

app.post( '/user/update/:id([0-9]+)',
    function ( request, response ) {
        response.contentType( 'json' );
        response.send({
            title: 'user with id ' + request.params.id + ' updated'
        });
    }
);

app.get( '/user/delete/:id([0-9]+)',
    function ( request, response ) {
        response.contentType( 'json' );
        response.send({
            title: 'user with id ' + request.params.id + ' deleted'
        });
    }
);
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

Создать эти маршруты было несложно, но вы, наверное, обратили внимание, что в каждом ответе мы задавали значение `contentType`. Это неэффективно и чревато ошибками – хорошо бы иметь возможность установить `contentType` сразу для всех ответов на операции CRUD. В идеале мы хотели бы создать маршрут, который перехватывает все заданные в конфигурации маршруты и устанавливает в ответе свойство `contentType` равным `json`. На этом пути нас подстерегают две трудности.

1. В одних запросах используется метод GET, в других – метод POST.
2. Мы хотим, чтобы после установки `contentType` маршрутизатор работал так же, как и раньше.

К счастью, Express опять приходит на помощь. Помимо методов `app.get` и `app.post`, существует метод `app.all`, который при сопоставлении не обращает внимания на метод отправки запроса. Кроме того, Express позволяет вернуть управление обратно маршрутизатору, чтобы узнать, есть ли еще маршруты, сопоставляемые с запросом. Для этого функции, вызываемой маршрутизатором, передается третий аргумент. По соглашению этот аргумент называется `next`, а действие его сводится к немедленной передаче управления следующей функции промежуточного уровня или следующему маршруту. Мы добавили метод `app.all` в листинге 7.19. Изменения выделены **полужирным** шрифтом.

Листинг 7.19 ❖ Использование `app.all()` для задания общих атрибутов – `webapp/app.js`

```
/*
 * app.js - Express-сервер с более сложными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...
// приведенная ниже конфигурация целиком относится к маршрутам
app.get('/', function ( request, response ) {
    response.redirect( '/spa.html' );
});

app.all( '/user/*?', function ( request, response, next ) {
    response.contentType( 'json' );
    next();
});

app.get( '/user/list', function ( request, response ) {
    // УДАЛИТЬ response.contentType( 'json' );
    response.send({ title: 'user list' });
});

app.post( '/user/create', function ( request, response ) {
    // УДАЛИТЬ response.contentType( 'json' );
    response.send({ title: 'user created' });
});

app.get( '/user/read:id([0-9]+)',
    function ( request, response ) {
        // УДАЛИТЬ response.contentType( 'json' );
        response.send({
```

```

        title: 'user with id ' + request.params.id + ' found'
    });
}
);

app.post( '/user/update/:id([0-9]+)',
function ( request, response ) {
    // УДАЛИТЬ response.contentType( 'json' );
    response.send({
        title: 'user with id ' + request.params.id + ' updated'
    });
}
);

app.get( '/user/delete/:id([0-9]+)',
function ( request, response ) {
    // УДАЛИТЬ response.contentType( 'json' );
    response.send({
        title: 'user with id ' + request.params.id + ' deleted'
    });
}
);
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...

```

В образце `/user/*?` символ `*` сопоставляется с произвольной строкой, а `?` означает, что эта строка необязательна. Таким образом, `/user/*?` сопоставится с любым из следующих маршрутов:

- `/user`
- `/user/`
- `/user/12`
- `/user/spa`
- `/user/create`
- `/user/delete/12`

Итак, маршруты мы определили. Но легко понять, что по мере добавления новых объектов количество маршрутов будет расти лавинообразно. А действительно ли нужно определять пять новых маршрутов для каждого типа объекта? К счастью, нет. Можно сделать маршруты обобщенными и вынести их в отдельный модуль.

7.3.2. Обобщенная маршрутизация для операций CRUD

Мы уже знаем, что в маршрутах можно указывать параметры, вместо которых подставляются аргументы из поступившей от клиента строки запроса. Но параметры также позволяют сделать маршруты более

общими. Нужно лишь сообщить Express, что некоторая часть URI должна рассматриваться как параметр. Вот как это делается:

```
app.get( '/:obj_type/read/:id([0-9]+)',
  function ( request, response ) {
    response.send({
      title: request.params.obj_type + ' with id '
        + request.params.id + ' found'
    });
  }
);
```

Теперь при поступлении запроса `/horse/read/12` мы получим тип объекта (`horse`) в параметре `request.params.obj_type` и отправим в ответе JSON-строку `{ title: "horse with id 12 found" }`. Применив этот способ к остальным методам, мы придем к коду, показанному в листинге 7.20. Изменения выделены **полужирным** шрифтом.

Листинг 7.20 ❖ Полный перечень обобщенных маршрутов – `webapp/app.js`

```
/*
 * app.js - Express-сервер с обобщенными маршрутами
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...
// приведенная ниже конфигурация целиком относится к маршрутам
app.get( '/', function ( request, response ) {
  response.redirect( '/spa.html' );
});

app.all( ('/:obj_type/*?'), function ( request, response, next ) {
  response.contentType( 'json' );
  next();
});

app.get( ('/:obj_type/list'), function ( request, response ) {
  response.send({ title: request.params.obj_type + ' list' });
});

app.post( ('/:obj_type/create'), function ( request, response ) {
  response.send({ title: request.params.obj_type + ' created' });
});

app.get( ('/:obj_type/read/:id([0-9]+)'),
  function ( request, response ) {
    response.send({
      title: request.params.obj_type
        + ' with id ' + request.params.id + ' found'
    });
  }
);
```

```

    });
  }
);

app.post(('/:obj_type/update/:id([0-9]+)',
  function ( request, response ) {
    response.send({
      title: request.params.obj_type
        + ' with id ' + request.params.id + ' updated'
    });
  }
);

app.get(('/:obj_type/delete/:id([0-9]+)',
  function ( request, response ) {
    response.send({
      title: request.params.obj_type
        + ' with id ' + request.params.id + ' deleted'
    });
  }
);
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----
...

```

Если теперь запустить приложение (`node app.js`) и перейти в браузере по адресу `http://localhost:3000`, то мы увидим знакомое SPA, показанное на рис. 7.1.

Это доказывает, что наша конфигурация позволяет браузеру читать любые статические файлы, будь то HTML, JavaScript или CSS.

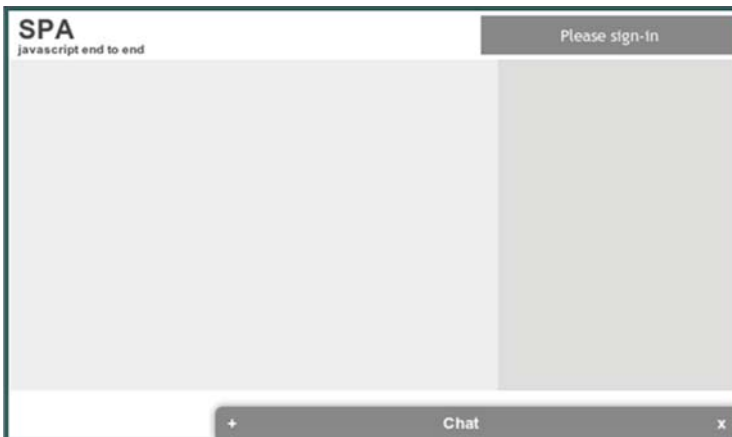


Рис. 7.1 ❖ Наше SPA в браузере – `http://localhost:3000`

Но при этом мы еще имеем доступ к API CRUD-операций. Перейдя в браузере по адресу `http://localhost:3000/user/read/12`, мы получим в ответ:

```
{
  title: "user with id 12 found"
}
```

А что, если бы по адресу `<root_directory>/user/read/12` находился реальный файл (не смейтесь, еще и не такое бывает)? В таком случае был бы возвращен этот файл, а не ответ на CRUD-операцию. И дело здесь в том, что метод промежуточного уровня `express.static` мы поместили до маршрутизатора:

```
...
app.configure( function () {
  app.use( express.bodyParser() );
  app.use( express.methodOverride() );
  app.use( express.static( __dirname + '/public' ) );
  app.use( app.router );
});
...
```

А если бы мы изменили порядок и поставили маршрутизатор первым, то сервер вернул бы ответ на CRUD-операцию, а не статический файл. Преимуществом такого подхода является более быстрый ответ на CRUD-запросы, недостатком – более медленный и не всегда успешный доступ к файлам. *Правильное* решение – ставить в начале любого CRUD-запроса общий логический корень, например `/api/1.0.0/`, тогда статическое и динамическое содержимые будут четко отделены друг от друга.

Итак, у нас есть чистый обобщенный маршрутизатор, пригодный для управления объектами любого типа. Очевидно, что в нем не учтены вопросы авторизации, но это мы добавим позже. А пока переместим всю логику маршрутизации в отдельный модуль.

7.3.3. Перенос маршрутизации в отдельный модуль Node.js

Хранить все маршруты в головном файле `app.js` – все равно что помещать клиентский JavaScript-код в HTML-страницу: это только загромождает приложение и препятствует четкому разделению обязанностей. Познакомимся поближе с системой модулей в Node.js, которая позволяет писать модульный код.

Модули Node

Модули в Node.js загружаются с помощью функции `require`.

```
var spa = require( './routes' );
```

Передаваемая в аргументе строка – это путь к загружаемому файлу. Необходимо запомнить несколько синтаксических правил, поэтому наберитесь терпения. Для удобства правила сведены в табл. 7.3.

Таблица 7.3. Логика поиска файла в функции Node `require`

Синтаксис	Пути поиска в порядке приоритета
<code>require('./routes.js');</code>	app/routes.js
<code>require('./routes');</code>	app/routes.js app/routes.json app/routes.node
<code>require('../routes.js');</code>	../routes.js
<code>require('routes.');</code>	app/node_modules/routes.js app/node_modules/routes/index.js <system_install>/node_modules/routes.js <system_install>/node_modules/routes/index.js Такой же синтаксис применяется для ссылки на базовые модули node.js, например http

Внутри модуля Node видимость переменных, объявления которых начинаются с ключевого слова `var`, ограничена самим модулем, поэтому не нужно писать самовыполняющуюся анонимную функцию, чтобы не дать переменной просочиться в глобальную область видимости, как на стороне клиента. Вместо этого существует объект `module`. Значение, присвоенное атрибуту `module.exports`, станет значением, которое вернет метод `require`. Давайте напишем модуль `routes`, как показано в листинге 7.21.

Листинг 7.21 ❖ Модуль `routes` – `webapp/routes.js`

```
module.exports = function () {
    console.log( 'Вы включили модуль routes.' );
};
```

Значение `module.exports` может иметь любой тип: функция, объект, массив, строка, число или булева величина. В данном случае значением `module.exports` является анонимная функция. Включим файл

`routes.js` в `app.js` с помощью `require` и сохраним возвращенное значение в переменной `routes`. Впоследствии мы сможем вызвать возвращенную функцию, как показано в листинге 7.22. Изменения выделены **полужирным** шрифтом.

Листинг 7.22 ❖ Включение модуля и использование возвращенного значения – `webapp/app.js`

```
/*
 * app.js - Express-сервер с примером модуля
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  http    = require( 'http' ),
  express = require( 'express' ),
  routes  = require( './routes' ),
  app     = express(),
  server  = http.createServer( app );

routes();
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
...
```

Выполнив команду `node app.js`, мы увидим такие сообщения:

Вы включили модуль `routes`.

Express-сервер прослушивает порт 3000 в режиме `development`

Теперь перенесем в модуль `routes` конфигурацию маршрутизатора.

Перенос маршрутов в модуль

Создавая нетривиальное приложение, мы хотим, чтобы все маршруты находились в одном файле в главной папке приложения. Если маршрутов очень много, то можно распределить их по разным файлам и поместить их в папку `routes`.

Поскольку наше приложение нельзя назвать тривиальным, создадим в корневом каталоге `spa` файл `routes.js` и скопируем в него существующие маршруты, поместив их в тело функции `module.exports` (рис. 7.23).

Листинг 7.23 ❖ Перенос маршрутов в отдельный модуль – `webapp/routes.js`

```
/*
 * routes.js - модуль, содержащий маршруты
 */
/*jslint      node : true, continue : true,
   devel     : true, indent : 2,    maxerr  : 50,
```

```

newcap : true, nomen : true, plusplus : true,
regexp : true, sloppy : true, vars      : false,
white  : true
*/
/*global */
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var configRoutes;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
configRoutes = function ( app, server ) {
  app.get( '/', function ( request, response ) {
    response.redirect( '/spa.html' );
  });

  app.all( '/:obj_type/*?', function ( request, response, next ) {
    response.contentType( 'json' );
    next();
  });

  app.get( '/:obj_type/list', function ( request, response ) {
    response.send({ title: request.params.obj_type + ' list' });
  });

  app.post( '/:obj_type/create', function ( request, response ) {
    response.send({ title: request.params.obj_type + ' created' });
  });

  app.get( '/:obj_type/read/:id([0-9]+)',
    function ( request, response ) {
      response.send({
        title: request.params.obj_type
          + ' with id ' + request.params.id + ' found'
      });
    }
  );

  app.post( '/:obj_type/update/:id([0-9]+)',
    function ( request, response ) {
      response.send({
        title: request.params.obj_type
          + ' with id ' + request.params.id + ' updated'
      });
    }
  );

  app.get( '/:obj_type/delete/:id([0-9]+)',
    function ( request, response ) {

```

Переменные `app` и `server` не глобальные, поэтому их нужно передавать функции явно. Node.js прилагает все усилия к тому, чтобы переменные, определенные в одном модуле или в головном приложении, не оказывали влияния на переменные в других модулях.

← Устанавливаем тип содержимого json.

```

    response.send({
      title: request.params.obj_type
        + ' with id ' + request.params.id + ' deleted'
    });
  }
};
module.exports = { configRoutes : configRoutes };
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Экспортируем метод, чтобы его можно было вызвать, когда файл `webapp/app.js` будет к этому готов.

Теперь можно воспользоваться модулем маршрутизации в файле `webapp/app.js`, как показано в листинге 7.24. Изменения выделены **полужирным шрифтом**.

Листинг 7.24 ❖ Серверное приложение пользуется маршрутами, заданными во внешнем модуле – `webapp/app.js`

```

/*
 * app.js - Express-сервер с модулем маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  http    = require( 'http' ),
  express = require( 'express' ),
  routes  = require( './routes' ), ← Загружаем модуль routes.

  app      = express(),
  server   = http.createServer( app );
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
  app.use( express.bodyParser() );
  app.use( express.methodOverride() );
  app.use( express.static( __dirname + '/public' ) );
  app.use( app.router );
});

app.configure( 'development', function () {
  app.use( express.logger() );
  app.use( express.errorHandler({
    dumpExceptions : true,
    showStack : true
  }) );
});

app.configure( 'production', function () {
  app.use( express.errorHandler() );

```

```

});
routes.configRoutes( app, server );
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----

// ----- НАЧАЛО ЗАПУСКА СЕРВЕРА -----
server.listen( 3000 );
console.log(
  'Express-сервер прослушивает порт %d в режиме %s',
  server.address().port, app.settings.env
);
// ----- КОНЕЦ ЗАПУСКА СЕРВЕРА -----

```

Используем метод `configRoutes`
для настройки маршрутов.

В результате файл `app.js` получается достаточно прозрачным: он загружает необходимые библиотечные модули, создает Express-приложение, конфигурирует промежуточный уровень, добавляет маршруты и запускает сервер. Чего он не делает, так это сохранения данных в базе в результате выполнения запрошенных действий. Это упущение мы исправим в следующей главе, после того как настроим MongoDB и подключим ее к нашему приложению Node.js. Но прежде рассмотрим еще кое-какие вещи, которые понадобятся раньше.

7.4. Добавление аутентификации и авторизации

Создав маршруты для выполнения CRUD-операций над объектами, мы должны позаботиться об аутентификации. Можно пойти трудным путем и написать весь код самостоятельно или упростить себе жизнь, воспользовавшись еще одним встроенным в Express методом промежуточного уровня. Гм. Думай, голова... Что же выбрать?

7.4.1. Базовая аутентификация

Базовая аутентификация – описанный в стандартах HTTP/1.0 и 1.1 способ предоставления клиентом имени и пароля пользователя при отправке запроса; часто его называют просто *basic auth*. Напомним, что функции промежуточного уровня вызываются в том порядке, в котором перечислены в приложении, поэтому если мы хотим, чтобы приложение авторизовало доступ к маршрутам, то соответствующую функцию следует добавить раньше маршрутизатора. Это несложно – изменения выделены в листинге 7.25 **полужирным** шрифтом.

Листинг 7.25 ❖ Добавление базовой аутентификации в серверное приложение – webapp/app.js

```

/*
 * app.js - Express-сервер с базовой аутентификацией
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
    app.use( express.bodyParser() );
    app.use( express.methodOverride() );
    app.use( express.basicAuth( 'user', 'spa' ) );
    app.use( express.static( __dirname + '/public' ) );
    app.use( app.router );
});
...

```

В данном случае мы «зашили» в код имя `user` и пароль `spa`. Метод `basicAuth` принимает также в качестве третьего параметра функцию, которая может реализовывать более сложный механизм, например поиск в базе данных. Эта функция должна вернуть `true`, если пользователь аутентифицирован, и `false` – в противном случае. После того как мы перезапустим сервер и перезагрузим страницу в браузере, должно появиться диалоговое окно, показанное на рис. 7.2, в котором для получения доступа предлагается ввести имя пользователя и пароль.

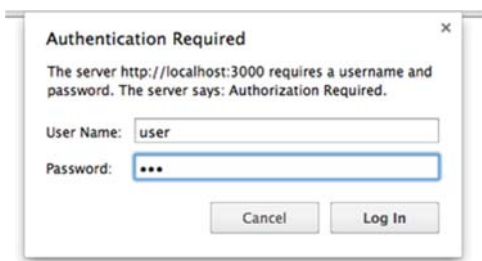


Рис. 7.2 ❖ Диалоговое окно аутентификации в Chrome

Если мы введем неверный пароль, то окно появится снова – и так до получения правильного ответа. Нажав кнопку **Cancel**, мы увидим страницу с сообщением `Unauthorized`.

Базовую аутентификацию *не* рекомендуется использовать в производственных приложениях. В этом случае регистрационные данные посылаются в каждом запросе в открытом виде – эксперты по

безопасности называют это *большим вектором атаки*. И даже зашифровав передачу с помощью SSL (HTTPS), мы добавим всего один уровень безопасности между клиентом и сервером.

Изобретение собственных механизмов аутентификации постепенно уходит в прошлое. Многие стартапы и даже давно зарекомендовавшие себя компании начинают пользоваться аутентификацией через сторонние сайты, например Facebook или Google. В Сети существует немало руководств, показывающих, как интегрировать свое приложение с этими сервисами; в качестве отправной точки можете воспользоваться функцией промежуточного уровня Passport, входящей в состав Node.js.

7.5. Веб-сокеты и Socket.IO

Веб-сокеты – впечатляющая технология, которая все шире поддерживается браузерами. Веб-сокет позволяет поддерживать постоянный, двунаправленный, не слишком ресурсоемкий канал связи между клиентом и сервером по одному TCP-соединению. В результате как клиент, так и сервер получают возможность в реальном времени посылать сообщения без накладных расходов и задержек, присущих циклу запрос–ответ в протоколе HTTP. До появления веб-сокетов разработчики использовали другие – не столь эффективные – способы добиться того же результата. К ним относятся сокеты Flash; долгий опрос, когда браузер открывает запрос серверу, а затем повторно инициализирует его, когда приходит ответ или возникает тайм-аут; и периодический опрос сервера с небольшим интервалом (например, один раз в секунду).

Недостаток веб-сокетов заключается в том, что окончательная спецификация еще не готова, а старые браузеры эту технологию никогда не будут поддерживать. Socket.IO – это модуль Node.js, который решает последнюю проблему за счет того, что пользуется веб-сокетами, когда они доступны, но переходит на другие технологии в противном случае.

7.5.1. Простой пример применения Socket.IO

Сейчас мы напишем приложение, которое один раз в секунду обновляет некий счетчик на сервере и с помощью Socket.IO рассылает новое значение всем подключившимся клиентам. Для установки Socket.IO модифицируем наш файл `package.json`, как показано в листинге 7.26.

Листинг 7.26 ❖ Установка Socket.IO – webapp/package.json

```
{
  "name"      : "SPA",
  "version"   : "0.0.3",
  "private"   : true,
  "dependencies" : {
    "express"  : "3.2.x",
    "socket.io" : "0.9.x"
  }
}
```

Выполнив `npm install`, мы можем быть уверены, что Express и Socket.IO установлены.

Добавим два файла: серверное приложение `webapp/socket.js` и HTML-документ `webapp/socket.html`. Сначала напишем серверное приложение, которое умеет обслуживать статические файлы и содержит таймер, по которому счетчик увеличивается один раз в секунду. Поскольку мы знаем, что воспользуемся библиотекой Socket.IO, включим и ее тоже. Новое серверное приложение `socket.js` приведено в листинге 7.27.

Листинг 7.27 ❖ Начало серверного приложения – webapp/socket.js

```
/*
 * socket.js – простой пример применения socket.io
 */
/*jslint      node    : true, continue : true,
  devel      : true, indent : 2,  maxerr  : 50,
  newcap     : true, nomen  : true, plusplus : true,
  regexp     : true, sloppy : true, vars   : false,
  white      : true
*/
/*global */

// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  countUp,
  http    = require( 'http' ),
  express = require( 'express' ),
  socketIo = require( 'socket.io' ),
  app      = express(),
  server   = http.createServer( app ),
  countIdx = 0 ◀ Объявляем переменную-счетчик в области видимости модуля.
  ;

// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
countUp = function () { ◀ Функция, которая увеличивает счетчик и протоколирует это действие.
```

```

    countIdx++;
    console.log( countIdx );
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
    app.use( express.static( __dirname + '/' ) );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----

// ----- НАЧАЛО ЗАПУСКА СЕРВЕРА -----
server.listen( 3000 );
console.log(
    'Express-сервер прослушивает порт %d в режиме %s',
    server.address().port, app.settings.env
);
setInterval( countUp, 1000 );
// ----- КОНЕЦ ЗАПУСКА СЕРВЕРА -----

```

Настраиваем приложение для обслуживания статических файлов из текущего каталога.

С помощью встроенной в JavaScript функции `setInterval` организуем вызов функции `countUp` один раз в секунду.

Запустив сервер (`node socket.js`), мы увидим, что на консоли печатаются сообщения с монотонно увеличивающимся счетчиком. Теперь создадим показанный в листинге 7.28 файл `webapp/socket.html`, который выводит счетчик. Мы включаем библиотеку jQuery, потому что с ее помощью легко манипулировать тегом `body`.

Листинг 7.28 ❖ HTML-документ – `webapp/socket.html`

```

<!doctype html>
<!-- socket.html - простой пример применения socket.io -->
<html>
<head>
    <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"
    ></script>
</head>
<body>
    Загрузка...
</body>
</html>

```

Теперь мы можем перейти в браузере по адресу `http://localhost:3000` – и увидеть почти пустую страницу. Чтобы заставить Socket.IO посылать информацию клиенту, нужно добавить всего две строки в сервер-

ное приложение. Изменения показаны в листинге 7.29 полужирным шрифтом.

Листинг 7.29 ❖ Добавление веб-сокета в серверное приложение – webapp/socket.js

```
...
server = http.createServer( app ),
io = socketIo.listen( server ), ← Инструктируем Socket.IO прослушивать,
countIdx = 0                      применяя наш HTTP-сервер.
;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
countUp = function () {
    countIdx++;
    console.log( countIdx );
    io.sockets.send( countIdx ); ← Пыслаем значение счетчика во все подключенные сокеты.
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
...
```

Чтобы воспользоваться Socket.IO на стороне клиента, в головной HTML-документ нужно добавить шесть строчек. Они выделены в листинге 7.30 полужирным шрифтом.

Листинг 7.30 ❖ Добавление веб-сокета в HTML-документ – webapp/socket.html

```
<!doctype html>
<!-- socket.html – простой пример применения socket.io -->
<html>
<head>
    <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"
></script>
    <script src="/socket.io/socket.io.js"></script>
    <script>
        io.connect().on( 'message', function ( count ) {
            $('body').html( count );
        });
    </script>
</head>
<body>
    Загрузка...
</body>
</html>
```

JavaScript-файл /socket.io/socket.io.js является частью дистрибутива Socket.IO, так что нам его создавать не нужно. На самом деле

это «волшебный» файл, которого на сервере вообще нет, вы тщетно будете искать его на диске. Метод `io.connect()` возвращает соединение `Socket.IO`, а метод `on` аналогичен методу `bind` в `jQuery` – он просит браузер следить за возникновением определенного события `Socket.IO`. В данном случае интересующее нас событие – приход нового сообщения по соединению. Затем мы с помощью `jQuery` записываем в тело документа новое значение счетчика. А ведь вы все-таки попробовали найти файл `socket.io.js` на сервере, а?

Перейдя в браузере по адресу `http://localhost:3000/`, мы увидим, что счетчик увеличивается. Открыв ту же страницу на другой вкладке, мы увидим, что счетчик меняется и там – в том же темпе и принимая то же самое значение. Так происходит потому, что `countIdx` – переменная в серверном приложении, находящаяся в области видимости модуля.

7.5.2. `Socket.IO` и сервер обмена сообщениями

Используя `Socket.IO` для передачи сообщений между клиентами и серверами, мы создаем сервер обмена сообщениями. Другим примером сервера обмена сообщениями является *Openfire*, который передает сообщения по протоколу `XMPP`, используемому в `Google Chat` и `Jabber`. Сервер обмена сообщениями должен держать открытыми соединения со всеми клиентами, чтобы они могли быстро получать сообщения и отвечать на них. Сервер также должен минимизировать размер сообщения, чтобы не передавать излишние данные.

Традиционные веб-серверы типа `Apache2` не годятся на роль серверов обмена сообщениями, потому что они ассоциируют с каждым соединением процесс (или поток), и *каждый такой процесс должен работать, пока существует соединение*. Нетрудно догадаться, что когда количество соединений достигнет нескольких сотен или тысяч, эти процессы потребуют все ресурсы веб-сервера. `Apache2` никогда не предназначался для работы в таком режиме; его проектировали как сервер содержимого, который должен как можно быстрее отдать данные в ответ на запрос и сразу после этого закрыть соединение. Вот на таких задачах `Apache2` работает великолепно: взгляните на `YouTube` – и убедитесь.

Напротив, `Node.js` прекрасно себя чувствует в роли сервера обмена сообщениями. Благодаря модели событий он *не* создает отдельного процесса для каждого соединения, а просто следит за тем, когда соединение открывается и закрывается, и ведет внутренний учет. Поэтому он может обрабатывать десятки и сотни тысяч соединений на сравни-

тельно скромном оборудовании. Node.js не делает ничего существенного, пока на одном из открытых соединений не произойдет событие, требующее обмена сообщениями, – например, запрос или ответ.

Сколько обменивающихся сообщениями клиентов способен поддерживать Node.js, зависит от фактической нагрузки на сервер. Если клиенты в основном молчат, а выполняемые сервером задачи не требуют много ресурсов, то сервер может обслуживать *ну очень много* клиентов. Если же клиенты болтливы, а поручаемые серверу задачи посложнее, то количество обслуживаемых клиентов будет *гораздо меньше*. Вполне можно представить себе высоконагруженную среду, в которой балансировщик нагрузки распределяет трафик между кластером серверов Node.js, обеспечивающим обмен сообщениями, другим кластером серверов Node.js, обслуживающим динамическое содержимое, и кластером серверов Apache2, отвечающим за статическое содержимое.

У Node.js, по сравнению с другими протоколами обмена сообщениями, в частности XMPP, есть целый ряд преимуществ. Назовем лишь некоторые.

Благодаря Socket.IO организация кросс-браузерного обмена сообщениями в веб-приложении становится почти тривиальным делом. Раньше нам довелось писать промышленное приложение на основе XMPP. Поверьте нам – работы гораздо больше.

Можно обойтись без отдельного сервера и его настройки. Еще один существенный выигрыш.

Можно работать с естественным протоколом на базе JSON, а не с другим языком. XMPP основан на XML и нуждается в сложном программном обеспечении для кодирования и декодирования.

Нам не пришлось мучиться (по крайней мере, на первых порах) с проклятым «правилом ограничения домена», которое отравляет жизнь на других платформах. Это правило запрещает загружать в браузер содержимое, если оно находится не на том сервере, с которого был загружен JavaScript-код.

А теперь посмотрим на применение Socket.IO, которое, безусловно, произведет на вас впечатление: динамическое обновление нашего SPA.

7.5.3. Обновление JavaScript-кода с помощью Socket.IO

В нашем SPA есть одна важная проблема: как обеспечить согласованность клиентской и серверной частей приложения? Представьте, что Бобби загрузила SPA в свой браузер, а аккурат через пять минут мы

обновили серверную часть. Теперь у Бобби неприятности – новое серверное приложение передает данные в новом формате, а клиентская часть в браузере Бобби рассчитана на старый формат. Решить проблему можно, заставив Бобби полностью перезагрузить SPA, как только будет обнаружено, что оно устарело, – скажем, после того как мы отправим сообщение с объявлением о выходе новой версии серверной части. Но можно поступить хитрее – избирательно обновить только нуждающиеся в изменении JavaScript-файлы, не перезагружая приложения целиком.

Но как совершить это волшебное обновление? Необходимо решить три задачи.

1. Обнаруживать, что JavaScript-файлы изменились.
2. Уведомлять клиента о том, что файл изменился.
3. Обновлять JavaScript-файлы на стороне клиента после получения уведомления об изменении.

Первую задачу – обнаружение изменившихся файлов – можно решить с помощью системного модуля `fs`, входящего в состав Node. Для решения второй достаточно с помощью `Socket.IO` послать уведомление браузеру, как описано в предыдущем разделе. А для обновления клиентской части можно, получив такое уведомление, динамически вставить новый тег `script`. Новая версия серверного приложения показана в листинге 7.31. Изменения выделены **полужирным** шрифтом.

Листинг 7.31 ❖ Отслеживание файлов в серверном приложении – `webapp/socket.js`

```
/*
 * socket.js – динамическая загрузка JS
 */
/*jslint      node    : true, continue : true,
  devel      : true, indent : 2,  maxerr  : 50,
  newcap     : true, nomen  : true, plusplus : true,
  regexp     : true, sloppy : true, vars    : false,
  white      : true
*/
/*global */

// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  setWatch,

  http    = require( 'http' ),
  express = require( 'express' ),
  socketIo = require( 'socket.io' ),
```



```

fsHandle = require( 'fs' ), ← Загружаем модуль файловой системы в fsHandle.

app      = express(),
server   = http.createServer( app ),
io       = socketIo.listen( server ),
watchMap = {}
;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
setWatch = function ( url_path, file_type ) {
  console.log( 'setWatch called on ' + url_path );

  if ( ! watchMap[ url_path ] ) {
    console.log( 'начинаю наблюдение за ' + url_path );

    fsHandle.watchFile( ← Настраиваем модуль файловой системы
                        на отслеживание изменения файлов.

                        url_path.slice(1), ←
                        function ( current, previous ) {
                          console.log( 'обнаружен доступ к файлу' );
                          if ( current.mtime !== previous.mtime ) { ←
                            console.log( 'файл изменен' );
                            io.sockets.emit( file_type, url_path );
                          }
                        }
                      );
    watchMap[ url_path ] = true;
  }
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
  app.use( function ( request, response, next ) { ←
    if ( request.url.indexOf( '/js/' ) >= 0 ) { ← Если запрошен файл из каталога js, считаем, что это скрипт (событие script).
      setWatch( request.url, 'script' );
    }
    else if ( request.url.indexOf( '/css/' ) >= 0 ) { ←
      setWatch( request.url, 'stylesheet' );
    }
    next();
  });
  app.use( express.static( __dirname + '/' ) );
});

app.get( '/', function ( request, response ) {
  response.redirect( '/socket.html' );
});
// ----- КОНЕЦ КОНФИГУРАЦИИ СЕРВЕРА -----

```

Убираем / из пути url_path, потому что модуль файловой системы предполагает, что задан путь относительно текущего каталога.

Сравниваем текущее время последней модификации (mtime) файла с предыдущим. Если они различаются, значит, файл был изменен.

Отправляем клиенту событие script или stylesheet, указывая в нем путь к измененному файлу.

Используем нестандартный метод промежуточного уровня для наблюдения за статическими файлами.

Если запрошен файл из каталога css, считаем, что это таблица стилей (событие stylesheet).

```
// ----- НАЧАЛО ЗАПУСКА СЕРВЕРА -----
server.listen( 3000 );
console.log(
  'Express-сервер прослушивает порт %d в режиме %s',
  server.address().port, app.settings.env
);

// ----- КОНЕЦ ЗАПУСКА СЕРВЕРА -----
```

Подготовив серверную часть, обратимся к клиентской: сначала рассмотрим JavaScript-файл, который собираемся обновлять, а затем – главную страницу. Наш файл данных `webapp/js/data.js` содержит единственную строку, в которой переменной присваивается текстовое значение (листинг 7.32).

Листинг 7.32 ❖ Создание файла данных – `webapp/js/data.js`

```
var b = 'SPA';
```

В головной HTML-файл придется внести более существенные изменения, они выделены **полужирным** шрифтом в листинге 7.33.

```
<!doctype html>
<!-- socket.html - пример динамической загрузки JS -->
<html>
<head>
  <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"
  ></script>
  <script src="/socket.io/socket.io.js"></script>
  <script id="script_a" src="/js/data.js"></script>
  <script>
    $(function () {
      $( 'body' ).html( b );
    });
    io.connect('http://localhost').on( 'script', function ( path ) {
      $( '#script_a' ).remove();
      $( 'head' ).append(
        '<script id="script_a" src="'
        + path +
        '></scr' + 'ipt>'
      );
      $( 'body' ).html( b );
    });
  </script>
</head>
<body>
  Загрузка...
</body>
</html>
```

Включаем JavaScript-файл, который собираемся обновить.

При первой загрузке страницы помещаем в тело HTML-страницы значение переменной `b`, присвоенное в файле `data.js`.

Получив от сервера событие `script`, выполняем эту функцию.

Удаляем старый тег `script` и вставляем новый, указывающий на измененный JavaScript-файл (`webapp/js/data.js`). В результате будет выполнен находящийся в этом файле JavaScript-код – в данном случае присваивание нового значения переменной `b`.

Заменяем тело HTML-документа новым значением переменной `b`.

Вот теперь можно сотворить волшебство. Сначала запустите серверное приложение (введите команду `node socket.js`). Затем откройте головной HTML-документ (`webapp/socket.html`) в браузере. На странице появится слово SPA. Далее замените в файле `webapp/js/data.js` слово SPA на смысл жизни в брютке или еще какое-нибудь столь же глубокомысленное высказывание. Вернувшись в браузер, вы увидите, что содержимое страницы изменилось (хотя вы ничего не перезагружали). Возможна задержка в несколько секунд, так как команда `watchFile` обнаруживает, что файл изменился, не сразу¹.

7.6. Резюме

В этой главе мы видели, что хотя значительная часть логики SPA перенесена на сторону клиента, сервер по-прежнему отвечает за аутентификацию, валидацию и хранение данных. Мы установили сервер Node.js и с помощью каркасов промежуточного уровня Connect и Express без труда реализовали маршрутизацию, протоколирование и аутентификацию.

Вынесение логики маршрутизации и конфигурирования в отдельные файлы позволило сохранить понятность главной программы, а Express дает возможность определять разные конфигурации в зависимости от окружения. Express также предоставляет средства создания не зависящих от типа объекта маршрутов для операций CRUD.

Мы еще не приступали к валидации и хранению данных – это задача следующей главы, где мы сведем вместе приложение и данные.

¹ В производственной системе мы обычно хотим опрашивать файлы (системный вызов `fstat`) как можно реже, потому что эта операция здорово сажает производительность. У метода `watchFile` есть параметры, позволяющие снизить частоту опроса файлов. Например, можно задать частоту 30 000 (опрашивать один раз в 30 000 миллисекунд, или 30 секунд) вместо подразумеваемого по умолчанию значения 0 (то есть «опрашивай часто, еще чаще, как можно чаще»).

Глава 8

Серверная база данных

В этой главе:

- ✧ Роль базы данных в SPA.
- ✧ Использование JavaScript как языка базы данных в MongoDB.
- ✧ Драйвер MongoDB для Node.js.
- ✧ Реализация операций CRUD.
- ✧ Применение JSV для валидации данных.
- ✧ Уведомление клиента об изменении данных с помощью Socket.IO.

Эта глава опирается на код, написанный в главе 7. Мы рекомендуем скопировать все созданные там файлы в новый каталог «chapter_8» и уже в нем производить изменения.

В этой главе мы добавим в наше SPA базу данных для постоянного хранения. Это последний кирпичик в нашем представлении о сквозном использовании JavaScript – в базе данных, на сервере и на клиенте. Закончив работу, мы сможем запустить серверное приложение Node.js и пригласить друзей зарегистрироваться в нашем SPA на своих компьютерах или сенсорных устройствах. После этого они смогут поболтать друг с другом или поменять свои аватары, причем все участники будут видеть изменения почти в реальном масштабе времени. Начнем с более пристального изучения роли базы данных.

8.1. Роль базы данных

Мы используем сервер базы данных для обеспечения надежного и постоянного хранения данных. Для этой цели необходим сервер, потому что данные, хранящиеся на стороне клиента, носят переходный характер и уязвимы для ошибок приложения и пользователя, а также для злонамеренного манипулирования пользователем. К тому же данные, хранящиеся на стороне клиента, трудно предоставить в общее пользование, и доступны они, только когда клиент находится в сети.

8.1.1. Выбор хранилища данных

Выбор серверных решений для хранения данных весьма широк, назовем хотя бы реляционные СУБД, хранилища ключей и значений и базы данных NoSQL. Но какое решение лучшее? Как часто бывает в жизни, ответ зависит от обстоятельств. Нам доводилось работать с веб-приложениями, в которых для разных целей использовалось сразу несколько технологий. Написаны тома о достоинствах различных хранилищ: реляционных баз данных (например, *MySQL*), хранилищ ключей и значений (например, *memcached*), графовых баз данных (например, *Neo4J*) или документных баз данных (например, *Cassandra* и *MongoDB*). Их сравнение выходит за рамки этой книги, хотя авторы в этом отношении считают себя агностиками и полагают, что у любого решения есть своя ниша.

Допустим, мы написали текстовый процессор в виде SPA. Мы могли бы хранить сами файлы в стандартной файловой системе, но индексировать их с помощью *MySQL*. А объекты, необходимые для аутентификации, хранить в *MongoDB*. В любом случае пользователь почти наверняка ожидает, что его документы будут длительно храниться на сервере. Иногда пользователь захочет сохранить файл на своем локальном диске и прочитать его оттуда, и, конечно же, мы предоставим ему такую возможность. Но локальное хранение постепенно утрачивает притягательность по мере того, как растут полезность, надежность и доступность сетей и удаленных хранилищ.

Мы выбрали в качестве хранилища данных *MongoDB* по ряду причин: на практике доказано, что эта СУБД надежна, масштабируема, обладает приличной производительностью и – в отличие от других NoSQL-решений – позиционируется как база данных общего назначения. Мы считаем, что она хорошо приспособлена для SPA, потому что позволяет использовать JavaScript и JSON на обеих сторонах. В ее командном интерфейсе языком запросов является JavaScript, так что для исследования базы и для манипулирования данными мы можем использовать такие же конструкции JavaScript, как в серверном или в браузерном окружении. В качестве формата хранения используется JSON, а средства управления данными специально «заточены» под JSON.

8.1.2. Исключение преобразования данных

Рассмотрим традиционное веб-приложение, написанное на *MySQL*/*Ruby on Rails* (или *mod_perl*, *PHP*, *ASP*, *Java*, *Python*) и JavaScript: разработчик должен писать код для преобразований по цепочке SQL

-> Active Record -> JSON на пути к клиенту и затем по цепочке JSON -> Active Record -> SQL на обратном пути (см. рис. 8.1). Мы имеем три языка (SQL, Ruby, JavaScript), три формата данных (SQL, Active Record, JSON) и четыре преобразования. В лучшем случае это влечет за собой растраниживание ресурсов сервера, которые можно было бы с пользой потратить на что-то другое. В худшем – каждое преобразование является потенциальным источником ошибок, а для его реализации и сопровождения приходится тратить немало усилий.

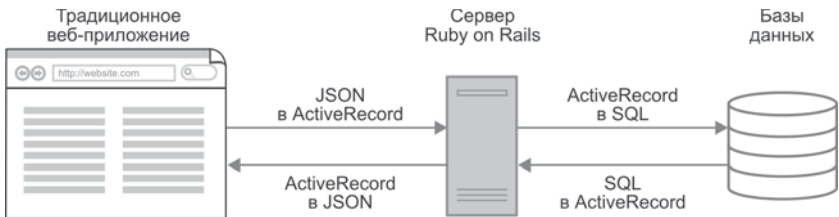


Рис. 8.1 ❖ Преобразования данных в веб-приложении

Мы же используем MongoDB, Node.js и написанное на JavaScript естественное SPA, поэтому преобразование данных на пути туда и обратно выглядит так: JSON -> JSON -> JSON (см. рис. 8.2). Мы работаем с одним языком (JavaScript) и одним форматом данных (JSON), так что никакой трансформации данные не подвергаются. В результате когда-то сложная система существенно упрощается.

Благодаря этой же простоте мы получаем большую гибкость при решении вопроса о том, в каком месте разместить логику приложения.

8.1.3. Помещайте логику туда, где она нужнее

В традиционном веб-приложении возникает вопрос: куда поместить ту или иную логику? В хранимую SQL-процедуру? Или в серверную часть приложения? А может, лучше в клиентскую часть? Перенос же логики с одного уровня на другой обычно становится грандиозной проблемой из-за использования разных языков и форматов данных. Другими словами, ошибка обходится несоразмерно дорого (представьте, что нужно переписать код с Java на JavaScript). Это ведет к компромиссным «безопасным» решениям, ограничивающим возможности приложения.

Использование единого языка и формата данных заметно снижает цену изменения решения. Это позволяет проявлять больше выдумки

при разработке, потому что *плата за ошибку минимальна*. Если возникнет необходимость переместить часть логики с сервера на клиент, то мы сможем воспользоваться уже написанным JavaScript-кодом с минимальными изменениями или вообще без оных.

Давайте познакомимся с выбранной базой данных MongoDB поближе.

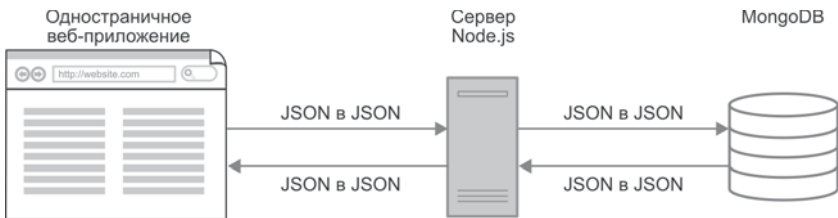


Рис. 8.2 ❖ При использовании MongoDB преобразований данных Node.js и SPA нет вовсе

8.2. Введение в MongoDB

Согласно информации на сайте, MongoDB – это «масштабируемая высокопроизводительная база данных класса NoSQL с открытым исходным кодом» на основе документоориентированного хранилища с динамическими схемами, обеспечивающего «простоту и мощь». Разберемся, что это описание означает.

- *Масштабируемая высокопроизводительная* – MongoDB проектировалась для масштабирования по горизонтали с использованием недорогих серверов. В случае реляционных баз данных единственный способ масштабирования базы данных – приобретение более качественного оборудования¹. В случае MongoDB для повышения емкости или производительности достаточно поставить дополнительный сервер.
- *Документоориентированное хранилище* – в MongoDB данные хранятся в виде документов формата JSON, а не в виде таблиц со строками и столбцами. Документы – грубый аналог строк SQL – организованы в коллекции – аналог таблиц SQL.
- *Динамические схемы* – в реляционных базах необходима схема, определяющая, что можно хранить в таблицах, тогда как в Мон-

¹ Да, можно создавать кластеры и реплики даже в реляционных СУБД, но их настройка и обслуживание требуют больших знаний и опыта. Купить более быстрый сервер гораздо проще.

goDB схема не нужна. В коллекции можно сохранить любой JSON-документ. У разных документов в пределах одной коллекции может быть совершенно различная структура, причем во время обновления структура может полностью измениться.

Известие о производительности придется по душе любому, особенно руководителям отделов эксплуатации. Два других пункта представляют особый интерес для разработчиков SPA и заслуживают подробного рассмотрения. Если вы уже знакомы с MongoDB, можете сразу перейти к разделу 8.3, где мы подключим базу к нашему приложению для Node.js.

8.2.1. Документоориентированное хранилище

В MongoDB данные хранятся в виде документов в формате JSON, что очень удобно для большинства SPA. Конкретно в нашем SPA JSON-документы можно сохранять и извлекать без преобразования¹. Это заманчиво, так как дает возможность не тратить времени на этапе разработки и выполнения на многократное переформатирование данных. Поскольку форматы идентичны, то, обнаружив ошибку в данных на стороне клиента, мы просто должны проверить, есть ли она также и в базе.

Это не только ускоряет и упрощает разработку приложения, но и сулит выигрыш в его производительности. Сервер просто передает клиенту то, что хранится в базе, без каких бы то ни было манипуляций. Это положительно сказывается также на стоимости хостинга и масштабировании приложения, потому что серверам приходится выполнять меньше работы. В данном случае нагрузка не переносится на сторону клиента, а благодаря единому формату данных просто ликвидируется. Это необязательно означает, что комбинация Node.js + MongoDB быстрее, чем Java + PostgreSQL, – на общее быстродействие приложения влияет много других факторов, – но совершенно точно можно сказать, что при прочих равных условиях единый формат данных должен обеспечивать лучшую производительность.

8.2.2. Динамическая структура документа

В MongoDB на структуру документов не налагается ограничений. Не определяя никакой структуры, мы просто добавляем документы

¹ Сравните с реляционными СУБД, где сначала нужно преобразовать документ в команду SQL для сохранения, а после извлечения преобразовать данные в формат JSON.

в коллекцию. Даже не нужно предварительно создавать коллекцию командой `create` – при вставке данных в несуществующую коллекцию она создается автоматически. Сравните с реляционной базой данных, где необходимо явно определять таблицы и схемы, а всякое изменение в структуре данных требует изменения схемы. База данных, обходящаяся без схемы, дает ряд интересных преимуществ.

- *Гибкая структура документов.* В MongoDB документы хранятся безотносительно к структуре. Даже если документ вообще не структурирован или его структура часто изменяется, MongoDB сохранит его, не требуя никаких корректировок.
- *При изменении приложения изменять базу данных необязательно.* Добавив в документ новые атрибуты или изменив существующие, мы можем развернуть новую версию приложения, и оно сразу же начнет сохранять документы с новой структурой. С другой стороны, для обработки атрибутов, которых не было в ранее сохраненных документах, код, вероятно, придется модифицировать.
- *Изменение схемы не влечет за собой простоя и задержек.* Не требуется блокировать части базы данных, чтобы изменить структуру документов. Правда, как и прежде, приложение, например, придется подправить.
- *Не нужно специально изучать проектирование схем.* Раз схемы отсутствуют, то при создании приложений можно обойтись без освоения целой отрасли знаний. Это означает, что для разработки достаточно специалистов широкого профиля, а объем предварительного планирования сокращается.

Но отсутствию схемы присущи и некоторые недостатки.

- *Система не проверяет структуру документа.* На уровне базы данных нет механизма контроля структуры, и изменения структуры автоматически не распространяются на существующие документы. Это особенно неприятно, когда одной и той же коллекцией пользуются несколько приложений.
- *Не существует определения структуры документа.* Нет в СУБД такого места, куда программист или приложение могли бы заглянуть, чтобы узнать, какой должна быть структура данных. Довольно трудно понять назначение коллекции, изучая отдельные документы, потому что нет никакой гарантии, что структура всех документов одинакова.
- *Отсутствует корректное определение.* Для документных баз данных нет корректного математического определения. Когда

данные хранятся в реляционной базе, часто можно с математической точностью определить оптимальные с точки зрения гибкости и скорости стратегии доступа к данным. В MongoDB ничего подобного такой строгой оптимизации нет, хотя некоторые традиционные методы, например построение индексов, поддерживаются.

Получив некоторое представление о том, как в MongoDB хранятся данные, приступим к работе с этой базой.

8.2.3. Начало работы с MongoDB

Чтобы освоиться с MongoDB, установим ее, а затем поработаем с коллекциями и документами, используя оболочку. Скачайте дистрибутив MongoDB с сайта <http://www.mongodb.org/downloads>, выполните процедуру установки и запустите на сервере процесс `mongod`. Процедура запуска зависит от ОС, детали можно найти в документации (<http://docs.mongodb.org/manual/tutorial/manage-mongodb-processes/>). Запустив СУБД, откройте окно терминала и запустите оболочку командой `mongo` (`mongo.exe` в Windows). Вы увидите примерно такую картину:

```
MongoDB shell version: 2.4.3
connecting to: test
>
```

Важно понимать, что при взаимодействии с MongoDB ни базы данных, ни коллекции вручную не создаются; это делается автоматически по мере необходимости. Чтобы «создать» новую базу данных, выполните команду, в которой эта база используется. Чтобы «создать» коллекцию, вставьте в эту коллекцию документ. При упоминании несуществующей коллекции в запросе ошибки не произойдет; система будет вести себя так, будто коллекция существует, но не создаст ее, пока вы не вставите документ. В табл. 8.1 показаны некоторые типичные операции. Рекомендуем опробовать их по порядку, указывая в качестве *database_name* «`spr`».

Разумеется, функциональность MongoDB отнюдь не ограничивается командами, показанными в таблице. Есть методы для сортировки, для возврата части существующих полей, для вставки или обновления (`upsert`) документов, для инкремента и других способов модификации атрибутов, для манипулирования массивами, для построения индекса и многое, многое другое. Если вы хотите подробно изучить все возможности MongoDB, рекомендуем книгу

Таблица 8.1. Основные команды оболочки MongoDB

Команда	Описание
show dbs	Показать список всех баз данных в экземпляре MongoDB
use database_name	Сделать текущей базу данных с именем <i>database_name</i> . Если такой базы данных еще нет, она будет создана при вставке первого документа в любую коллекцию в этой базе
db	Текущая база данных
help	Получить справку по системе в целом. Команда <code>db.help()</code> дает справку по методам объекта <code>db</code>
db.getCollectionNames()	Показать список всех коллекций в текущей базе данных
db.collection_name	Коллекция в текущей базе данных
db.collection_name.insert({ 'name': 'Josh Powell' })	Вставить документ, содержащий поле <i>name</i> со значением «Josh Powell», в коллекцию с именем <i>collection_name</i>
db.collection_name.find()	Вернуть все документы, хранящиеся в коллекции с именем <i>collection_name</i>
db.collection_name.find({ 'name': 'Josh Powell' })	Вернуть все документы из коллекции <i>collection_name</i> , в которых есть поле <i>name</i> со значением «Josh Powell»
db.collection_name.update({ 'name': 'Josh Powell' }, { 'name': 'Mr. Joshua C. Powell' })	Найти все документы, в которых значение поля <i>name</i> равно «Josh Powell», и заменить в них значение на { 'name': 'Mr. Joshua C. Powell' }
db.collection_name.update({ 'name': 'Mr. Joshua C. Powell' }, { \$set: { 'job': 'Author' } })	Найти все документы, в которых значение поля <i>name</i> равно { 'name': 'Mr. Joshua C. Powell' }, и добавить или изменить в них поля, указанные в атрибуте <i>\$set</i>
db.collection_name.remove({ 'name': 'Mr. Joshua C. Powell' })	Удалить из коллекции <i>collection_name</i> все документы, в которых значение поля <i>name</i> равно «Mr. Joshua C. Powell»
exit	Выйти из оболочки MongoDB

«MongoDB in Action» (Manning 2011)¹, онлайн-руководство по MongoDB (<http://docs.mongodb.org/manual/>) или книгу «Little MongoDB Book» (<http://openmymind.net/mongodb.pdf>). Ну а теперь, изучив простейшие команды MongoDB, давайте подключим базу данных к нашему приложению. Но сначала нужно подготовить файлы проекта.

¹ Имеется русский перевод: Бэнкер К. MongoDB в действии. – М.: ДМК Пресс, 2012. – Прим. перев.

8.3. Драйвер MongoDB

Чтобы эффективно взаимодействовать с MongoDB, необходим драйвер для языка, на котором написано приложение. Без драйвера единственным способом взаимодействия остается оболочка. Написано немало драйверов MongoDB для самых разных языков, в том числе для JavaScript в Node.js. Хороший драйвер берет на себя многие низкоуровневые задачи общения с базой данных, не затрудняя ими разработчика. Например, речь может идти о восстановлении соединения с базой в случае его разрыва, об управлении соединениями с наборами реплик, организации пула буферов и о поддержке курсоров.

8.3.1. Подготовка файлов проекта

В этой главе мы продолжим работу, начатую в главе 7. Скопируйте все файлы, созданные в главе 7, в новый каталог «chapter_8». В листинге 8.1 показана текущая структура каталога. Файлы и каталоги, подлежащие удалению, выделены **полужирным** шрифтом.

Листинг 8.1 ❖ После копирования файлов из главы 7

```
chapter_8
|-- webapp
|   |-- app.js
|   |-- js
|   |   |-- data.js
|   |   |-- node_modules
|   |   |-- package.json
|   |   |-- public
|   |   |   |-- css/
|   |   |   |-- js/
|   |   |   |-- spa.html
|   |   |-- routes.js
|   |   |-- socket.html
|   |   |-- socket.js
```

Удалите каталог js, а также файлы socket.html и socket.js. Каталог node_modules тоже нужно удалить, потому что во время установки модуля он будет создан заново. Измененная структура каталога показана в листинге 8.2.

Листинг 8.2 ❖ После удаления ненужных файлов и каталогов

```
chapter_8
|-- webapp
|   |-- app.js
|   |-- package.json
|   |-- public
```

```
| |-- css/  
| |-- js/  
| '-- spa.html  
|-- routes.js
```

Приведя в порядок каталог проекта, мы готовы подключить MongoDB к нашему приложению. Первым делом установим драйвер MongoDB.

8.3.2. Установка и подключение MongoDB

Мы считаем, что драйвер MongoDB – хорошее решение для многих приложений. Он простой, быстрый и понятный. Если нужно больше возможностей, то можно подумать об использовании *средства объектно-документного отображения* (Object Document Mapper – ODM). ODM – это аналог *средства объектно-реляционного отображения* (Object Relational Mapper – ORM), которое часто применяется при работе с реляционными базами данных. Существует несколько вариантов: *Mongoskin*, *Mongoose*, *Mongolia* и др.

Для нашего приложения мы возьмем простой драйвер MongoDB, потому что ассоциации и высокоуровневое моделирование данных производятся по большей части на стороне клиента. Нам не нужны имеющиеся в ODM возможности валидации данных, так как для проверки структуры документов мы будем пользоваться универсальным валидатором JSON-схем. Мы приняли такое решение, поскольку валидатор JSON-схем согласован со стандартами и работает как на клиенте, так и на сервере, тогда как ODM-валидаторы пока существуют только для сервера.

Для установки драйвера MongoDB воспользуемся файлом `package.json`. Как и раньше, мы явно указываем основной и дополнительный номера версии и заказываем последние исправления (см. листинг 8.3). Изменения выделены **полужирным** шрифтом.

Листинг 8.3 ❖ Обновленный манифест для `npm install` – `webapp/package.json`

```
{  
  "name" : "SPA",  
  "version" : "0.0.3",  
  "private" : true,  
  "dependencies" : {  
    "express" : "3.2.x",  
    "mongodb" : "1.3.x",  
    "socket.io" : "0.9.x"  
  }  
}
```

Выполнив команду `npm install`, мы установим все перечисленные в манифесте модули, в том числе и драйвер MongoDB. Изменим файл `routes.js` – включим драйвер `mongodb` и установим соединение. Изменения выделены в листинге 8.4 **полужирным** шрифтом.

Листинг 8.4 ❖ Открытие соединения с MongoDB – `webapp/routes.js`

```
/*
 * routes.js - модуль маршрутизации
 */
...
// ---- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  configRoutes,
  mongodb = require( 'mongodb' ),
  mongoServer = new mongodb.Server(
    'localhost',
    mongodb.Connection.DEFAULT_PORT
  ),
  dbHandle = new mongodb.Db(
    'spa', mongoServer, { safe : true }
  );
dbHandle.open( function () {
  console.log( '** Установлено подключение к MongoDB **' );
});
// ---- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
...
```

Включаем драйвер MongoDB.

Настраиваем объект соединения с MongoDB, передавая URL (localhost) и номер порта.

Создаем дескриптор базы данных MongoDB, передавая объект соединения и параметры. Начиная с версии драйвера 1.3.6, параметр `safe` считается нереконмендуемым. При работе с одним сервером MongoDB того же результата можно достичь с помощью настройки `{ w : 1 }`.

Открываем соединение с базой данных. Указываем функцию, которая будет вызвана после установления соединения.

Можно также убрать из серверного приложения базовую аутентификацию, как показано в листинге 8.5.

Листинг 8.5 ❖ Исключение базовой аутентификации из серверного приложения – `webapp/app.js`

```
/*
 * app.js - Express-сервер с маршрутизацией
 */
...
// ----- НАЧАЛО КОНФИГУРАЦИИ СЕРВЕРА -----
app.configure( function () {
  app.use( express.bodyParser() );
  app.use( express.methodOverride() );
  app.use( express.static( __dirname + '/public' ) );
  app.use( app.router );
});
...
```

Строка `app.use(express.basicAuth('user', 'spa'))`; удалена.

Если теперь запустить серверное приложение (командой `node app.js`), то мы увидим следующее:

Express-сервер прослушивает порт 3000 в режиме development
** Установлено подключение к MongoDB MongoDB **

Подключившись к MongoDB, займемся исследованием основных операций создания/чтения/обновления/удаления (CRUD).

8.3.3. Использование методов CRUD в MongoDB

Прежде чем продолжать модификацию приложения, неплохо бы освоиться с методами CRUD в MongoDB. Откройте окно терминала и запустите оболочку MongoDB командой `mongo`. Теперь мы можем *создать* несколько документов в коллекции (методом `insert`), как показано в листинге 8.6. **Полужирным** шрифтом выделен текст, который мы вводим.

Листинг 8.6 ❖ Создание документов в MongoDB

```
> use spa;
switched to db spa
> db.user.insert({
  "name" : "Майк Миковски",
  "is_online" : false,
  "css_map": {"top": 100, "left": 120,
    "background-color": "rgb(136, 255, 136)"
  }
});

> db.user.insert({
  "name" : "Мр. Джошуа К. Пауэлл, скромный гуманитарий",
  "is_online": false,
  "css_map": {"top": 150, "left": 120,
    "background-color": "rgb(136, 255, 136)"
  }
});

> db.user.insert({
  "name": "Здесь должно быть ваше имя",
  "is_online": false,
  "css_map": {"top": 50, "left": 120,
    "background-color": "rgb(136, 255, 136)"
  }
});

> db.user.insert({
  "name": "Злополучный чужак",
  "is_online": false,
  "css_map": {"top": 0, "left": 120,
    "background-color": "rgb(136, 255, 136)"
  }
});
```

Чтобы убедиться, что все документы действительно добавлены, мы можем *прочитать* их (методом `find`), как показано в листинге 8.7. **Полужирным** шрифтом выделен текст, который мы вводим.

Листинг 8.7 ❖ Чтение документов из базы MongoDB

```
> db.user.find()
{ "_id" : ObjectId("5186aae56f0001debc935c33"),
  "name" : "Майк Миковски",
  "is_online" : false,
  "css_map" : {
    "top" : 100, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
},
{ "_id" : ObjectId("5186aaed6f0001debc935c34"),
  "name" : "Мр. Джошуа К. Пауэлл, скромный гуманитарий",
  "is_online" : false,
  "css_map" : {
    "top" : 150, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
},
{ "_id" : ObjectId("5186aaf76f0001debc935c35"),
  "name" : "Здесь должно быть ваше имя",
  "is_online" : false,
  "css_map" : {
    "top" : 50, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
},
{ "_id" : ObjectId("5186aaff6f0001debc935c36"),
  "name" : "Злополучный чужак",
  "is_online" : false,
  "css_map" : {
    "top" : 0, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
}
```

Обратите внимание, что MongoDB автоматически добавляет в любой вставленный документ поле `_id`, содержащее уникальный идентификатор. Гм, в поле `name` одного из документов имя автора написано хотя и правильно (оценка, пожалуй, несколько занижена), но как-то слишком формально. Уберем эту официальность, *обновив* документ (методом `update`), как показано в листинге 8.8. **Полужирным** шрифтом выделен текст, который мы вводим.

Листинг 8.8 ❖ Обновление документов в MongoDB

```
> db.user.update(
  { "_id" : ObjectId("5186aaed6f0001debc935c34") },
  { $set : { "name" : "Джош Пауэлл" } }
);

db.user.find(
  "_id" : ObjectId("5186aaed6f0001debc935c34")
);

{ "_id" : ObjectId("5186aaed6f0001debc935c34"),
  "name" : "Джош Пауэлл",
  "is_online" : false,
  "css_map" : {
    "top" : 150, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
}
```

Нельзя не заметить, что в нашу базу проник *злополучный чужак*. Как и одетый в красную рубашку член десантной группы в «Стартреке», злополучный чужак не должен дожить до конца эпизода. Мы не хотим нарушать традицию, поэтому расправимся с чужаком раз и навсегда и *удалим* документ (методом `remove`), как показано в листинге 8.9. **Полужирным** шрифтом выделен текст, который мы вводим.

Листинг 8.9 ❖ Обновление документов в MongoDB

```
> db.user.remove(
  { "_id" : ObjectId("5186aaff6f0001debc935c36") }
);

> db.user.find()
{ "_id" : ObjectId("5186aae56f0001debc935c33"),
  "name" : "Майк Миковски",
  "is_online" : false,
  "css_map" : {
    "top" : 100, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
},
{ "_id" : ObjectId("5186aaed6f0001debc935c34"),
  "name" : "Джош Пауэлл",
  "is_online" : false,
  "css_map" : {
    "top" : 150, "left" : 120,
    "background-color" : "rgb(136, 255, 136)"
  }
}
{ "_id" : ObjectId("5186aaf76f0001debc935c35"),
```

```

    "name" : "Здесь должно быть ваше имя",
    "is_online" : false,
    "css_map" : {
      "top" : 50, "left" : 120,
      "background-color" : "rgb(136, 255, 136)"
    }
  }
}

```

Мы рассмотрели все операции CRUD на консоли MongoDB. А теперь поддержим их в нашем серверном приложении.

8.3.4. Добавление операций CRUD в серверное приложение

Коль скоро мы работаем с Node.js, взаимодействие с MongoDB будет не таким, как в большинстве других языков, потому что JavaScript-код основан на событиях. Имея в базе данных несколько документов, модифицируем наш маршрутизатор, так чтобы с помощью MongoDB извлекался список объектов пользователей. Изменения в листинге 8.10 выделены **полужирным** шрифтом.

Листинг 8.10 ❖ Модификация маршрутизатора для извлечения списка пользователей – `webapp/routes.js`

```

/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
configRoutes = function ( app, server ) {
  ...
  app.get( '/:obj_type/list', function ( request, response ) {
    dbHandle.collection(
      request.params.obj_type,
      function ( outer_error, collection ) {
        collection.find().toArray(
          function ( inner_error, map_list ) {
            response.send( map_list );
          }
        );
      }
    );
  });
};
...
};

module.exports = { configRoutes : configRoutes };
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
...

```

Используем объект `dbHandle` для выборки коллекции, определяемой параметром `:obj_type` в URL, и передаем функцию обратного вызова.

Находим все документы в коллекции (`dbHandle.collection`) и преобразуем результаты в массив.

Отправляем список JSON-объектов клиенту.

Перед тем как просматривать результаты в браузере, имеет смысл установить расширение или дополнение к браузеру, показывающее JSON-документы в удобочитаемом виде. Мы используем *JSONView 0.0.32* в Chrome и *JSONovich 1.9.5* в Firefox. То и другое можно скачать с сайта расширений для соответствующего браузера.

Для запуска приложения введите команду `node app.js` в окне терминала. Перейдя в браузере по адресу `http://localhost:3000/user/list`, мы увидим представление JSON-документа, показанное на рис. 8.3.

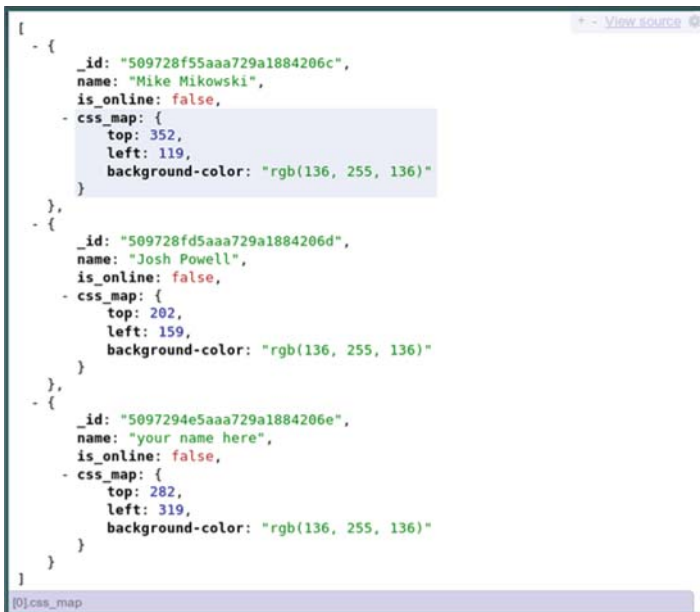


Рис. 8.3 ❖ Ответ, полученный Node.js от MongoDB и переданный клиенту

Теперь можно добавить остальные операции CRUD, как показано в листинге 8.11. Изменения выделены **полужирным** шрифтом.

Листинг 8.11 ❖ Добавление драйвера MongoDB и операций CRUD в маршрутизатор – `routes.js`

```

/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  
```

```

'use strict';
var
  configRoutes,
  mongodb = require( 'mongodb' ),

  mongoServer = new mongodb.Server(
    'localhost',
    mongodb.Connection.DEFAULT_PORT
  ),
  dbHandle = new mongodb.Db(
    'spa', mongoServer, { safe : true }
  ),
  makeMongoId = mongodb.ObjectId;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
configRoutes = function ( app, server ) {
  app.get( '/', function ( request, response ) {
    response.redirect( '/spa.html' );
  });

  app.all( '/:obj_type/*?', function ( request, response, next ) {
    response.contentType( 'json' );
    next();
  });

  app.get( '/:obj_type/list', function ( request, response ) {
    dbHandle.collection(
      request.params.obj_type,
      function ( outer_error, collection ) {
        collection.find().toArray(
          function ( inner_error, map_list ) {
            response.send( map_list );
          }
        );
      }
    );
  });

  app.post( '/:obj_type/create', function ( request, response ) {
    dbHandle.collection(
      request.params.obj_type,
      function ( outer_error, collection ) {
        var
          options_map = { safe: true },
          obj_map = request.body;

        collection.insert(
          obj_map,
          options_map,
          function ( inner_error, result_map ) {
            response.send( result_map );
          }
        );
      }
    );
  });
}

```

Копируем функцию `ObjectId` в переменную `makeMongoId` в области видимости модуля – просто для удобства. Отметим, что соединение с базой данных теперь открывается в конце модуля.

Добавляем возможность получения списка пользователей. Это уже было показано выше в данном разделе. Не включайте дважды.

Вставляем документ в MongoDB. Параметр `safe` означает, что функцию не следует вызывать раньше, чем документ будет успешно вставлен в базу; в противном случае функция вызывается немедленно, не дожидаясь уведомления об успехе операции. Чему отдать предпочтение: скорости или безопасности, – решать вам. В данном случае можно было бы и не задавать этот параметр, потому что мы установили его по умолчанию, когда настраивали соединение с базой. Кроме того, выше было отмечено, что теперь появился новый параметр `w`, а параметр `safe` объявлен нерекомендуемым.

```

    }
  );
}
);
});

app.get( '/:obj_type/read/:id', function ( request, response ) {
  var find_map = { _id: makeMongoId( request.params.id ) };
  dbHandle.collection(
    request.params.obj_type,
    function ( outer_error, collection ) {
      collection.findOne( ←
        find_map,
        function ( inner_error, result_map ) {
          response.send( result_map );
        }
      );
    }
  );
});

app.post( '/:obj_type/update/:id', function ( request, response ) {
  var
    find_map = { _id: makeMongoId( request.params.id ) },
    obj_map = request.body;
  dbHandle.collection(
    request.params.obj_type,
    function ( outer_error, collection ) {
      var
        sort_order = [],
        options_map = {
          'new': true, upsert: false, safe: true
        };

      collection.findAndModify( ←
        find_map,
        sort_order,
        obj_map,
        options_map,
        function ( inner_error, updated_map ) {
          response.send( updated_map );
        }
      );
    }
  );
});

app.get( '/:obj_type/delete/:id', function ( request, response ) {
  var find_map = { _id: makeMongoId( request.params.id ) };
  dbHandle.collection(

```

С помощью метода `findOne`, предоставляемого драйвером MongoDB для Node.js, находим и возвращаем первый документ, удовлетворяющий параметрам поиска. Поскольку может существовать только один объект с данным идентификатором, то больше одного документа искать не имеет смысла.

Используем метод `findOne`, предоставляемый драйвером MongoDB для Node.js. Он находит все документы, удовлетворяющие критериям поиска, и заменяет их объектом, указанным в параметре `obj_map`. Да, мы знаем, что имя неудачное, но ведь не мы писали драйвер MongoDB.

```

request.params.obj_type,
function ( outer_error, collection ) {
    var options_map = { safe: true, single: true };

    collection.remove( ← С помощью метода remove удаляем все документы,
                        find_map, удовлетворяющие критериям, заданным в объекте
                        options_map, find_map. Поскольку мы указали параметр single:
                        function ( inner_error, delete_count ) { true, то будет удалено не более одного документа.
                            response.send({ delete_count: delete_count });
                        }
                    );
}
});
});
};

```

Добавляем секцию инициализации модуля.

```

module.exports = { configRoutes : configRoutes };
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

// ----- НАЧАЛО ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----
dbHandle.open( function () {
    console.log( '** Установлено подключение к MongoDB **' );
});
// ----- КОНЕЦ ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----

```

Теперь у нас имеются операции CRUD, которые клиент запрашивает у сервера Node.js, тот обращается к MongoDB и возвращает результаты обратно клиенту. Следующая задача – проверить правильность полученных от клиента данных.

8.4. Валидация данных, поступивших от клиента

В MongoDB нет механизма, который позволил бы указать, что можно добавлять в коллекцию, а чего нельзя. Мы должны сами проверять клиентские данные перед сохранением их в базе. Мы хотим, чтобы обмен данными был организован, как показано на рис. 8.4.

Первый шаг – определить, какие типы объектов допустимы.

8.4.1. Проверка типа объекта

На данный момент мы принимаем любой маршрут и передаем объекты MongoDB, даже не проверив, допустим ли их тип. Например, POST-запрос для создания лошади будет работать. Ниже приведен пример с использованием `wget`. **Полужирным** шрифтом выделен текст, который мы вводим.

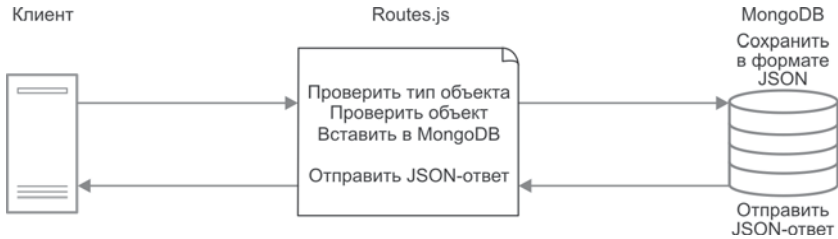


Рис. 8.4 ❖ Валидация клиентских данных – программная обработка

```

# Создаем новую коллекцию лошадей в MongoDB
wget http://localhost:3000/horse/create \
--header='content-type: application/json' \
--post-data='{ "css_map": { "color": "#ddd"}, "name": "Ed"}' \
-O -

# Добавляем еще одну лошадь
wget http://localhost:3000/horse/create \
--header='content-type: application/json' \
--post-data='{ "css_map": { "color": "#2e0"}, "name": "Winney"}' \
-O -

# Проверяем содержимое конюшни
wget http://localhost:3000/horse/list -O -
[ {
  "css_map": {
    "color": "#ddd"
  },
  "name": "Ed",
  "_id": "51886ac7e7f0be8d20000001"
},
{
  "css_map": {
    "color": "#2e0"
  },
  "name": "Winney",
  "_id": "51886adae7f0be8d20000002"
}]

```

Ситуация даже хуже, чем может показаться. MongoDB не только сохраняет документ, но и *создает новую коллекцию* (как в примере выше), на что уходит немало ресурсов. Такой программе не место в производственном режиме, потому что даже самый тупой хакер-дилетант за несколько минут поставит сервер (или несколько серверов) на колени, напустив на него скрипт, который создает тысячи коллекций MongoDB¹. Мы должны разрешать доступ только к объектам допустимых типов, как показано на рис. 8.5.

¹ На моей 64-разрядной машине для разработки *почти пустая* коллекция занимает 64 Мб на диске.

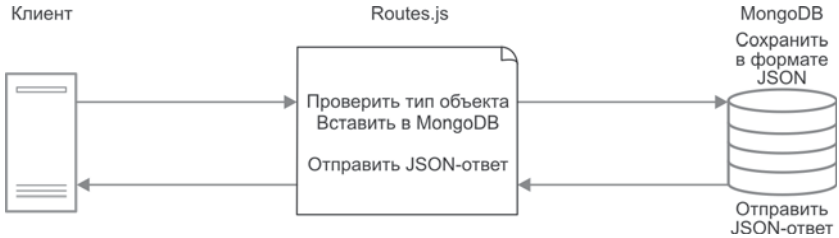


Рис. 8.5 ❖ Валидация типа объекта

Реализовать это совсем несложно. Создадим хэш допустимых типов объектов и будем проверять по нему в маршрутизаторе. В листинге 8.12 показано, как нужно модифицировать файл routes.js. Изменения выделены **полужирным** шрифтом.

Листинг 8.12 ❖ Валидация входящих маршрутов – routes.js

```

/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
...
makeMongoId = mongodb.ObjectId,
objTypeMap = { 'user': {} };
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
configRoutes = function ( app, server ) {
  app.get( '/', function ( request, response ) {
    response.redirect( '/spa.html' );
  });
  app.all( '('/:obj_type/*?', function ( request, response, next ) {
    response.contentType( 'json' );
    if ( objTypeMap[ request.params.obj_type ] ) {
      next();
    }
    else {
      response.send({ error_msg : request.params.obj_type
        + ' is not a valid object type'
      });
    }
  });
};
...

```

Объявляем и инициализируем хэш допустимых типов объектов.

Если тип объекта (:obj_type) присутствует в хэше типов, переходим к следующему обработчику маршрута.

Если тип объекта (:obj_type) отсутствует в хэше типов, отправляем JSON-ответ с сообщением о том, что маршрут недопустим.

Но мы не хотим ограничиваться проверкой допустимости типа объекта. Нужно еще убедиться, что клиентские данные имеют ожидаемую структуру. Займемся этим.

8.4.2. Проверка объекта

Клиентский браузер посылает серверу JSON-документ с представлением объекта. Многие читатели, несомненно, знают, что формат JSON вытеснил XML из многих API, применяемых в веб, потому что он компактнее и проще для обработки.

Но у XML есть одна замечательная особенность: возможность задать *определение типа документа* (*Document Type Definition – DTD*), в котором описывается, какое содержимое допустимо. Для JSON имеется аналогичный, хотя и не такой зрелый, стандарт, который называется *JSON-схема*. Для валидации JSON-схемы предназначена программа *JSV*. Она работает как на клиенте, так и на сервере, поэтому можно обойтись без написания и поддержки двух разных (и неизбежно чуть-чуть несовместимых) библиотек валидации. Ниже перечислено, что нужно сделать для валидации наших объектов:

- установить модуль *JSV* для *Node.js*;
- создать JSON-схему;
- загрузить JSON-схемы;
- написать функцию валидации;
- проверить правильность входящих данных.

Первым делом установим *JSV*.

Установка модуля JSV для Node.js

Изменим файл *package.json*, как показано в листинге 8.13, включив модуль *JSV 4.0.2*.

Листинг 8.13 ❖ Модификация манифеста с целью включения модуля *JSV – webapp/package.json*

```
{ "name"      : "SPA",  
  "version"   : "0.0.3",  
  "private"   : true,  
  "dependencies" : {  
    "express"  : "3.2.x",  
    "mongodb"  : "1.3.x",  
    "socket.io" : "0.9.x",  
    "JSV"      : "4.0.x"  
  }  
}
```

После запуска `npm install` модуль *JSV* будет установлен.

Создание JSON-схемы

Прежде чем валидировать объект пользователя, необходимо решить, какие свойства допустимы и какие значения они могут при-

нимать. JSON-схема предлагает изящный и стандартизованный механизм описания таких ограничений. Код схемы приведен в листинге 8.14. Внимательно читайте аннотации, потому что в них объясняется смысл ограничений.

Листинг 8.14 ❖ Схема объекта пользователя – webapp/user.json

Этот объект представляет схему объекта ("type" : "object"). С его помощью можно описать ограничения на целочисленные, строковые, булевы значения и массивы.

```
{ "type" : "object",
  "additionalProperties" : false,
  "properties" : {
    "_id" : {
      "type" : "string",
      "minLength" : 25,
      "maxLength" : 25
    },
    "name" : {
      "type" : "string",
      "minLength" : 2,
      "maxLength" : 127
    },
    "is_online" : {
      "type" : "boolean"
    },
    "css_map" : {
      "type" : "object",
      "additionalProperties" : false,
      "properties" : {
        "background-color" : {
          "required" : true,
          "type" : "string",
          "minLength" : 0,
          "maxLength" : 25
        },
        "top" : {
          "required" : true,
          "type" : "integer"
        },
        "left" : {
          "required" : true,
          "type" : "integer"
        }
      }
    }
  }
}
```

Тип object может принимать или отвергать свойства, которые не объявлены явно. Если значение атрибута additionalProperties равно false, то валидатор не пропустит необъявленные свойства. Почти всегда правильно устанавливать именно такой режим.

В атрибуте properties задает хэш свойств для схемы объекта, индексированный по имени свойства.

Свойство _id – строка длиной ровно 25 знаков ("minLength" : 25, "maxLength" : 25).

Свойство name аналогично _id в том смысле, что его длина может быть переменной.

Свойство is_online должно принимать значение true или false.

Свойство css_map должно быть объектом и не допускает наличия необъявленных свойств.

Свойство background-color объекта css_map обязательно, является строковым, его длина не должна превышать 25 знаков.

Свойство top объекта css_map обязательно и должно быть целым числом.

Свойство left объекта css_map обязательно и должно быть целым числом.

Вероятно, вы обратили внимание, что мы определили схему, которая налагает ограничения на объект и на объект *внутри* этого объекта.

Это показывает, что JSON-схема может быть рекурсивной, причем глубина рекурсии произвольна. JSON-схемы, как и XML-схемы, могут расширять другие схемы. Если вы хотите подробнее познакомиться с JSON-схемами, загляните на официальный сайт json-schema.org. Теперь мы можем загрузить эту схему и с ее помощью проверить, что полученный объект пользователя содержит только допустимые данные.

Загрузка JSON-схем

Будем загружать схемы в память на этапе запуска сервера. Это позволит избежать дорогостоящего поиска файлов во время работы серверного приложения. Загрузить можно только один документ со схемой для каждого типа объекта, определенного в хэше `objTypeMap`. Код показан в листинге 8.15, а изменения выделены **полужирным шрифтом**¹.

Листинг 8.15 ❖ Загрузка схемы в маршрутизаторе – `webapp/routes.js`

```
/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  loadSchema, configRoutes,
  mongodb = require( 'mongodb' ),
  fsHandle = require( 'fs' ), ← Включаем модуль для работы с файловой системой.

  mongoServer = new mongodb.Server(
    'localhost',
    mongodb.Connection.DEFAULT_PORT
  ),
  dbHandle = new mongodb.Db(
    'spa', mongoServer, { safe : true }
  ),

  makeMongoId = mongodb.ObjectID,
  objTypeMap = { 'user': {} };
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
loadSchema = function ( schema_name, schema_path ) { ←
  fsHandle.readFile( schema_path, 'utf8', function ( err, data ) {
```

Служебный метод `loadSchema`, который читает файл и сохраняет его в хэше типов объектов (`objTypeMap`).

¹ В Windows необходимо заменить каждый символ `/` в путях к файлам двумя символами `\\`.

```

    objTypeMap[ schema_name ] = JSON.parse( data );
  });
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
...
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

// ----- НАЧАЛО ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----
dbHandle.open( function () {
  console.log( '** Установлено подключение к MongoDB **' );
});

// загружаем схемы в память (objTypeMap)
(function () {
  var schema_name, schema_path;
  for ( schema_name in objTypeMap ) {
    if ( objTypeMap.hasOwnProperty( schema_name ) ) {
      schema_path = __dirname + '/' + schema_name + '.json';
      loadSchema( schema_name, schema_path );
    }
  }
})();
// ----- КОНЕЦ ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----

```

Для каждого типа объектов, определенного в objTypeMap, читаем файл. В данном случае есть только один тип объектов: user.

Разбираем содержимое файла, сохраняем его в JSON-объекте, а ссылку на этот объект — в хэше типов. Мы используем внешнюю функцию (loadSchema), потому что определять функцию внутри цикла не рекомендуется, и JSLint сочтет это ошибкой.

Загрузив схемы, мы можем написать функцию валидации.

Реализация функции валидации

После того как JSON-схема `user` загружена, мы можем сопоставить с ней данные, поступившие от клиента. В листинге 8.16 приведена простая функция, которая именно это и делает. Изменения выделены **полужирным** шрифтом.

Листинг 8.16 ❖ Добавление функции для валидации документов — `webapp/routes.js`

```

/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  loadSchema, checkSchema, configRoutes,
  mongodb = require( 'mongodb' ),
  fsHandle = require( 'fs' ),

```

```

JSV = require( 'JSV' ).JSV, ← Включаем модуль JSV.

mongoServer = new mongodb.Server(
  'localhost',
  mongodb.Connection.DEFAULT_PORT
),
dbHandle = new mongodb.Db(
  'spa', mongoServer, { safe : true }
),
validator = JSV.createEnvironment(), ← Создаем окружение для валидатора JSV.

makeMongoId = mongodb.ObjectID,
objTypeMap = { 'user': {} };
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
loadSchema = function ( schema_name, schema_path ) {
  fsHandle.readFile( schema_path, 'utf8', function ( err, data ) {
    objTypeMap[ schema_name ] = JSON.parse( data );
  });
};
// Валидатор принимает три аргумента: сам объект (obj_map),
// имя схемы, относительно которой нужно валидировать данные
// (obj_type), и функцию обратного вызова callback.

checkSchema = function ( obj_type, obj_map, callback ) { ←
  var
    schema_map = objTypeMap[ obj_type ],
    report_map = validator.validate( obj_map, schema_map );

  callback( report_map.errors ); ←
};
// После завершения валидации вызываем функцию
// callback, передавая ей список ошибок. Если
// этот список пуст, то объект является допустимым.
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
...

```

Имея средства для загрузки JSON-схемы и функцию валидации, мы можем проверить поступающие от клиента данные.

Проверка правильности входящих данных

Теперь завершим построение механизма валидации. Необходимо лишь подправить маршруты для приема клиентских данных – create и update, – включив в них валидацию. В обоих случаях затребованное действие будет выполняться, только если список ошибок пуст, в противном случае мы вернем сообщение об ошибке. Изменения показаны в листинге 8.17 и выделены **полужирным** шрифтом.

Листинг 8.17 ❖ Маршруты create и update с валидацией – webapp/routes.js

```

/*
 * routes.js - модуль маршрутизации
 */

```

Вызываем определенную выше функцию валидации (`checkSchema`), указывая тип объекта, хэш объекта и функцию обратного вызова.

```

    obj_type,
    function ( outer_error, collection ) {
        var
            sort_order = [],
            options_map = {
                'new' : true, upsert: false, safe: true
            };

        collection.findAndModify(
            find_map,
            sort_order,
            obj_map,
            options_map,
            function ( inner_error, updated_map ) {
                response.send( updated_map );
            }
        );
    }
};
}
else {
    response.send({ ← Если список ошибок не пуст, возвращаем
                     сообщение об ошибке.
                     error_msg : 'Input document not valid',
                     error_list : error_list
    });
}
}
);
});
...
};

module.exports = { configRoutes : configRoutes };
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
...

```

Разобравшись с валидацией, посмотрим, что мы сделали. Для начала убедимся, что все наши модули проходят JSLint (`jslint user.json app.js routes.js`), затем запустим приложение (`node app.js`). Далее применим свои знания о работе с `wget`, чтобы отправить POST-запрос с плохими и хорошими данными, как показано в листинге 8.18. **Полужирным** шрифтом выделен текст, который мы вводим.

Листинг 8.18 ❖ Отправка POST-запросов с плохими и хорошими данными с помощью `wget`

```

# Попробуем некорректные данные
wget http://localhost:3000/user/create \
  --header='content-type: application/json' \
  --post-data='{ "name": "Betty",

```

```

    "css_map": {"background-color": "#ddd",
    "top" : 22 }
  }' -0 -

--2013-06-07 22:20:17-- http://localhost:3000/user/create
Resolving localhost (localhost)... 127.0.0.1
Connecting to localhost (localhost)|127.0.0.1|:3000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 354 [application/json]
Saving to: â€”STDOUTâ€”

...
{
  "error_msg": "Input document not valid",
  "error_list": [
    {
      "uri": "urn:uuid:8c05b92a...",
      "schemaUri": "urn:uuid:.../properties/css_map/properties/left",
      "attribute": "required",
      "message": "Property is required",
      "details": true
    }
  ]
}
# Ой, пропустили свойство "left". Исправим это.
wget http://localhost:3000/user/create \
  --header='content-type: application/json' \
  --post-data='{ "name": "Betty",
    "css_map": {"background-color": "#ddd",
    "top" : 22, "left" : 500 }
  }' -0 -

--2013-05-07 22:24:02-- http://localhost:3000/user/create
Resolving localhost (localhost)... 127.0.0.1
Connecting to localhost (localhost)|127.0.0.1|:3000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 163 [application/json]
Saving to: â€”STDOUTâ€”

...
{
  "name": "Betty",
  "css_map": {
    "background-color": "#ddd",
    "top": 22,
    "left": 500
  },
  "_id": "5189e172ac5a4c5c68000001"
}
# Получилось!
```

Обновление данных о пользователе с помощью **wget** оставляем читателю в качестве упражнения.

В следующем разделе мы перенесем всю функциональность CRUD в отдельный модуль. И тем самым получим более чистый, более простой для понимания и удобный для сопровождения код.

8.5. Создание отдельного модуля CRUD

В данный момент вся логика операций CRUD и маршрутизации находится в файле `routes.js`, как показано на рис. 8.6.

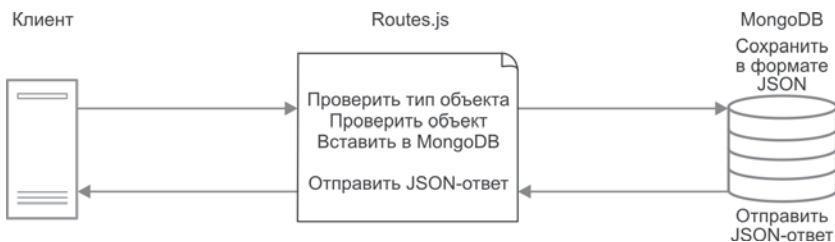


Рис. 8.6 ❖ Программная обработка

У нас есть сервер, который принимает запросы от клиентов, производит валидацию данных и сохраняет их в базе. Единственный способ осуществить валидацию и сохранение данных – послать HTTP-запрос с указанием правильного маршрута. Если бы этим функции приложения и исчерпывались, то можно было бы поставить точку и не заниматься дальнейшим абстрагированием. Однако наше SPA должно еще создавать и модифицировать объекты через веб-сокеты. Поэтому мы создадим модуль CRUD, в который поместим всю логику валидации документов и работы с базой данных. Тогда маршрутизатор будет использовать этот модуль всякий раз, как потребуется выполнить операцию CRUD.

Но сначала мы хотим объяснить, почему до сих пор тянули с созданием модуля CRUD. Мы хотим, чтобы наш код был ясным и простым настолько, насколько возможно, но не проще. Если некоторая операция встречается в коде всего один раз, то мы предпочитаем реализовать ее по месту или оформить в виде локальной функции. Но как только выясняется, что операция встречается два или более раз, мы ее абстрагируем. Это, скорее всего, не уменьшит времени написания первоначального кода, но почти наверняка позволит сэкономить на сопровождении, поскольку логика оказывается в одном месте, и мы избегаем тонких ошибок, связанных с небольшими различиями в реализации. Разумеется, нужно решить, насколько далеко заходить в применении

такой философии. Например, мы считаем, что абстрагировать циклы `for` в общем случае не стоит, хотя в JavaScript это возможно.

После того как мы перенесем установление соединения с MongoDB и валидацию в отдельный модуль `CRUD`, на маршрутизатор больше не будет возлагаться реализация хранения данных, и он станет больше похож на контроллер в том смысле, что поручает выполнение запросов другим модулям, а не выполняет их сам. Эта организация показана на рис. 8.7.

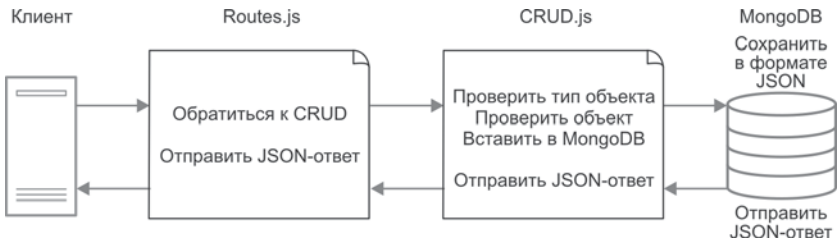


Рис. 8.7 ❖ Программная обработка на сервере

Первый шаг в создании модуля `CRUD` – подготовка структуры файлов.

8.5.1. Подготовка структуры файлов

Структура файлов оставалась неизменной с начала этой главы. Но теперь, когда дело дошло до добавления еще одного модуля, ее стоит немного пересмотреть. Сейчас структура выглядит, как показано на рис. 8.19:

Листинг 8.19 ❖ Текущая структура файлов

```
chapter_8
|-- webapp
|   |-- app.js
|   |-- node_modules/
|   |-- package.json
|   |-- public
|   |   |-- css/
|   |   |-- js/
|   |   |-- spa.html
|   |-- user.json
|   |-- routes.js
```

Мы хотим хранить наши модули в отдельном каталоге `lib`. Так мы наведем порядок в каталоге `webapp` и вынесем свои модули из каталога `node_modules`. В каталоге `node_modules` должны находиться только внешние модули, добавленные командой `npm install`, чтобы

его можно было удалить и заново создать, не затрагивая модулей, написанных нами. В листинге 8.20 показано, какой мы хотим видеть структуру файлов.

Листинг 8.20 ❖ Новая – просветленная – структура файлов

```
chapter_8
|-- webapp
|   |-- app.js
|   |-- lib
|       |-- crud.js
|       |-- routes.js
|       |-- user.json
|-- node_modules/
|-- package.json
|-- public
|   |-- css/
|   |-- js/
|   |-- spa.html
```

Первым шагом на пути к просветлению будет перенос файла маршрутизатора в каталог `webapp/lib`. После этого мы должны будем изменить серверное приложение, указав новый путь, как показано **полужирным** шрифтом в листинге 8.21.

Листинг 8.21 ❖ Указание нового пути к файлу `routes.js` в `app.js` – `webapp/app.js`

```
/*
 * app.js - Express-сервер с маршрутизацией
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  http    = require( 'http' ),
  express = require( 'express' ),
  routes  = require( './lib/routes' ),
  app     = express(),
  server  = http.createServer( app );
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
...
```

Следующий шаг – включение модуля `CRUD` в модуль `routes`, как показано в листинге 8.22.

Листинг 8.22 ❖ Включение модуля `CRUD` в модуль `routes` – `webapp/lib/routes.js`

```
/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
```

```
'use strict';
var
  loadSchema, checkSchema, configRoutes,
  mongodb = require( 'mongodb' ),
  fsHandle = require( 'fs' ),
  JSV      = require( 'JSV' ).JSV,
  crud     = require( './crud' ),
  ...
```

Теперь можно создать модуль CRUD и прикинуть его API. Мы воспользуемся объектом `module.exports` для предоставления общего доступа к методам CRUD, как показано в листинге 8.23.

Листинг 8.23 ❖ Создание модуля CRUD – `webapp/lib/crud.js`

```
/*
 * crud.js – модуль, предоставляющий операции CRUD с базой данных
 */
/*jslint      node    : true, continue : true,
  devel      : true, indent : 2,  maxerr  : 50,
  newcap     : true, nomen  : true, plusplus : true,
  regexp     : true, sloppy : true, vars   : false,
  white      : true
*/
/*global */
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  checkType, constructObj, readObj,
  updateObj, destroyObj;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
checkType = function () {};
constructObj = function () {};
readObj = function () {};
updateObj = function () {};
destroyObj = function () {};

module.exports = {
  makeMongoId : null,
  checkType   : checkType,
  construct    : constructObj, ← Называем наш метод construct, потому что create –
                                корневой метод в прототипе объекта JavaScript Object.
  read         : readObj,
  update       : updateObj,
  destroy      : destroyObj ← Называем наш метод destroy, потому что delete –
                              ключевое слово языка JavaScript.
};
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

// ----- НАЧАЛО ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----
console.log( '** Модуль CRUD загружен **' );
// ----- КОНЕЦ ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----
```

Команда `node app.js` должна запустить наш сервер без ошибок и напечатать сообщения:

```
** Модуль CRUD загружен **
Express-сервер прослушивает порт 3000 в режиме development
** Установлено подключение к MongoDB **
```

Отметим, что мы добавили два метода, помимо стандартных операций CRUD. Метод `makeMongoId` служит для создания идентификатора MongoDB, а методом `checkType` мы собираемся пользоваться для проверки допустимости типа объекта. Организовав файлы, мы можем перенести логику CRUD в подходящий модуль.

8.5.2. Перенос операций CRUD в отдельный модуль

Для завершения модуля CRUD скопируем наши методы из модуля `routes` и заменим специфичные для HTTP параметры более общими. Детали опустим, потому что считаем это преобразование очевидным. Окончательный вид модуля показан в листинге 8.24. Обращайте внимание на аннотации, потому что в них имеются дополнительные пояснения.

Листинг 8.24 ❖ Перенос логики в модуль CRUD – `webapp/lib/crud.js`

```
/*
 * crud.js – модуль, предоставляющий операции CRUD с базой данных
 */
/*jslint      node    : true, continue : true,
   devel     : true, indent : 2,  maxerr  : 50,
   newcap    : true, nomen  : true, plusplus : true,
   regexp    : true, sloppy : true, vars   : false,
   white     : true
 */
/*global */

// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  loadSchema, checkSchema, clearIsOnline,
  checkType, constructObj, readObj,
  updateObj, destroyObj,

  mongodb = require( 'mongodb' ),
  fsHandle = require( 'fs' ),
  JSV      = require( 'JSV' ).JSV,

  mongoServer = new mongodb.Server(
    'localhost',
    mongodb.Connection.DEFAULT_PORT
```

Включаем библиотеки, необходимые для операций CRUD.

Объявляем переменные, относящиеся к соединению с базой данных (`mongodb` и `dbHandle`), и валидатор JSON-схемы.

```

    ),
    dbHandle = new mongodb.Db(
        'spa', mongoServer, { safe : true }
    ),
    validator = JSV.createEnvironment(),

    objTypeMap = { 'user' : {} };
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
loadSchema = function ( schema_name, schema_path ) {
    fsHandle.readFile( schema_path, 'utf8', function ( err, data ) {
        objTypeMap[ schema_name ] = JSON.parse( data );
    });
};

checkSchema = function ( obj_type, obj_map, callback ) {
    var
        schema_map = objTypeMap[ obj_type ],
        report_map = validator.validate( obj_map, schema_map );

    callback( report_map.errors );
};

clearIsOnline = function () {
    updateObj(
        'user',
        { is_online : true },
        { is_online : false },
        function ( response_map ) {
            console.log( 'All users set to offline', response_map );
        }
    );
};

// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
checkType = function ( obj_type ) {
    if ( ! objTypeMap[ obj_type ] ) {
        return ( { error_msg : 'Object type "' + obj_type
            + '" is not supported.'
        } );
    }
    return null;
};

constructObj = function ( obj_type, obj_map, callback ) {
    var type_check_map = checkType( obj_type );
    if ( type_check_map ) {
        callback( type_check_map );
    }
};

```

Объявляем хэш допустимых типов объектов (objTypeMap).

Добавляем методы загрузки и проверки схемы.

Метод clearIsOnline вызывается после подключения к MongoDB. Он гарантирует, что сразу после запуска сервера все пользователи помечены как не находящиеся в онлайн.

Метод проверяет, является ли тип объекта, например user или horse, допустимым. В настоящее время допустим только тип user.

Переносим логику создания объекта (construct) из webapp/lib/routes.js в этот модуль. Логика, по существу, та же самая, но чуть более общая. Это упрощает вызов как из модуля routes, так и из других модулей.

Добавляем проверку допустимости типа объекта. Если тип недопустим, возвращаем JSON-документ с описанием ошибки.

```

    return;
  }
  checkSchema(
    obj_type, obj_map,
    function ( error_list ) {
      if ( error_list.length === 0 ) {
        dbHandle.collection(
          obj_type,
          function ( outer_error, collection ) {
            var options_map = { safe: true };

            collection.insert(
              obj_map,
              options_map,
              function ( inner_error, result_map ) {
                callback( result_map );
              }
            );
          }
        );
      }
      else {
        callback({
          error_msg : 'Input document not valid',
          error_list : error_list
        });
      }
    }
  );
};

Метод read перенесен из webapp/lib/routes.js.
Логика сделана чуть более общей.
readObj = function ( obj_type, find_map, fields_map, callback ) {
  var type_check_map = checkType( obj_type );
  if ( type_check_map ) {
    callback( type_check_map );
    return;
  }

  dbHandle.collection(
    obj_type,
    function ( outer_error, collection ) {
      collection.find( find_map, fields_map ).toArray(
        function ( inner_error, map_list ) {
          callback( map_list );
        }
      );
    }
  );
};
};

```

Проверяем допустимость типа объекта. Если тип недопустим, возвращаем JSON-документ с описанием ошибки.

```

updateObj = function ( obj_type, find_map, set_map, callback ) {
  var type_check_map = checkType( obj_type );
  if ( type_check_map ) {
    callback( type_check_map );
    return;
  }
}

checkSchema(
  obj_type, set_map,
  function ( error_list ) {
    if ( error_list.length === 0 ) {
      dbHandle.collection(
        obj_type,
        function ( outer_error, collection ) {
          collection.update(
            find_map,
            { $set : set_map },
            { safe : true, multi : true, upsert : false },
            function ( inner_error, update_count ) {
              callback({ update_count : update_count });
            }
          );
        }
      );
    }
    else {
      callback({
        error_msg : 'Input document not valid',
        error_list : error_list
      });
    }
  }
);

destroyObj = function ( obj_type, find_map, callback ) {
  var type_check_map = checkType( obj_type );
  if ( type_check_map ) {
    callback( type_check_map );
    return;
  }
}

dbHandle.collection(
  obj_type,
  function ( outer_error, collection ) {
    var options_map = { safe: true, single: true };
    collection.remove( find_map, options_map,
      function ( inner_error, delete_count ) {
        callback({ delete_count: delete_count });
      }
    );
  }
);

```

Метод update перенесен из webapp/lib/routes.js. Логика сделана чуть более общей.

Проверяем допустимость типа объекта. Если тип недопустим, возвращаем JSON-документ с описанием ошибки.

Метод удаления (destroy) перенесен из webapp/lib/routes.js. Логика сделана чуть более общей.

Проверяем допустимость типа объекта. Если тип недопустим, возвращаем JSON-документ с описанием ошибки.


```

    );
  }
);
};

module.exports = { ← Экспортируем все открытые методы.
  makeMongoId : mongodb.ObjectId,
  checkType   : checkType,
  construct    : constructObj,
  read         : readObj,
  update       : updateObj,
  destroy      : destroyObj
};

// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

// ----- НАЧАЛО ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----
dbHandle.open( function () {
  console.log( '** Установлено подключение к MongoDB **' );
  clearIsOnline(); ← Вызываем метод clearIsOnline сразу после подключения
                    | к MongoDB.
});

// загружаем схемы в память (objTypeMap)
(function () { ← Инициализируем в памяти схему.
  var schema_name, schema_path;
  for ( schema_name in objTypeMap ) {
    if ( objTypeMap.hasOwnProperty( schema_name ) ) {
      schema_path = __dirname + '/' + schema_name + '.json';
      loadSchema( schema_name, schema_path );
    }
  }
})();
// ----- КОНЕЦ ИНИЦИАЛИЗАЦИИ МОДУЛЯ -----

```

Модуль routes стал гораздо проще, поскольку большая часть находившейся в нем логики и зависимостей перенесена в модуль CRUD. Модифицированный модуль routes приведен в листинге 8.25. Изменения выделены **полужирным** шрифтом.

Листинг 8.25 ❖ Модифицированный модуль routes – webapp/lib/routes.js

```

/*
 * routes.js - модуль маршрутизации
 */

/*jslint      node    : true, continue : true,
  devel      : true, indent : 2,  maxerr  : 50,
  newcap     : true, nomen  : true, plusplus : true,
  regexp     : true, sloppy : true, vars    : false,

```

```

    white : true
  */
  /*global */

  // ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  'use strict';
  var
    configRoutes, ← Большая часть объявлений переменных перенесена
                    в модуль CRUD, а отсюда удалена.
    crud          = require( './crud' ),
    makeMongoId   = crud.makeMongoId;
  // ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
  ← Удаляем секцию служебных методов.
  // ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
  configRoutes = function ( app, server ) {
    app.get( '/', function ( request, response ) {
      response.redirect( '/spa.html' );
    });
    app.all( '/:obj_type/*?', function ( request, response, next ) {
      response.contentType( 'json' );
      next();
    });
    app.get( '/:obj_type/list', function ( request, response ) {
      crud.read(
        request.params.obj_type,
        {}, {},
        function ( map_list ) { response.send( map_list ); }
      );
    });
    app.post( '/:obj_type/create', function ( request, response ) {
      crud.construct(
        request.params.obj_type,
        request.body,
        function ( result_map ) { response.send( result_map ); }
      );
    });
    app.get( '/:obj_type/read/:id', function ( request, response ) {
      crud.read(
        request.params.obj_type,
        { _id: makeMongoId( request.params.id ) },
        {},
        function ( map_list ) { response.send( map_list ); }
      );
    });
    app.post( '/:obj_type/update/:id', function ( request, response ) {
      crud.update(

```

Используем метод `read` из модуля CRUD для получения списка объектов. В ответ могут быть возвращены данные или сообщение об ошибке. В любом случае отдаем клиенту полученный результат без изменения.

Используем метод `construct` из модуля CRUD для создания пользователя. Отдаем клиенту полученный результат без изменения.

Используем метод `read` из модуля CRUD для получения одного объекта. Отдаем клиенту полученный результат без изменения. Отметим, что этот случай отличается от предыдущего применения метода `read`, потому что в случае успеха возвращается массив, состоящий из единственного объекта.

Используем метод `update` из модуля CRUD для обновления одного объекта. Отдаем клиенту полученный результат без изменения.

```

    request.params.obj_type,
    { _id: makeMongoId( request.params.id ) },
    request.body,
    function ( result_map ) { response.send( result_map ); }
  );
});
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Используем метод `destroy` из модуля `CRUD` для удаления одного объекта. Отдаем клиенту полученный результат без изменения.

Экспортируем метод конфигурирования, как и раньше.

Удаляем секцию инициализации.

Теперь модуль `routes` стал гораздо меньше и пользуется модулем `CRUD` для обслуживания маршрутов. И что, пожалуй, важнее, модуль `CRUD` готов к использованию в модуле `chat`, который мы реализуем в следующем разделе.

8.6. Реализация модуля `chat`

Мы хотим, чтобы серверное приложение предоставляло нашему SPA функциональность чата. До сих пор мы были заняты созданием клиента, пользовательского интерфейса и поддерживающей инфраструктуры на стороне сервера. На рис. 8.8 показано, как должно выглядеть приложение, когда все будет готово.

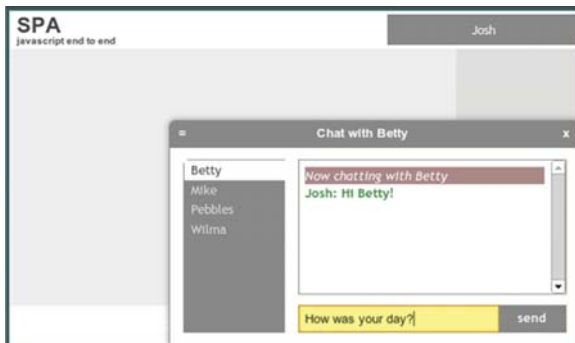


Рис. 8.8 ❖ Законченное приложение `Chat`

К концу этого раздела у нас будет работоспособный сервер чата. Начнем с создания модуля chat.

8.6.1. Начало модуля Chat

Библиотека Socket.IO уже должна быть установлена в каталог webapp. Проверьте, все ли модули присутствуют в манифесте webapp/package.json.

```
{ "name" : "SPA",
  "version" : "0.0.3",
  "private" : true,
  "dependencies" : {
    "express" : "3.2.x",
    "mongodb" : "1.3.x",
    "socket.io" : "0.9.x",
    "JSV" : "4.0.x"
  }
}
```

Убедившись, что манифест правилен, мы можем выполнить команду `npm install`, которая установит `socket.io` и все прочие необходимые модули.

Теперь можно заняться модулем, отвечающим за обмен сообщениями в чате. Мы включим модуль `CRUD`, потому что он заведомо понадобится. Мы сконструируем объект `chatObj` и экспортируем его с помощью `module.exports`. Поначалу в этом объекте будет всего один метод `connect`, который принимает в качестве аргумента `server` экземпляр `http.Server` и начинает прослушивать сокет в ожидании запросов на соединение. Первый вариант модуля показан в листинге 8.26.

Листинг 8.26 ❖ Первый вариант модуля обмена сообщениями – `webapp/lib/chat.js`

```
/*
 * chat.js – модуль обмена сообщениями в чате
 */

/*
 *jslint      node    : true, continue : true,
  devel      : true, indent : 2,  maxerr  : 50,
  newcap     : true, nomen  : true, plusplus : true,
  regexp     : true, sloppy : true, vars    : false,
  white      : true
 */
/*global */

// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
```

```

var
  chatObj,
  socket = require( 'socket.io' ),
  crud   = require( './crud' );
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
chatObj = {
  connect : function ( server ) {
    var io = socket.listen( server );
    return io;
  }
};

module.exports = chatObj;
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Напомним (см. главу 6), что клиент посылает серверу сообщения – `adduser`, `updatechat`, `leavechat`, `disconnect` и `updateavatar`, – пользуясь пространством имен `/chat`. Настроим клиент чата для обработки этих сообщений, как показано в листинге 8.27. Изменения выделены **полужирным** шрифтом.

Листинг 8.27 ❖ Настройка приложения и заголовка обработчиков сообщений – `webapp/lib/chat.js`

```

/*
 * chat.js – модуль обмена сообщениями в чате
 */
...
// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
chatObj = {
  connect : function ( server ) {
    var io = socket.listen( server );

    // Начало настройки объекта io
    io
      .set( 'blacklist' , [] )
      .of( '/chat' )
      .on( 'connection', function ( socket ) {
        socket.on( 'adduser', function () {} );
        socket.on( 'updatechat', function () {} );
        socket.on( 'leavechat', function () {} );
        socket.on( 'disconnect', function () {} );
        socket.on( 'updateavatar', function () {} );
      }
    );
  }
};
// Конец настройки объекта io

```

Настраиваем Socket.IO, так чтобы не помещать в черный список при получении сообщения `disconnect` или любого другого. Активация `disconnect` позволила бы получать уведомления об отключении клиента, обнаруженном с помощью периодических контрольных сообщений Socket.IO.

Настраиваем Socket.IO, так чтобы отвечать на сообщения в пространстве имен `/chat`.

Определяем функцию, которая вызывается, когда клиент подключается в пространстве имен `/chat`.

Создаем обработчики сообщений в пространстве имен `/chat`.

```

        return io;
    }
};

module.exports = chatObj;
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Вернемся к модулю `routes`, включим в него модуль `chat`, а затем с помощью метода `chat.connect` инициализируем соединение `Socket.IO`. В качестве аргумента `server` передаем экземпляр `http.Server`, как показано в листинге 8.28. Изменения выделены **полужирным** шрифтом.

Листинг 8.28 ❖ Инициализация чата в модуле `routes` – `webapp/lib/routes.js`

```

/*
 * routes.js - модуль маршрутизации
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
    configRoutes,
    crud      = require( './crud' ),
    chat      = require( './chat' ),
    makeMongoId = crud.makeMongoId;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
configRoutes = function ( app, server ) {
    ...
    chat.connect( server );
};

module.exports = { configRoutes : configRoutes };
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Запустив сервер командой `node app.js`, мы увидим в журнале `Node.js` сообщение `info - socket.io started`. Как и раньше, мы можем перейти в браузере по адресу `http://localhost:3000` для просмотра приложения и управления объектами пользователей.

Мы объявили обработчики всех сообщений, но теперь надо добить-ся, чтобы они отвечали. Начнем с обработчика сообщения `adduser`.

Почему именно веб-сокеты?

У веб-сокетов, по сравнению с другими применяемыми в браузерах способами коммуникации, почти в реальном времени имеется ряд существенных преимуществ.

- Для поддержания соединения веб-сокету требуется кадр данных, содержащий всего два байта. Сравните с HTTP-вызовом AJAX (применяе-

мым для реализации долгого опроса), где зачастую в кадре передается несколько килобайтов информации (точное значение зависит от количества и размера куков).

- По сравнению с долгим опросом, веб-сокеты потребляют приблизительно 1–2% пропускной способности сети, а время задержки примерно в три раза меньше. Кроме того, веб-сокеты лучше работают с брандмауэрами.
- Веб-сокеты – полнодуплексный механизм, тогда как большинство других решений таковыми не являются и нуждаются в двух соединениях.
- В отличие от сокетов Flash, веб-сокеты работают в любом современном браузере и почти на любой платформе, в том числе на мобильных устройствах – смартфонах и планшетах.

В библиотеке Socket.IO предпочтение отдается веб-сокетам, но вам будет приятно узнать, что в случае, когда они недоступны, производится согласование оптимальной альтернативы с клиентом.

8.6.2. Создание обработчика сообщения `adduser`

Когда пользователь пытается аутентифицироваться, клиент посылает нашему серверному приложению сообщение `adduser`, сопровождаемое данными о пользователе. Обработчик этого сообщения должен:

- попытаться найти в базе данных MongoDB пользователя с указанным именем, применяя модуль CRUD;
- если пользователь с указанным именем существует, использовать найденный объект;
- если пользователя с указанным именем *не* существует, создать новый объект пользователя с таким именем и вставить его в базу данных. Использовать вновь созданный объект;
- обновить объект пользователя в MongoDB, отметив, что он находится в онлайн (`is_online: true`);
- обновить объект `chatterMap`, сохранив в нем идентификатор пользователя и соответствующее ему соединение в виде пары ключ–значение.

Реализация этого плана показана в листинге 8.29. Изменения выделены **полужирным** шрифтом.

Листинг 8.29 ❖ Создание обработчика сообщения `adduser` – `webapp/lib/chat.js`

```
/*
 * chat.js – модуль обмена сообщениями в чате
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  emitUserList, signIn, chatObj, ← Объявляем служебные методы emitUserList и signIn.
```

```

socket = require( 'socket.io' ),
crud   = require( './crud' ),

makeMongoId = crud.makeMongoId,
chatterMap = {};
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
// emitUserList - отправить список пользователей всем подключенным
// клиентам
//
emitUserList = function ( io ) {
  crud.read(
    'user',
    { is_online : true },
    {},
    function ( result_list ) {
      io
        .of( '/chat' )
        .emit( 'listchange', result_list );
    }
  );
};

// signIn - обновить свойство is_online и объект chatterMap
//
signIn = function ( io, user_map, socket ) {
  crud.update(
    'user',
    { '_id' : user_map._id },
    { is_online : true },
    function ( result_map ) {
      emitUserList( io );
      user_map.is_online = true;
      socket.emit( 'userupdate', user_map );
    }
  );
  chatterMap[ user_map._id ] = socket;
  socket.user_id = user_map._id;
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
chatObj = {
  connect : function ( server ) {
    var io = socket.listen( server );

    // Начало настройки объекта io
    io

```

Добавляем объект `chatterMap` для хранения соответствия между идентификаторами пользователей и сокетами.

Метод `emitUserList` рассылает список людей в онлайн всем подключенным клиентам.

Список людей в онлайн рассылается в составе сообщения `listchange`.

Метод `signIn` аутентифицирует существующего пользователя и обновляет его состояние (`is_online : true`).

Аутентифицировав пользователя, вызываем метод `emitUserList`, чтобы разослать всем подключенным клиентам список людей в онлайн.

Добавляем пользователя в `chatterMap`, указывая в качестве ключа его идентификатор, а в качестве значения – соответствующий сокет.


```

.set( 'blacklist' , [] )
.of( '/chat' )
.on( 'connection', function ( socket ) {

    // Начало обработчика сообщения /adduser/
    // Описание: обеспечивает аутентификацию
    // Аргументы: один объект user_map.
    // user_map должен обладать следующими свойствами:
    //   name = имя пользователя
    //   cid = клиентский идентификатор
    // Действие:
    //   Если пользователь с указанным именем уже существует в
    //   Mongo, использовать существующий объект пользователя
    //   и игнорировать остальные данные.
    //   Если пользователя с указанным именем не существует в
    //   Mongo, создать новый объект и использовать его.
    //   Послать отправителю сообщение 'userupdate', завершив
    //   тем самым цикл аутентификации. В ответное сообщение
    //   включить клиентский идентификатор, чтобы клиент мог
    //   коррелировать его с пользователем, но не сохранять
    //   этот идентификатор в MongoDB.
    //   Пометить, что пользователь в онлайн, и отправить обновленный
    //   список пользователей в онлайн всем клиентам, в
    //   том числе тому, от которого поступило сообщение
    //   'adduser'.
    //   Модифицируем обработчик сообщения adduser, так чтобы
    //   он принимал полученный от клиента объект user_map.

    socket.on( 'adduser', function ( user_map ) {
        crud.read(
            'user',
            { name : user_map.name },
            {},
            function ( result_list ) {
                var
                    result_map,
                    cid = user_map.cid;

                delete user_map.cid;

                // использовать существующего пользователя с указанным именем
                if ( result_list.length > 0 ) {
                    result_map = result_list[ 0 ];
                    result_map.cid = cid;
                    signIn( io, result_map, socket );
                }

                // создать нового пользователя
                // с указанным именем
                else {
                    user_map.is_online = true;
                    crud.construct(

```

Документируем обработчик сообщения adduser.

Используем метод crud.read для поиска всех пользователей с указанным именем.

Если объект пользователя с указанным именем найден, вызываем метод signIn, передавая ему найденный объект. Этот метод отправляет клиенту сообщение updateuser, передавая в нем объект user_map в качестве данных. Он также вызывает метод emitUserList для рассылки списка людей в онлайн всем подключенным клиентам.

Если объект пользователя с указанным именем не найден, создаем новый объект и сохраняем его в коллекции MongoDB. Добавляем этот объект в chatterMap, указывая в качестве ключа идентификатор пользователя, а в качестве значения — соответствующий socket. Затем вызываем метод emitUserList для рассылки списка людей в онлайн всем подключенным клиентам.

```

        'user',
        user_map,
        function ( result_list ) {
            result_map      = result_list[ 0 ];
            result_map.cid = cid;
            chatterMap[ result_map._id ] = socket;
            socket.user_id = result_map._id;
            socket.emit( 'userupdate', result_map );
            emitUserList( io );
        }
    );
}
}
});
// Конец обработчика сообщения /adduser/

socket.on( 'updatechat',    function () {} );
socket.on( 'leavechat',    function () {} );
socket.on( 'disconnect',   function () {} );
socket.on( 'updateavatar', function () {} );
}
);
// End io setup
return io;
}
};

module.exports = chatObj;
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

На привыкание к программированию с помощью обратных вызовов может потребоваться некоторое время, но принцип всегда один и тот же: мы вызываем некоторый метод, а когда он завершается, выполняется предоставленная нам функция обратного вызова. По существу, процедурный код вида

```

var user = user.create();
if ( user ) {
    // сделать что-то с объектом user
}

```

в событийно-управляемой программе превращается в такой:

```

user.create( function ( user ) {
    // сделать что-то с объектом user
});

```

Мы пользуемся обратными вызовами, потому что многие функции в Node.js асинхронны. В примере выше после вызова `user.create`

интерпретатор JavaScript продолжит выполнение следующего далее кода, не дожидаясь завершения вызова. Один из способов воспользоваться результатами, как только они будут готовы, — указать функцию обратного вызова¹. Если вы знакомы с реализацией AJAX-вызовов в jQuery, то знаете, что там тоже используется механизм обратных вызовов:

```
$.ajax({
  'url': '/path',
  'success': function ( data ) {
    // сделать что-то с data
  }
});
```

Теперь мы можем перейти в браузере по адресу localhost:3000 и аутентифицироваться. Призываем вас поэкспериментировать с этим дома. А мы пока дадим людям возможность пообщаться в чате.

8.6.3. Создание обработчика сообщения updatechat

Для реализации аутентификации потребовалось написать довольно много кода. Теперь приложение хранит в базе данных MongoDB сведения о пользователях, управляет их состоянием и рассылает список людей в онлайн всем подключенным клиентам. Поддержка обмена сообщениями сравнительно проста, особенно после того, как логика аутентификации реализована.

Посылая сообщение `updatechat` серверу, клиент просит доставить кому-то сообщение. Обработчик сообщения `updatechat` должен делать следующее:

- извлечь из находящихся в сообщении данных имя получателя;
- определить, находится ли получатель в онлайн;
- если получатель находится в онлайн, отправить ему данные через ассоциированный с ним сокет;
- если получатель *не* находится в онлайн, отправить сообщение отправителю через ассоциированный с ним сокет. В сообщении

¹ Другой механизм называется *обещанием*, он, вообще говоря, более гибкий, чем простые обратные вызовы. Обещания реализованы в библиотеках `Q` (`npm install q`) и `Promised-IO` (`npm install promised-io`). В библиотеке jQuery для Node.js также есть богатый набор методов, относящихся к обещаниям, со знакомым интерфейсом. В приложении Б демонстрируется использование jQuery совместно с Node.js.

должно находиться уведомление о том, что получатель не в онлайне.

Эта логика реализована в листинге 8.30. Изменения выделены **полужирным** шрифтом.

Листинг 8.30 ❖ Создание обработчика сообщения `updatechat` – `webapp/lib/chat.js`

```

/*
 * chat.js – модуль обмена сообщениями в чате
 */
...
// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
chatObj = {
  connect : function ( server ) {
    var io = socket.listen( server );

    // Начало настройки объекта io
    io
      .set( 'blacklist' , [] )
      .of( '/chat' )
      .on( 'connection', function ( socket ) {
        ...
        // Начало обработчика сообщения /adduser/
        ...
        socket.on( 'adduser', function ( user_map ) {
          ...
        });
        // Конец обработчика сообщения /adduser/

        // Начало обработчика сообщения /updatechat/
        // Описание: занимается обработкой сообщений в чате
        // Аргументы: один объект chat_map.
        // chat_map должен обладать следующими свойствами:
        //   dest_id   = идентификатор получателя
        //   dest_name = имя получателя
        //   sender_id = идентификатор отправителя
        //   msg_text  = текст сообщения
        // Действие:
        //   Если получатель в онлайн, отправить ему chat_map.
        //   Если нет, послать отправителю сообщение
        //   'пользователь вышел из чата'
        //
        socket.on( 'updatechat', function ( chat_map ) {
          if ( chatterMap.hasOwnProperty( chat_map.dest_id ) ) {
            chatterMap[ chat_map.dest_id ]
              .emit( 'updatechat', chat_map );
          }
          else {

```

Документируем обработчик сообщения `updatechat`.

В аргументе `chat_map` передаются данные, полученные от клиента.

Если получатель находится в онлайн (его идентификатор имеется в `chatterMap`), то отправляем ему `chat_map` через соответствующий сокет.

```

        socket.emit( 'updatechat', { ←
            sender_id : chat_map.sender_id,
            msg_text : chat_map.dest_name + ' вышел из чата.'
        });
    }
});
// Конец обработчика сообщения /updatechat/

socket.on( 'leavechat',    function () {} );
socket.on( 'disconnect',  function () {} );
socket.on( 'updateavatar', function () {} );
}
);
// Конец настройки объекта io
return io;
}
};

module.exports = chatObj;
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Если получатель не находится в онлайн, то посылаем отправителю новый объект `chat_map` с уведомлением о том, что получатель уже вышел из чата.

Теперь мы можем перейти в браузере по адресу `localhost:3000` и аутентифицироваться. Аутентифицировавшись в другом окне от имени другого пользователя, мы сможем обмениваться сообщениями. Как обычно, призываем вас поэкспериментировать дома. Осталось еще реализовать отключение и аватары. В следующем разделе мы рассмотрим отключение.

8.6.4. Создание обработчиков отключения

Клиент может закрыть сеанс одним из двух способов. Во-первых, пользователь может выйти, щелкнув по своему имени в правом верхнем углу окна браузера. При этом серверу посылается сообщение `leavechat`. Во-вторых, пользователь может просто закрыть окно браузера. Тогда серверу посылается сообщение `disconnect`. В любом случае Socket.IO закрывает сокет и выполняет необходимую очистку.

Получив любое из сообщений `leavechat` или `disconnect`, наше серверное приложение должно предпринять два действия: пометить, что пользователь больше не в онлайн (`is_online : false`), и разослать обновленный список людей в онлайн всем подключенным клиентам. Эта логика реализована в листинге 8.31. Изменения выделены **полужирным шрифтом**.

Листинг 8.31 ❖ Добавление обработчиков отключения – `webapp/lib/chat.js`

```

/*
 * chat.js - модуль обмена сообщениями в чате

```

```

*/
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
    emitUserList, signIn, signOut, chatObj,
    socket = require( 'socket.io' ),
    crud = require( './crud' ),

    makeMongoId = crud.makeMongoId,
    chatterMap = {};
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
...
// signOut - изменить свойство is_online и объект chatterMap
//
signOut = function ( io, user_id ) {
    crud.update(
        'user',
        { '_id' : user_id },
        { is_online : false },
        function ( result_list ) { emitUserList( io ); }
    );
    delete chatterMap[ user_id ];
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
chatObj = {
    connect : function ( server ) {
        var io = socket.listen( server );

        // Начало настройки объекта io
        io
            .set( 'blacklist' , [] )
            .of( '/chat' )
            .on( 'connection', function ( socket ) {

                ...
                // Начало методов отключения
                socket.on( 'leavechat', function () {
                    console.log(
                        '** пользователь %s вышел **', socket.user_id
                    );
                    signOut( io, socket.user_id );
                });

                socket.on( 'disconnect', function () {
                    console.log(

```

Пометить, что пользователь вышел из чата, присвоив атрибуту is_online значение false.

После выхода пользователя всем подключенным клиентам рассылается новый список людей в онлайн.

Ушедший пользователь удаляется из chatterMap.

```
        '** пользователь %s закрыл окно или вкладку браузера **',
        socket.user_id
    ); signOut( io, socket.user_id );
    });
    // Конец методов отключения

    socket.on( 'updateavatar', function () {} );
}
);
// Конец настройки объекта io

return io;
}
};

module.exports = chatObj;
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
```

Теперь мы можем открыть несколько окон браузера, перейти во всех по адресу <http://localhost:3000> и аутентифицироваться от имени разных пользователей, щелкнув мышью в правом верхнем углу каждого окна. После этого можно будет посылать сообщения. Мы намеренно не стали исправлять одну ошибку, оставив это в качестве упражнения для читателя: сервер разрешает одному и тому же пользователю зайти из разных клиентов. Такого быть не должно. Вы можете исправить это, немного изменив алгоритм работы с объектом `chatMap` в обработчике сообщения `adduser`.

Осталась нереализованной только одна функция: синхронизация аватаров.

8.6.5. Создание обработчика сообщения `updateavatar`

Обмен сообщениями через веб-сокеты можно использовать для любого взаимодействия между клиентом и сервером. Если необходимо взаимодействие с браузером в режиме, близком к реальному времени, то часто эта технология оказывается наилучшим выбором. Чтобы продемонстрировать еще одно применение `Socket.IO`, мы включили в чат аватары; пользователи могут перемещать их по экрану и изменять цвет. При любом изменении аватара `Socket.IO` немедленно распространяет изменение всем остальным участникам. Как это выглядит, показано на рис. 8.9, 8.10 и 8.11.

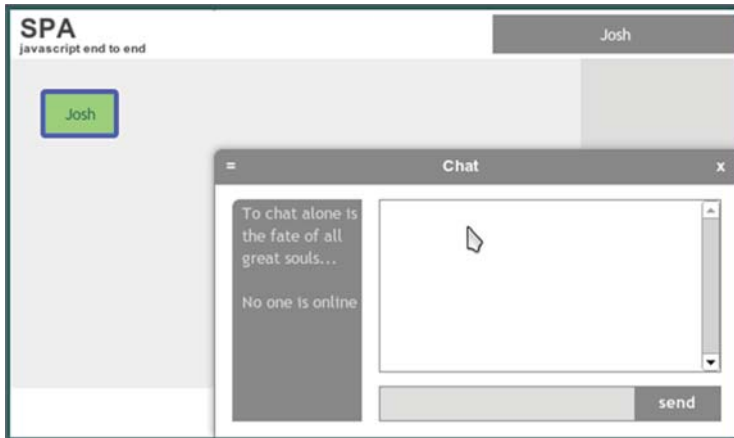


Рис. 8.9 ❖ Аватар после аутентификации

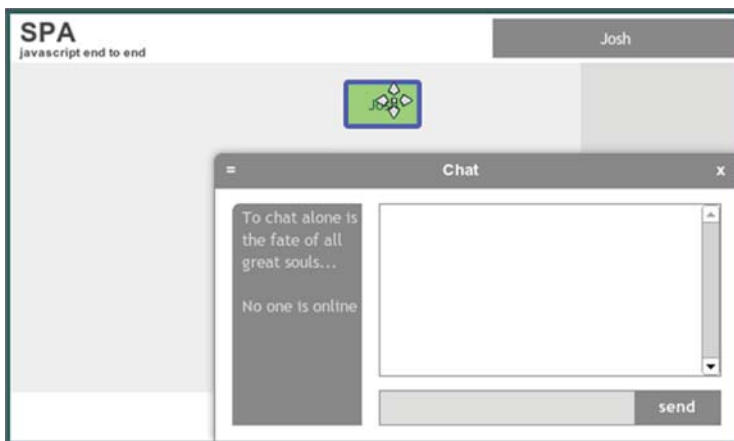


Рис. 8.10 ❖ Перемещение аватара

Соответствующий код на стороне клиента был приведен в главе 6, а сейчас мы должны собрать все воедино. После того как мы настроили Node.js, MongoDB и Socket.IO, серверный код для поддержки этой возможности оказывается на удивление коротким. В листинге 8.32 показан обработчик сообщения, который мы добавили в файл `lib/chat.js`.

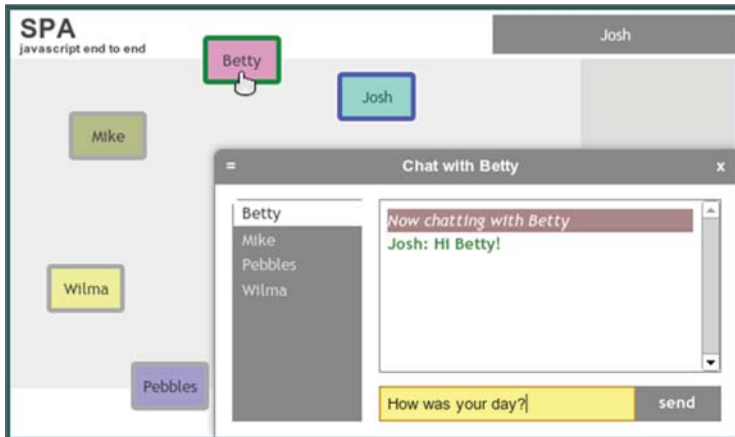


Рис. 8.11 ❖ Аватары после аутентификации других пользователей

Листинг 8.32 ❖ А вот и аватары – webapp/lib/chat.js

```

/*
 * chat.js - модуль обмена сообщениями в чате
 */
...

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
chatObj = {
  connect : function ( server ) {
    var io = socket.listen( server );

    // Начало настройки объекта io
    io
      .set( 'blacklist' , [] )
      .of( '/chat' )
      .on( 'connection', function ( socket ) {

        ...

        // Конец методов отключения

        // Начало обработчика сообщения /updateavatar/
        // Описание: обрабатывает обновления аватаров на стороне клиента
        // Аргументы: один объект avtr_map.
        // avtr_map должен обладать следующими свойствами:
        //   person_id = идентификатор человека, чей аватар обновлен
        //   css_map = хэш, в котором содержатся координаты левого
        //     верхнего угла и цвет фона
        // Действие:

```

```

// Этот обработчик обновляет записи в MongoDB, а затем
// рассылает измененный список людей всем клиентам.
//
socket.on( 'updateavatar', function ( avtr_map ) {
  crud.update(
    'user',
    { '_id' : makeMongoId( avtr_map.person_id ) },
    { css_map : avtr_map.css_map },
    function ( result_list ) { emitUserList( io ); }
  );
});
// Конец обработчика сообщения /updateavatar/
}
);
// Конец настройки объекта io

return io;
}
};

module.exports = chatObj;
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Запустим сервер командой `node app.js`, перейдем в браузере по адресу `http://localhost:3000/` и аутентифицируемся. Откроем второе окно браузера и аутентифицируемся от имени другого пользователя. В этот момент мы видим только один аватар, потому что они перекрываются. Мы можем перетащить аватар мышью или пальцем. Чтобы изменить цвет аватара, нужно щелкнуть по нему мышью или коснуться пальцем. Работает и на настольном ПК, и на сенсорном устройстве. В любом случае серверное приложение синхронизирует аватары почти в реальном времени.

Обмен сообщениями – взаимодействие в режиме, близком к реальному времени. С помощью веб-сокетов мы можем создавать приложения, позволяющие людям, которые находятся в разных местах, совместно собирать пазл, проектировать двигатель или рисовать картину – возможности безграничны. Так с каждым днем приближается обещанная веб в реальном времени.

8.7. Резюме

В этой главе мы настроили базу данных MongoDB, подключились к ней из Node.js и выполнили ряд основных операций CRUD. Мы познакомились с MongoDB и обсудили ее достоинства и недостатки. Мы также продемонстрировали, как осуществить валидацию данных

перед помещением их в базу, применяя на сервере тот же код, что и на стороне клиента. Это решает болезненную проблему написания сначала валидатора на сервере, а затем переписывания его на JavaScript для браузера.

Мы представили библиотеку Socket.IO и показали, как с ее помощью организовать обмен сообщениями в чате. Мы вынесли функциональность **CRUD** в отдельный модуль, чтобы использовать общий код для обслуживания запросов через HTTP и через Socket.IO. И наконец, мы применили технику обмена сообщениями для почти мгновенной синхронизации аватаров на различных клиентах.

В следующей главе мы обсудим вопрос о подготовке нашего SPA к промышленной эксплуатации. Мы рассмотрим некоторые проблемы, с которыми сталкивались при организации хостинга SPA, и поговорим о том, как их решить.

Глава 9

Подготовка SPA к промышленной эксплуатации

В этой главе:

- ✧ Оптимизация SPA для поисковых систем.
- ✧ Использование Google Analytics.
- ✧ Размещение статического содержимого в сети доставки содержимого (CDN).
- ✧ Протоколирование ошибок на стороне клиента.
- ✧ Кэширование и отключение кэширования.

Эта глава опирается на код, написанный в главе 8. Мы рекомендуем скопировать все созданные там файлы в новый каталог «chapter_9» и уже в нем производить изменения.

Мы закончили реализацию быстрого одностраничного приложения на базе проверенной практикой архитектуры, но остаются еще некоторые вопросы, относящиеся не столько к программированию, сколько к эксплуатации. Мы должны сделать так, чтобы Google и другие поисковые системы могли найти в нашем SPA то, что им нужно. Наш веб-сервер должен взаимодействовать с роботами-обходчиками, которые видят и индексируют наш сайт по-другому, поскольку не выполняют JavaScript-код, с помощью которого SPA генерирует его содержимое. Мы хотим также использовать аналитические средства. В традиционном сайте аналитические данные обычно собираются за счет включения в каждую HTML-страницу небольшого фрагмента JavaScript-кода. Но поскольку HTML-разметка в SPA генерируется JavaScript-кодом, нам необходим другой подход.

Мы также хотим подправить наше SPA, так чтобы оно предоставляло детальный журнал со сведениями о трафике, поведении пользователя и ошибках. Ведение подобного журнала на сервере дает много интересной информации о традиционных сайтах. Но в SPA большая

часть взаимодействия с пользователем перенесена на сторону клиента, поэтому следует действовать иначе. Мы хотим, чтобы SPA отвечать на действия пользователя очень быстро. Один из способов уменьшить время реакции – использовать сеть доставки содержимого (CDN) для возврата статических файлов и данных. Другой способ – использовать кэширование на сервере и средствами протокола HTTP.

Для начала сделаем так, чтобы по содержимому нашего SPA можно было осуществлять поиск.

9.1. Поисковая оптимизация SPA

При индексировании сайтов Google и другие поисковые системы не выполняют JavaScript-код. На первый взгляд, это ставит SPA в исключительно невыгодное положение по сравнению с традиционными сайтами. Если Google не знает о сайте, бизнес можно считать мертвым, и эта ужасающая перспектива может вообще отвлечь от идеи SPA.

Но на самом деле SPA даже имеют преимущество над традиционными сайтами в плане поисковой оптимизации, потому что Google и другие компании осознали проблему и приняли вызов. Они создали механизм, который позволяет не просто индексировать динамические страницы SPA, но еще и оптимизировать их специально для роботов. В этом разделе мы будем говорить только о крупнейшей поисковой системе, Google, но Yahoo, Bing и другие большие поисковики поддерживают тот же механизм.

9.1.1. Как Google индексирует SPA

Когда Google индексирует традиционный сайт, его робот-обходчик (он называется *Googlebot*) сначала сканирует и индексирует содержимое URI верхнего уровня (например, www.myhome.com). Затем он следует по всем ссылкам, найденным на этой странице, и индексирует страницы, на которые они ведут. Этот процесс повторяется для ссылок на страницах второго уровня и т. д. В конечном итоге будет проиндексировано все содержимое данного сайта и тех, на которые он ссылается.

Пытаясь проиндексировать SPA, Googlebot видит всего лишь пустой HTML-контейнер (обычно пустой тег `div` или `body`), а значит, нечего индексировать и нечего обходить, поэтому и результат соответственный (его можно найти в такой цилиндрической «папке», стоящей на полу сбоку от стола).

Если бы история на этом заканчивалась, то тут бы и конец «одностраничности» многих веб-приложений и сайтов. К счастью, Google и другие поисковики понимают важность SPA и предоставили разработчикам инструменты, которые позволяют сообщить роботу информацию, даже более содержательную, чем в случае традиционных сайтов.

Чтобы SPA распознавалось роботом, мы прежде всего должны принять во внимание, что сервер может узнать, от кого поступил запрос: от робота или от человека, работающего с браузером. Узнать и соответственно отреагировать. Если зашел человек, то сервер отвечает как обычно, а если робот – то сервер возвращает оптимизированную страницу, содержащую информацию, которую мы хотим сообщить роботу в формате, который тот без труда разберет.

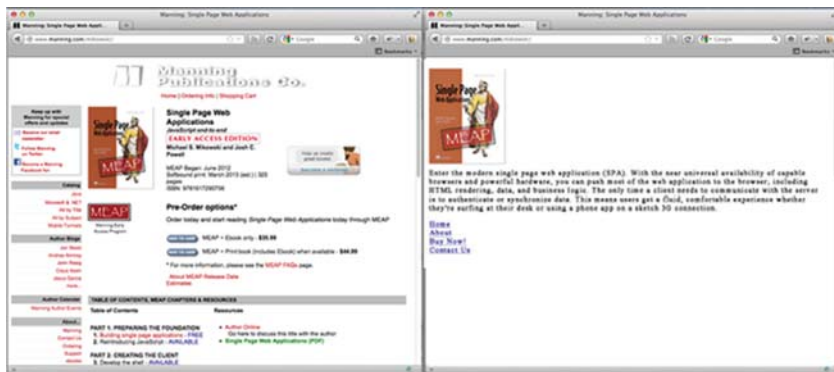


Рис. 9.1 ❖ Вид начальной страницы для браузера и для робота-обходчика

Как должна выглядеть начальная страница нашего сайта, оптимизированная для робота? Наверное, на ней должен быть логотип или другое главное изображение, которое мы хотели бы видеть в результатах поиска, какой-то предназначенный специально для поисковой системы текст, объясняющий, что делает приложение, и список ссылок на другие страницы, которые мы просим Google проиндексировать. Чего на такой странице нет, так это стилей CSS и сложной HTML-разметки. Нет там и JavaScript-кода или ссылок на области сайта, которые не предназначены для индексирования (например, страницы с заявлением об отказе от ответственности и других страниц, на которые пользователи не должны попадать из результатов

поиска). На рис. 9.1 показано, как страница может выглядеть для браузера и для робота.

Робот следует по ссылкам на странице не так, как человек, потому что мы включили в URI якорь, начинающийся со специальной комбинации символов `#!`. Например, если в нашем SPA ссылка на страницу пользователя имеет вид `/index.htm#!page=user:id,123`, то робот увидит символы `#!` и будет знать, что нужно искать страницу по адресу `/index.htm?_escaped_fragment_=page=user:id,123`. Зная, как поведет себя робот, мы можем запрограммировать сервер, чтобы он возвращал в ответ на такой запрос «слепок» – страницу с такой HTML-разметкой, которая обычно генерируется JavaScript-кодом в браузере. Google проиндексирует этот слепок, но человек, который щелкнет по соответствующей ссылке в результатах поиска, попадет на страницу `/index.htm#!page=user:id,123`. Теперь в дело вступает JavaScript-код SPA и отрисовывает страницу, как положено.

Это дает разработчикам SPA возможность настраивать сайт отдельно для Google и отдельно для пользователей. Вместо того чтобы в муках придумывать текст, который был бы разборчивым и привлекательным для людей и в то же время понятным роботу, мы можем оптимизировать страницы для каждой категории «посетителей» порознь, не заботясь об интересах другой. Путь обходчика по нашему сайту можно контролировать, направляя людей с результатов поиска на определенный набор парадных страниц. Со стороны программиста это требует больше работы, но окупается сторицей с точки зрения положения в результатах поиска и удержания клиентов.

На момент написания этой книги Googlebot объявляет о себе, включая в запрос заголовок User-Agent вида `Googlebot/2.1 (+http://www.googlebot.com/bot.html)`. Мы можем написать для нашего серверного приложения функцию промежуточного уровня, которая сравнивает строку пользовательского агента с этим значением, и если они совпадают, то отправляет оптимизированную для робота начальную страницу. В противном случае запрос обрабатывается как обычно. Можно также встроить эту логику в код маршрутизатора, как показано в листинге 9.1.

Листинг 9.1 ❖ Обнаружение Googlebot и отправка альтернативного содержимого в файле `routes.js`

```
...
var agent_text = 'Добро пожаловать в современное одностраничное веб-приложение
+ '(SPA). Благодаря почти повсеместному присутствию развитых браузеров и '
+ 'мощному оборудованию мы можем перенести в браузер большую часть'
+ 'веб-приложения, в том числе отрисовку HTML, данные и бизнес-логику.'
```

HTML-разметка, отдаваемая роботу-обходчику. ←

```
+ 'Клиент должен обращаться к серверу только для аутентификации и '
+ 'синхронизации данных. Это означает, что пользователи могут получить '
+ 'гибкие и удобные средства работы вне зависимости от способа выхода '
+ 'в сеть: с настольного ПК или с телефона по 3G-соединению.'
+ '<br><br>'
+ '<a href="/index.htm#page=home">В начало</a><br>'
+ '<a href="/index.htm#page=about">О программе</a><br>'
+ '<a href="/index.htm#page=buynow">Купить</a><br>'
+ '<a href="/index.htm#page=contact us">Контакты</a><br>';
```

```
app.all( '*', function ( req, res, next ) {
  if ( req.headers['user-agent'] ===
    'Googlebot/2.1 (+http://www.googlebot.com/bot.html)' ) {
    res.contentType( 'html' );
    res.end( agent_text );
  }
  else {
    next();
  }
});
...
```

Обнаруживаем Googlebot по строке пользовательского агента; другие обходчики идентифицируют себя иначе, и, потратив немного времени, можно распознавать и их тоже. Если опознан робот, то в свойство `contentType` записывается HTML, и роботу посылается ответ в обход нормальной процедуры маршрутизации.

Если пользовательский агент — не робот, то вызываем `next()`, чтобы перейти к следующему маршруту и продолжить нормальную обработку.

На первый взгляд кажется, что такой код будет трудно протестировать, потому что Googlebot нам не принадлежит. Google предлагает сервис, который позволяет это сделать для промышленных сайтов, в составе инструментария Webmaster Tools (<http://support.google.com/webmasters/bin/answer.py?hl=en&answer=158587>), но можно поступить еще проще: подделать строку пользовательского агента. Раньше для этого требовалось запускать программы из командной строки, но с появлением инструментов разработчика в Chrome стало достаточно нажать кнопку и отметить флажок.

1. Откройте инструменты разработчика в Chrome, щелкнув по значку с тремя горизонтальными полосками в правом углу панели инструментов Google, выберите из меню пункт **Инструменты**, а затем **Инструменты разработчика**.
2. В правом нижнем углу окна находится значок шестеренки; щелкнув по нему, вы увидите дополнительные настройки, в том числе отключение кэширования (Disable cache) и включение протоколирования AJAX-запросов (Log XmlHttpRequests).
3. На второй вкладке **Overrides** (Переопределения) отметьте флажок **User Agent** (Пользовательский агент) и выберите произвольный агент из раскрывающегося списка: Chrome, Firefox, IE, iPad и прочие. Агент Googlebot в списке отсутствует. Чтобы установить его, выберите пункт **Other** (Прочие) и скопируйте строку пользовательского агента в поле ввода.

4. Теперь эта вкладка будет представляться от имени Googlebot, и, перейдя в ней по адресу нашего сайта, мы увидим страницу для обходчика.

Очевидно, что у разных приложений будут различные представления о том, как поступать с обходчиками, но, наверное, в нашем случае всегда отдавать роботу Googlebot одну и ту же страницу недостаточно. Мы должны решить, какие еще страницы нужно индексировать, и встроить в приложение средства для сопоставления URI-адресу, содержащему часть `_escaped_fragment_=key=value`, подходящего содержимого. Как бы то ни было, из этой книги вы должны вынести знания, которые позволят наилучшим образом представить содержимое своего приложения для робота. Возможно, вы захотите проявить фантазию и передать ответственность за формирование ответа сервера какому-нибудь стоящему перед ним инфраструктурному компоненту, но мы обычно выбираем более простой путь: создаем специальные страницы и помещаем их в отдельный файл с маршрутами для роботов.

В Сети существует много других законных роботов-обходчиков. Учтя в сервере робота Google, мы затем можем обобщить код и на все остальные.

9.2. Облачные и сторонние службы

Многие компании предлагают службы, полезные для разработки и управления приложениями и позволяющие сэкономить немало времени. Компании поменьше могут воспользоваться некоторыми из таких служб. Для разработки SPA наибольший интерес представляют анализ работы сайта, протоколирование клиентских операций и CDN-сети.

9.2.1. Анализ работы сайта

Важным инструментом в арсенале веб-разработчика является возможность собирать аналитические данные о работе сайта. Разработчики традиционных сайтов привыкли полагаться на такие инструменты, как Google Analytics и New Relic, которые дают подробные сведения о том, как посетители используют сайт, и помогают находить узкие места с точки зрения производительности приложения или его полезности для бизнеса (насколько эффективно сайт способствует сбыту). В SPA эти инструменты могут оказаться столь же эффективны – нужно только чуть-чуть изменить подход.

Google Analytics дает простой способ получить статистику о популярности нашего SPA и его различных состояний, а также о том, откуда приходят посетители. Чтобы воспользоваться Google Analytics в традиционном сайте, мы должны поместить фрагмент JavaScript-кода на каждую HTML-страницу, внося несколько мелких изменений с целью классификации страниц. Такой же подход можно было бы применить и к SPA, но тогда мы получили бы аналитические данные только о начальной загрузке страницы. Существует два способа действовать в SPA всю мощь Google Analytics.

1. Использовать Google Events для отслеживания изменений якоря.
2. Использовать Node.js для регистрации событий на стороне сервера.

Сначала рассмотрим Google Events.

Google Events

Компания Google давно поняла, что есть необходимость регистрировать и классифицировать события на странице; разработка SPA – дело сравнительно новое, но Ajax существует уже довольно давно (по меркам веб, очень давно – аж с 1999 года!). Отслеживать события нетрудно, но требует больше ручной работы, чем отслеживание просмотров страницы. В традиционном сайте фрагмент JavaScript-кода обращается к методу `_trackPageView` объекта `_gaq`. Этот метод принимает пользовательские переменные для задания информации о странице, на которой находится фрагмент. Метод отправляет эту информацию в Google, добавляя ее в набор параметров запроса на изображение. Серверы Google используют эти параметры для обработки информации о данном просмотре страницы. При использовании Google Events вызывается другой метод объекта `_gaq`, а именно метод `_trackEvent`, принимающий определенные параметры. Этот метод также отправляет запрос на изображение, добавляя в него параметры, чтобы Google мог обработать информацию о событии.

Последовательность настройки и использования механизма отслеживания событий довольно проста.

1. Настроить мониторинг нашего сайта на сайте Google Analytics.
2. Вызвать метод `_trackEvent`.
3. Просмотреть отчеты.

Метод `_trackEvent` принимает два обязательных параметра и три необязательных:

```
_trackEvent(category, action, opt_label, opt_value, opt_noninteraction)
```

Ниже приведено описание параметров.

- Параметр `category` обязателен и служит для задания имени группы событий, которой данное событие принадлежит. В отчетах он используется для разбивки событий по категориям.
- Параметр `action` обязателен и определяет конкретное действие, которое отслеживается с помощью события.
- Параметр `opt_label` необязателен и служит для именования дополнительных данных о событии.
- Параметр `opt_value` необязателен и служит для передачи числовых данных о событии.
- Параметр `opt_noninteraction` необязателен, он сообщает Google о том, что не нужно использовать данное событие при вычислении показателя отказов.

Например, если мы хотим отслеживать открытие пользователем окна чата, то можем добавить такой вызов `_trackEvent`:

```
_trackEvent( 'chat', 'open', 'home page' );
```

Впоследствии в отчетах мы увидим, что произошло событие чата, что пользователь открыл окно чата и что это случилось на начальной странице. Может быть и такой вызов:

```
_trackEvent( 'chat', 'message', 'game' );
```

Это означает, что произошло событие чата, что пользователь отправил сообщение и сделал это на странице игры. Как и в случае традиционного сайта, разработчик сам решает, как организовать и отслеживать события. Чтобы облегчить себе жизнь, мы можем не включать каждое событие в код модели на стороне клиента, а вставить вызов `_trackEvent` в клиентском маршрутизаторе (той части кода, которая следит за изменениями якоря), передав ему в качестве параметров категорию, действие и метку, определенные в результате анализа изменения.

Google Analytics на стороне сервера

Организовать отслеживание на стороне сервера полезно в том случае, когда мы хотим получить от сервера информацию о том, какие данные запрашивались. Но этот подход не позволяет отследить взаимодействия на стороне клиента, при которых серверу не посылались никаких запросов, а таковых в SPA совсем немало. Из-за невозможности отслеживать действия на стороне клиента этот способ может показаться не очень интересным, но его все же можно с пользой при-

менить для анализа запросов, которые не удалось обслужить из клиентского кэша. Так мы можем выявить медленно выполняющиеся запросы к серверу и другие особенности поведения. Но пусть даже мы и можем таким образом извлечь полезные знания, если бы пришлось выбирать, мы предпочли бы отслеживание на стороне клиента.

Поскольку на сервере работает JavaScript, то вроде бы не должно быть препятствий к модификации кода Google Analytics, так чтобы его можно было выполнить на стороне сервера. И действительно – это не только возможно, но, как и всякая удачная идея, скорее всего, уже реализовано сообществом. Поиск сразу же находит проекты *node-googleanalytics* и *nodeanalytics*.

9.2.2. Протоколирование ошибок на стороне клиента

В традиционном сайте информация обо всех ошибках, произошедших на сервере, записывается в файл журнала. Если аналогичная ошибка происходит на стороне клиента в SPA, то записать информацию о ней некуда. Придется либо самостоятельно написать код для отслеживания ошибок, либо обратиться к сторонней службе. Первый подход сулит большую гибкость в обработке ошибки, зато использование сторонней службы позволит направить время и ресурсы на что-то другое. Кроме того, сторонняя компания, скорее всего, реализовала куда более развитую функциональность, чем мы могли бы себе позволить. Наконец, никто не заставляет нас выбирать либо то, либо другое – вполне можно воспользоваться сторонней службой, а если она что-то не умеет делать, например отслеживать определенные ошибки или передавать их на более высокую ступень, то реализовать требуемые функции своими силами.

Сторонние службы протоколирования ошибок на стороне клиента

Существует несколько сторонних служб, которые умеют собирать и агрегировать информацию об ошибках и о производительности, порождаемую нашим приложением.

- *Airbrake* ориентирована на приложения для Ruby on Rails, но имеет экспериментальную поддержку JavaScript.
- *Bugsense* специализируется на решениях для мобильных приложений. Ее продукт работает с написанными на JavaScript SPA и платформенными мобильными приложениями. Если

ваше приложение ориентировано на мобильных пользователей, это может оказаться неплохим выбором.

- *Errorception* «заточена» под протоколирование ошибок в JavaScript-коде и потому хорошо сопрягается с клиентской частью SPA. Эта компания не так известна, как Airbrake или Bugsense, но нам нравится ее уверенность в себе. *Errorception* ведет блог разработчиков (<http://blog.errorception.com>), где можно узнать о тайнах протоколирования ошибок в программах на JavaScript.
- *New Relic* быстро превращается в отраслевой стандарт мониторинга работы веб-приложений. Эта служба позволяет протоколировать ошибки и вести учет метрик производительности для каждого шага цикла запрос–ответ: от времени, потраченного на поиск в базе данных, до времени отрисовки стилей CSS браузером. Объем данных о поведении клиентской и серверной частей приложения поистине впечатляет.

На момент написания этой книги мы отдаем предпочтение *New Relic* или *Errorception*. *New Relic* предоставляет больше данных, зато *Errorception* отлично справляется с ошибками в JavaScript и проста в настройке.

Протоколирование ошибок на стороне клиента вручную

По сути дела, все вышеупомянутые службы применяют для отправки сведений об ошибках в JavaScript-коде один из двух способов:

1. Перехват ошибок в обработчике события `window.onerror`.
2. Погружение кода в блок `try/catch` и отправка всего, что будет перехвачено.

Событие `window.onerror` лежит в основе большинства сторонних приложений. Это событие возникает для ошибок во время выполнения, но не для ошибок компиляции. Событие `onerror` неидеально, потому что разные браузеры поддерживают его по-разному и к тому же имеются потенциальные бреши в системе защиты, но в нашем арсенале средств для протоколирования ошибок в JavaScript-коде на стороне клиента оно будет основным оружием.

```
<script>
  var obj;
  obj.push( 'string' ); ← Приводит к ошибке, потому что у undefined нет метода push.

  window.onerror = function ( error ) {
    // обработать ошибку ← Ошибка доступна в этом блоке; атрибуты объекта error
    }                                     в разных браузерах отличаются.
</script>
```

Второй способ сводится к обертыванию главного вызова в SPA блоком `try/catch`. При этом перехватываются все синхронные ошибки в приложении, но, к сожалению, информация о них не доходит до обработчика `window.onerror` и не отображается на консоли ошибок. Таким способом невозможно перехватить ошибки в асинхронных вызовах, например в обработчиках событий или в функциях `setTimeout` и `setInterval`. Это означает, что весь код внутри нашей асинхронной функции также нужно обертывать блоком `try/catch`.

```
<script>
  setTimeout( function () {
    try {
      var obj;
      obj.push( 'string' );
    } catch ( error ) {
      // обработать ошибку
    }
  }), 1);
</script>
```

Необходимость делать это во всех асинхронных вызовах быстро надоедает, к тому же на консоли информация об ошибке все равно не отображается. Кроме того, погружение кода в блок `try/catch` препятствует его предварительной компиляции, а значит, код работает медленнее. Неплохой компромисс для SPA – обернуть наш вызов метода `init` блоком `try/catch`, внутри `catch` вывести информацию об ошибке на консоль и отправить серверу с помощью Ajax, а для перехвата всех асинхронных ошибок использовать обработчик `window.onerror`, в котором также отправлять информацию с помощью Ajax. Выводить сведения об асинхронных ошибках на консоль вручную нет необходимости, потому что они и так там появляются.

```
<script>
  $(function () {
    try {
      spa.initModule( $('#spa') );
    } catch ( error ) {
      // вывести информацию об ошибке на консоль
      // затем отправить ее сторонней службе протоколирования
    }
  });

  window.onerror = function ( error ) {
    // обработать асинхронные ошибки
  };
</script>
```

Теперь мы понимаем, какие ошибки имеют место на стороне клиента, и можем заняться вопросом о быстрой доставке содержимого посетителям сайта.

9.2.3. Сети доставки содержимого

Сеть доставки содержимого (content delivery network – CDN) предназначена для максимально быстрой доставки статических файлов. Это может быть единственный сервер Apache рядом с нашим сервером приложения или распределенная по всему миру инфраструктура с десятками центров. В любом случае для доставки статических файлов лучше выделить отдельный сервер и не возлагать эту задачу на приложение, увеличивая тем самым нагрузку. Node.js вообще очень плохо приспособлен к доставке больших статических файлов (изображений, таблиц стилей CSS, JavaScript-файлов), поскольку при таком использовании не задействуются его асинхронные возможности. Apache с его заранее запущенными дочерними процессами куда лучше.

Поскольку мы хорошо знакомы с Apache, то *могли бы* настроить «CDN-сеть с одним сервером» и использовать ее, пока не будем готовы масштабировать сайт. Но к нашим услугам и многочисленные сторонние CDN. Из наиболее крупных назовем Amazon, Akamai и Edgecast. У Amazon есть продукт Cloudfront, а Akamai и Edgecast предлагают свои услуги через другие компании, например Rackspace, Distribution Cloud и т. д. На самом деле компаний по организации CDN так много, что даже есть сайт, специально предназначенный для выбора подходящего поставщика: www.cdnplanet.com.

Еще одно достоинство глобальной распределенной сети CDN состоит в том, что наше содержимое раздается с ближайшего к посетителю сервера, в результате чего время доставки файла заметно сокращается. Среди различных мер повышения производительности использование CDN – часто самая простая.

9.3. Кэширование и отключение кэширования

Кэширование исключительно важно для быстродействия приложения. Из всех способов получения данных извлечение из клиентского кэша – самый быстрый, а кэширование на стороне сервера часто оказывается гораздо быстрее повторного вычисления одного и того же

ответа снова и снова. Есть много мест, где SPA потенциально может кэшировать данные и тем самым ускорить работу, а именно:

- веб-хранилище;
- HTTP-кэширование;
- кэширование на сервере;
- кэширование запросов к базе данных.

При кэшировании важно помнить об актуальности данных. Мы не хотим отправлять пользователям устаревшие данные, но в то же время хотим отвечать максимально быстро.

9.3.1. Варианты кэширования

У каждого из вышеупомянутых кэшей своя сфера ответственности и свои способы взаимодействия с клиентом для ускорения работы приложения.

- *Веб-хранилище* находится на стороне клиента, доступно приложению и используется для хранения строковых данных. Оно полезно, когда нужно сохранить готовую HTML-разметку, уже построенную на основе данных, полученных от сервера.
- *HTTP-кэширование* – механизм кэширования ответов от сервера на стороне клиента. Чтобы правильно управлять этим видом кэширования, нужно разобраться в большом количестве деталей, но зато потом мы получаем мощные средства кэширования почти даром.
- *Кэширование на сервере* с помощью Memcached или Redis часто применяется для кэширования результатов обработки запросов. Это первый из рассматриваемых видов кэширования, который позволяет сохранять данные для разных пользователей: если один пользователь запросил какие-то данные и они были помещены в кэш, то при следующем запросе тех же самых данных можно будет сэкономить на обращении к базе.
- *Кэширование запросов к базе данных* реализовано на уровне СУБД. Если этот режим включен, то запрос, идентичный ранее полученному, обслуживается из кэша, без повторной выборки данных.

На рис. 9.2 показан типичный цикл запрос–ответ для разных вариантов кэширования. Мы видим, как кэширование на разных уровнях уменьшает время ответа за счет пропуска других этапов цикла. HTTP-кэширование и кэширование запросов к БД реализовать проще всего, обычно для этого нужно лишь задать некоторые конфигурационные параметры. Кэширование в веб-хранилище и на сервере сложнее и требует больше усилий со стороны разработчика.

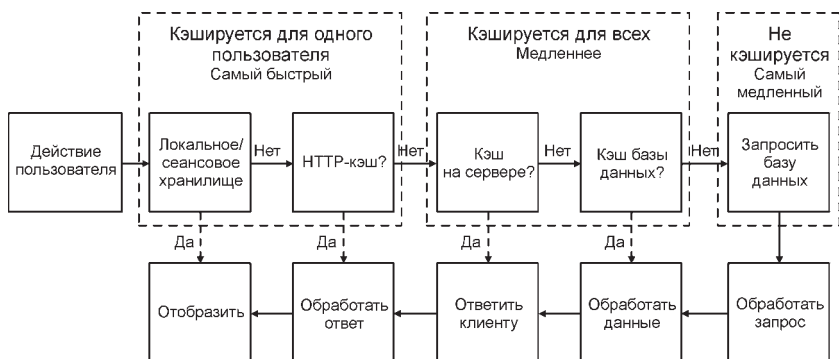


Рис. 9.2 ❖ Пропуск этапов цикла запрос–ответ благодаря кэшированию

9.3.2. Веб-хранилище

Веб-хранилище, известное также под названием *DOM-хранилище*, бывает двух видов: локальное и сеансовое. Оба вида поддерживаются всеми современными браузерами, включая IE8+. Это просто хранилище ключей и значений, в котором и ключ, и значение должны быть строками. В сеансовом хранилище хранятся лишь данные сеанса в текущей вкладке – закрытие вкладки приводит к закрытию сеанса и удалению данных. В локальном хранилище данные кэшируются бессрочно. В любом случае данные доступны только той веб-странице, которая их сохранила. Для SPA это означает, что доступ к хранилищу имеет весь сайт. Один из наиболее эффективных способов использования веб-хранилища – сохранение в нем уже построенных фрагментов HTML-кода, это позволяет пропустить весь цикл запрос–ответ и сразу перейти к отображению результата. Детали показаны на рис. 9.3.

Мы используем локальное хранилище для хранения несекретной информации, которая должна остаться и после завершения текущего сеанса работы с браузером. Сеансовое хранилище мы применяем для хранения данных, имеющих смысл только для текущего сеанса.

Поскольку в веб-хранилище можно хранить только строковые значения, обычно в него помещают данные в формате JSON или HTML-разметку. Сохранение JSON-данных избыточно, если в SPA используется HTTP-кэширование, которое мы будем обсуждать в следующем разделе. К тому же при этом все равно потребуется какая-то обработка. Часто лучше сохранить HTML-строку, чтобы можно было обойтись

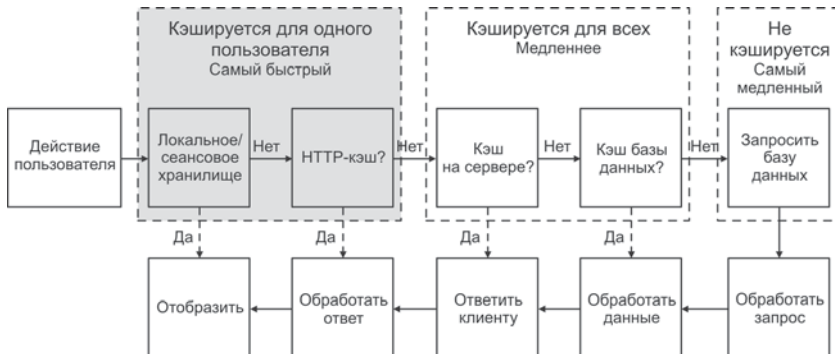


Рис. 9.3 ❖ Веб-хранилище

без ее повторного построения. Такой вид хранилища можно абстрагировать в виде JavaScript-объекта, занимающегося всеми деталями.

В сеансовом хранилище хранятся только данные для текущего сеанса, поэтому иногда – но не всегда, не всегда – можно не придавать большого значения проблеме устаревания данных. Если все-таки устаревание недопустимо, то один из способов принудительно выполнить обновление – вставить в ключ кэша время. Если мы хотим, чтобы данные перезагружались ежедневно, то можем вставить дату, если ежечасно – то дату и час. Этот способ годится не во всех случаях, но с точки зрения выполнения является, пожалуй, самым простым. Он показан в листинге 9.2.

Листинг 9.2 ❖ Добавление времени в ключ кэша

```
SPA.storage = (function () {
```

```
    var generateKey = function ( key ) {
        var date      = new Date(),
            datekey    = new String()
            + date.getYear()
            + date.getMonth()
            + date.getDay();
        return key + datekey;
    };
};
```

```
return {
    'set': function ( key, value ) {
        sessionStorage.setItem( generateKey( key ), value );
    },
```

```
    'get': function ( key ) {
```

Дописываем текущую дату в конец ключа, в результате чего в сеансовом хранилище кэшируются только данные за один день. Этот простой прием гарантирует, что по истечении определенного времени не будут возвращаться кэшированные данные.

Эти методы абстрагируют объект `sessionStorage`, так что впоследствии мы сможем заменить его объектом `localStorage` (или еще чем-нибудь), не изменяя прочего кода. Все методы вызывают функцию `generateKey`, которая дописывает дату в конец ключа, так что нам не нужно делать это при каждом обращении к хранилищу.

```

    return sessionStorage.getItem( generateKey( key ) );
  },

  'remove': function ( key ) {
    sessionStorage.removeItem( generateKey( key ) );
  },

  'clear': function () {
    sessionStorage.clear();
  }
}
})();

```

9.3.3. HTTP-кэширование

HTTP-кэширование имеет место, когда браузер кэширует данные, полученные от сервера, в соответствии с правилами, которые сервер задал в заголовке, или стандартными правилами кэширования, подразумеваемыми по умолчанию. Хотя этот способ, вообще говоря, медленнее веб-хранилища, так как результаты все-таки приходится обрабатывать, зачастую он оказывается гораздо проще и быстрее, чем кэширование на сервере. На рис. 9.4 показано место HTTP-кэширования в цикле запрос–ответ.

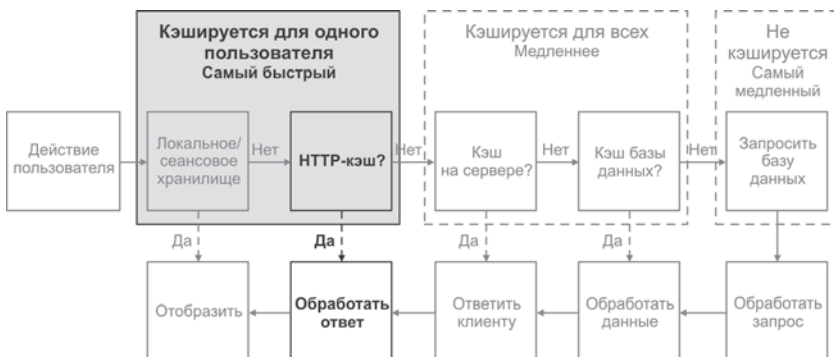


Рис. 9.4 ❖ HTTP-кэширование

HTTP-кэширование применяется для сохранения ответов сервера на стороне клиента, чтобы избежать повторного запроса. У этого подхода есть два варианта:

1. Обслуживать запрос из кэша, не спрашивая сервер об актуальности ответа.

2. Справиться у сервера насчет актуальности и обслуживать из кэша, если ответ еще актуален, или с сервера, если устарел.

Обслуживание напрямую из кэша без проверки актуальности данных быстрее, потому что мы вообще не обращаемся к серверу. Чаще такой режим применяется для изображений, таблиц стилей CSS и JavaScript-файлов, но мы можем настроить приложение так, чтобы и данные тоже кэшировались на протяжении какого-то времени. Например, если приложение обновляет некоторые данные ежедневно в полночь, то мы можем сказать клиенту, что кэшировать их можно только до полуночи.

Иногда кэшированная таким образом информация оказывается недостаточно свежей. Тогда мы можем потребовать, чтобы браузер обращался к серверу с просьбой проверить актуальность данных.

Обратимся к деталям работы этого механизма кэширования. Принцип его действия основан на том, что клиент анализирует заголовки полученного от сервера ответа. Интересуют его в первую очередь три атрибута: `max-age`, `no-cache` и `last-modified`. Все вместе они сообщают клиенту, сколько времени кэшировать данные.

max-age

Чтобы клиент пользовался данными, хранящимися в кэше, не пытаясь обратиться к серверу, первоначальный ответ должен содержать заголовок `Cache-Control`, в котором установлено значение атрибута `max-age`. Это значение говорит клиенту, как долго можно хранить в кэше данные без повторного обращения к серверу. Интервал `max-age` измеряется в секундах. Эта возможность очень удобна, но и потенциально опасна. Удобна – потому что это самый быстрый способ добраться до данных; после первоначальной загрузки данных в кэш приложение будет работать очень быстро. А опасна – потому что клиент не проверяет, изменились ли данные, поэтому мы должны задавать срок хранения очень осторожно.

Express позволяет добавить в ответ заголовок `Cache-Control` с атрибутом `max-age` следующим образом:

```
res.header("Cache-Control", "max-age=28800");
```

После того как данные помещены в кэш таким способом, единственный способ отменить кэширование и заставить клиент отправить новый запрос – изменить имя файла.

Очевидно, что менять имена файлов при каждом выкладывании на производственный сервер нежелательно. К счастью, добиться от-

мены кэширования можно и путем изменения параметров в запросе файла. Обычно для этого добавляют в конец URL номер версии или целое число, которое система сборки увеличивает на единицу при каждом развертывании. Сделать это можно разными способами, но мы предпочитаем заводить отдельный файл, в котором хранится увеличивающееся число, и добавлять это число в конец имени файла. Поскольку индексная страница у нас статическая, то средство развертывания можно настроить так, чтобы оно генерировало HTML-файл и включало номер версии в конец URL-адресов включаемых файлов. В листинге 9.3 показано, как может выглядеть HTML-файл, гарантирующий отмену кэширования.

Листинг 9.3 ❖ Отмена кэширования по max-age

```
<html>
<head>
  <link rel="stylesheet" type="text/css"
        href="/path/to/css/file?version=1.1" />
  <script src="/path/to/js/file?version=1.1"></script>
</head>
<body>

</body>
</html>
```

Наличие номера версии version=1.1 отменяет кэширование

Атрибут `max-age` можно использовать и по-другому; если присвоить ему значение 0, то клиент будет каждый раз подтверждать актуальность данных, то есть запрашивать у сервера, можно ли их использовать. Сервер вправе вернуть ответ 302, означающий, что данные не устарели и клиент может обслужить запрос из кэша. Побочный эффект установки `max-age=0` состоит в том, что промежуточные серверы – находящиеся между конечным сервером и клиентом – вправе возвращать ответ из устаревшего кэша при условии, что поднимают в ответе предупреждающий флаг.

Если же мы хотим, чтобы промежуточные серверы вообще не использовали свой кэш, то должны познакомиться с атрибутом `no-cache`.

no-cache

Согласно спецификации, атрибут `no-cache` работает настолько похоже на `max-age=0`, что может привести к путанице. Он говорит клиенту, что актуальность данных в кэше следует всякий раз подтверждать у сервера, но одновременно является указанием промежуточным серверам не возвращать устаревшее содержимое из кэша, даже сопровождая ответ предупреждением. В последние несколько лет возникла

интересная ситуация, потому что IE и Firefox начали интерпретировать этот режим как полный запрет кэширования данных при любых обстоятельствах. Это означает, что клиент даже не будет спрашивать у сервера, актуальны ли данные, полученные в прошлый раз; он просто не сохраняет их в кэше. В результате ресурсы, сопровождаемые заголовком **no-cache**, могут обслуживаться медленно – без всякой на то необходимости. Если действительно требуется запретить клиентам кэшировать ресурс, то следует использовать атрибут **no-store**.

no-store

Атрибут **no-store** означает, что ни клиенты, ни промежуточные серверы вообще не должны сохранять информацию об этой паре запрос–ответ в своем кэше. Хотя этот режим в какой-то мере повышает безопасность передачи данных, его ни в коем случае нельзя считать идеальной формой защиты. В правильно реализованных системах стираются все следы данных, но всегда есть шанс, что данные проходили через системы, написанные некорректно – случайно или намеренно – и уязвимые к подслушиванию.

last-modified

Если заголовок **Cache-Control** отсутствует, то для определения срока хранения данных в кэше клиент использует алгоритм, основанный на дате, указанной в заголовке **last-modified**. Обычно это треть от промежутка времени, прошедшего с даты последней модификации (**last-modified**). Так, если в момент запроса выясняется, что графический файл был модифицирован три дня назад, то клиент в следующий раз обратится к серверу только через день, а до тех пор будет обслуживать файл из кэша. Поэтому время хранения ресурса в кэше в значительной степени случайно и зависит от того, когда файл в последний раз был скопирован на производственную систему.

Есть еще много атрибутов, относящихся к кэшированию, но уже понимание этих основных поможет существенно повысить скорость загрузки приложения. HTTP-кэширование позволяет клиентам нашего приложения обслуживать запросы на ресурсы, которые они уже раньше видели, не обращаясь снова к серверу, или с минимальной задержкой, необходимой, чтобы поинтересоваться у сервера актуальностью данных. Это ускоряет реакцию данного клиента при последующих запросах, но что будет с другими клиентами, которые обращаются с точно такими же запросами? Тут HTTP-кэширование – не помощник, данные необходимо кэшировать на стороне сервера.

9.3.4. Кэширование на сервере

Для сервера самый быстрый способ обслужить динамический запрос от клиента – взять данные из своего кэша. Это позволяет не тратить времени на выборку данных из базы и последующее их преобразование в формат JSON. На рис. 9.5 показано место кэширования на сервере в цикле запрос–ответ.

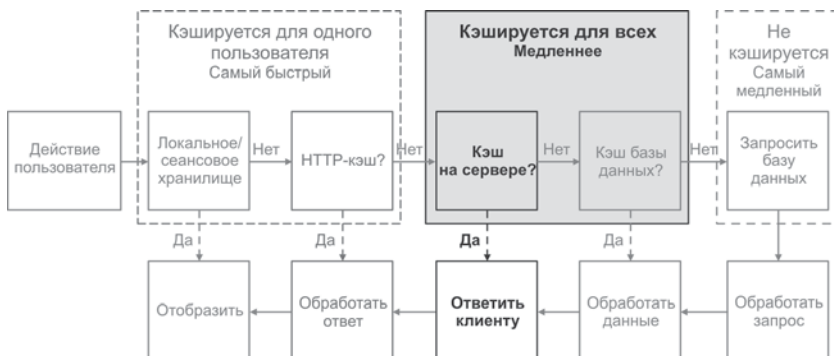


Рис. 9.5 ❖ Кэширование на сервере

Есть два популярных продукта для кэширования данных на сервере: Memcached и Redis. Согласно сайту memcached.org, «Memcached – это организованное в памяти хранилище ключей и значений, которое предназначено для хранения небольших фрагментов данных произвольного вида». Оно специально проектировалось как временный кэш для данных, извлеченных из базы, результатов вызовов API или уже построенных кусков HTML. Если у сервера заканчивается память, он автоматически начинает вытеснять из кэша данные, которые давно не использовались (алгоритм LRU). Redis – это *хранилище ключей и значений с дополнительными возможностями*, которое используется для хранения более сложных структур данных: строк, хэшей, списков, множеств и отсортированных множеств.

Общая идея кэширования – уменьшить нагрузку на сервер и время реакции. Когда поступает запрос, приложение сначала проверяет, есть ли уже ответ на него в кэше. Если ответ имеется, то приложение сразу возвращает его клиенту. В противном случае сначала выполняется сравнительно дорогая выборка из базы данных, и результат преобразуется в формат JSON. Результат преобразования сохраняется в кэше и возвращается клиенту.

При использовании кэша мы должны продумать вопрос о том, когда следует прекратить кэширование. Если в кэш пишет только наше приложение, то оно может очистить или регенерировать кэш при изменении данных. Если же в кэш пишут и другие приложения, то они также должны обновлять кэш. Решить эту проблему можно несколькими способами.

1. Можно делать кэш недействительным по истечении заданного промежутка времени и принудительно регенерировать данные. Если делать это один раз в час, то в течение суток мы получим всего 24 ответа, обслуженных не из кэша. Очевидно, это подходит не для всех приложений.
2. Можно сравнивать время последнего обновления данных с временной меткой данных в кэше. Это дольше, чем первое решение, но, возможно, все-таки не так долго, как выполнение полного запроса, зато мы гарантированно получаем актуальные данные.

Какой вариант выбрать, зависит от потребностей приложения. Для нашего SPA кэширование на сервере – вообще излишество. MongoDB демонстрирует великолепную производительность на нашем наборе данных. И полученный от MongoDB ответ мы никак не преобразовываем, а просто передаем напрямую клиенту.

Так когда же стоит задуматься о включении в веб-приложение кэширования на сервере? Когда обнаруживается, что база данных или веб-сервер – узкое место. Обычно кэширование позволяет снизить нагрузку и на сервер, и на базу данных, а значит, уменьшить время реакции. Безусловно, этот вариант стоит попробовать, прежде чем покупать новый дорогой сервер. Однако не забывайте, что для кэширования на сервере необходима еще одна служба (типа Memcached или Redis), требующая мониторинга и обслуживания, а это увеличивает сложность приложения.

В экосистеме Node.js имеются драйверы для Memcached и Redis. Давайте добавим в наше приложение Redis и воспользуемся им для кэширования данных о пользователях. Инструкции по установке Redis имеются на сайте <http://redis.io>. Установив и запустив Redis, мы можем проверить его работоспособность, открыв оболочку командой `redis-cli`.

Добавим драйвер Redis в манифест `npm`, как показано в листинге 9.4.

Листинг 9.4 ❖ Включение Redis в манифест `npm` – `webapp/package.json`

```
{ "name" : "SPA",  
  "version" : "0.0.3",  
  "private" : true,  
  "dependencies" : {
```



```

    "express" : "3.2.x",
    "mongodb" : "1.3.x",
    "socket.io" : "0.9.x",
    "JSV" : "4.0.x",
    "redis" : "0.8.x"
  }
}

```

Прежде чем приступить к делу, подумаем, что мы должны уметь делать с кэшем. На ум приходят две вещи: *установка* пары ключ–значение и *получение* из кэша значения по ключу. Еще нам, наверное, будет полезно *удаление* ключа из кэша. Имея это в виду, создадим в каталоге lib файл cache.js и включим в него методы установки, получения и удаления в соответствии с паттерном модуля. В листинге 9.5 показано, как подключиться к Redis из Node.js, и приведены заготовки методов.

Листинг 9.5 ❖ Создание кэша в Redis – webapp/lib/cache.js

```

/*
 * cache.js - реализация кэша с помощью Redis
 */
/*jshint
  node : true, continue : true,
  devel : true, indent : 2, maxerr : 50,
  newcap : true, nomen : true, plusplus : true,
  regexp : true, sloppy : true, vars : false,
  white : true
*/
/*global */

// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  redisDriver = require( 'redis' ),
  redisClient = redisDriver.createClient(),
  makeString, deleteKey, getValue, setValue;
updateObj, destroyObj;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
deleteKey = function ( key ) {};

getValue = function ( key, hit_callback, miss_callback ) {};

setValue = function ( key, value ) {};

module.exports = {
  deleteKey : deleteKey,
  getValue : getValue,
  setValue : setValue
};
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Теперь начнем реализовывать эти методы, в окончательном виде они приведены в листинге 9.6. Сначала напишем метод `setValue`, потому что он самый простой. Redis поддерживает много разных типов данных; какой из них выбрать, зависит от того, что мы кэшируем. В данном случае остановимся на простейшем варианте, когда и ключ, и значение – строки. С помощью драйвера Redis для установки значения достаточно вызвать метод `redis.set(key, value);`. Мы не задали функцию обратного вызова, потому что считаем, что метод завершается успешно; он будет работать асинхронно, а если возникнет ошибка, то сведения о ней будут отброшены. Можно было бы сделать что-то более интересное, например увеличивать счетчик ошибок на единицу. Призываем любопытных читателей исследовать эту возможность.

Метод `getValue` принимает три аргумента: искомый ключ `key`, функция `hit_callback`, вызываемая в случае, когда ключ найден (попадание кэша), и функция `miss_callback`, вызываемая в случае, когда ключ не найден (промах кэша). Этот метод просит Redis вернуть значение, ассоциированное с ключом. В случае попадания (значение не равно `null`) вызывается функция `hit_callback`, которой найденное значение передается в качестве аргумента. В случае промаха (значение равно `null`) вызывается функция `miss_callback`. Выборка из базы данных возлагается на вызывающую программу, а этот код занимается только кэшированием.

Метод `deleteKey` вызывает `redis.del`, передавая ему ключ кэша Redis. Функцию обратного вызова мы не задаем, потому что предполагаем, что метод работает без ошибок.

Служебная функция `makeString` служит для преобразования ключей и значений до передачи их Redis. Это необходимо, потому что иначе драйвер Redis вызвал бы для ключей и значений метод `toString()`, и в результате мы получили бы строку вида `[Object object]` – очевидно, не то, что нам нужно.

В листинге 9.6 приведен модифицированный модуль `cache`. Изменения выделены **полужирным** шрифтом.

Листинг 9.6 ❖ Окончательный модуль кэширования с помощью Redis – `webapp/lib/cache.js`

```
/*
 * cache.js – реализация кэша с помощью Redis
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
  redisDriver = require( 'redis' ),
```

```

redisClient = redisDriver.createClient(),
makeString, deleteKey, getValue, setValue;
updateObj, destroyObj;
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

// ----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
makeString = function ( key_data ) {
    return (typeof key_data === 'string' )
        ? key_data
        : JSON.stringify( key_data );
};
// ----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
deleteKey = function ( key ) {
    redisClient.del( makeString( key ) );
};

getValue = function ( key, hit_callback, miss_callback ) {
    redisClient.get(
        makeString( key ),
        function( err, reply ) {
            if ( reply ) {
                console.log( 'HIT' );
                hit_callback( reply );
            }
            else {
                console.log( 'MISS' );
                miss_callback();
            }
        }
    );
};

setValue = function ( key, value ) {
    redisClient.set(
        makeString( key ), makeString( value )
    );
};

module.exports = {
    deleteKey : deleteKey,
    getValue  : getValue,
    setValue  : setValue
};
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----

```

Метод makeString преобразует объект в JSON-строку; иначе драйвер Redis вызвал бы метод toString() для входных данных, что привело бы к созданию ключа [Object object] – не слишком полезного.

Метод delete вызывает метод Redis del, который удаляет ключ и ассоциированное с ним значение.

Метод getValue принимает ключ и две функции обратного вызова. Первая функция вызывается, если ключ найден в кэше, вторая – если не найден.

Метод setValue вызывает метод Redis set, который сохраняет строку. В Redis применяются разные команды сохранения в зависимости от типа сохраняемого объекта; эта система умеет хранить не только строки, что делает ее пригодной для реализации гибкого кэша.

Закончив с модулем кэширования, мы можем воспользоваться им в файле crud.js, для чего нужно добавить пять строк, выделенных в листинге 9.7 **полужирным** шрифтом.

Листинг 9.7 ❖ Чтение из кэша – `–webapp/lib/crud.js`

```

/*
 * crud.js – модуль, предоставляющий операции CRUD с базой данных
 */
...
// ----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
'use strict';
var
...
  JSV      = require( 'JSV' ).JSV,
  cache    = require( './cache' ), ← Включаем модуль cache в модуль CRUD.

  mongoServer = new mongodb.Server(
...
// ----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----
...

// ----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
...

readObj = function ( obj_type, find_map, fields_map, callback ) {
  var type_check_map = checkType( obj_type );
  if ( type_check_map ) {
    callback( type_check_map );
    return;
  }

  В качестве ключа используем find_map.
  cache.getValue( find_map, callback, function () { ←
    dbHandle.collection(
      obj_type,
      function ( outer_error, collection ) {
        collection.find( find_map, fields_map ).toArray(
          function ( inner_error, map_list ) {
            cache.setValue( find_map, map_list ); ←
            callback( map_list );
          }
        );
      }
    );
  }
  );
}); ← Конец вызова cache.getValue.
};
...

destroyObj = function ( obj_type, find_map, callback ) {
  var type_check_map = checkType( obj_type );
  if ( type_check_map ) {
    callback( type_check_map );
    return;
  }
}

```

Добавляем вызов `cache.getValue`, а прежнее обращение к `mongo` делаем функцией, которая вызывается в случае промаха кэша.

Добавляем запись в кэш с помощью `cache.setValue` в случае промаха кэша.

```

cache.deleteKey( find_map ); ← При удалении объекта из базы данных удаляем и
dbHandle.collection(                                     ключ из Redis с помощью cache.deleteKey.
  obj_type,
  function ( outer_error, collection ) {
    var options_map = { safe: true, single: true };
    collection.remove( find_map, options_map,
      function ( inner_error, delete_count ) {
        callback({ delete_count: delete_count });
      }
    );
  }
);
};
...
// ----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
...

```

Мы удаляем ключ из кэша Redis при удалении объекта из базы. Но это решение далеко от идеала. Оно не гарантирует, что удалены все экземпляры кэшированных данных; удаляются лишь кэшированные данные, ассоциированные с *ключом, который использовался для удаления объекта*. Мы можем, например, удалить только что уволенного работника *по идентификатору*, но он все равно сможет войти в систему и все в ней испортить, потому что информация, возможно, была кэширована по ключу *имя–пароль*. Та же проблема может возникнуть при обновлении объекта.

Это непростая проблема. В частности, по этой причине реализацию кэширования на сервере часто откладывают до тех пор, пока не возникнет необходимость вкладывать деньги в масштабирование системы. Наметим возможные решения: задавать срок хранения записей в кэше и удалять их после его окончания (уменьшает продолжительность окна рассогласования), очищать кэш пользователей целиком при удалении или обновлении любого пользователя (более безопасно, но увеличивает количество промахов кэша), вручную вести учет кэшированными объектам (чревато ошибками программирования).

У кэширования на сервере есть еще много возможностей и столько же проблем – хватило бы на отдельную книгу, – но мы надеемся, что для начала изложенного достаточно. Бросим теперь взгляд на последний метод кэширования: запросов к базе данных.

9.3.5. Кэширование запросов к базе данных

Под *кэшированием запросов* понимается сохранение результатов выполнения запросов на уровне самой СУБД. Это особенно важно в реляционных базах данных из-за необходимости преобразовывать ре-

зультаты в форму, понятную приложению. В кэше запросов хранится уже преобразованный результат. На рис. 9.6 показано место кэширования запросов к БД в цикле запрос–ответ.

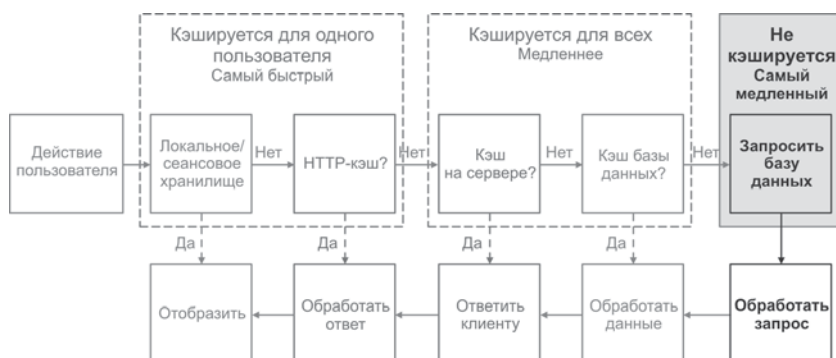


Рис. 9.6 ❖ Кэширование запросов к базе данных

В случае MongoDB проблема автоматически решается файловой системой ОС. Вместо того чтобы кэшировать результаты конкретного запроса, MongoDB пытается держать весь индекс в памяти, что приводит к исключительно быстрому выполнению запросов в тех случаях, когда весь набор данных уместается в памяти. MongoDB, а точнее подсистема управления памятью ОС, динамически выделяет память, исходя из потребностей сервера. Это означает, что в распоряжении MongoDB имеется вся свободная оперативная память, так что ей не приходится гадать, сколько памяти выделить. Если же память понадобится другим процессам, она автоматически освобождается. Поведение кэширования, в частности работа алгоритмов вытеснения по давности, определяется поведением операционной системы.

9.4. Резюме

В этой главе мы ответили на ряд типичных вопросов, возникающих при разворачивании SPA. Мы показали, как подготовить SPA к индексированию поисковой системой, как использовать аналитические средства (типа Google Analytics) и как передать информацию об ошибках приложения на сервер. Наконец, мы обсудили кэширование на всех уровнях приложения, объяснили, какую практическую пользу приносит кэширование на каждом уровне и как его реализовать.

Наши рекомендации по созданию надежных, тестопригодных и масштабируемых SPA почти подошли к концу. Мы настоятельно советуем прочитать приложения А и Б, потому что в них рассматриваются важные вопросы. В приложении А описываются стандарты кодирования, которые мы применяли в этой книге, а в приложении Б – использование режимов тестирования и автоматизации для обнаружения, изоляции и исправления ошибок в программе.

В части I этой книги мы реализовали наше первое SPA и обсудили, почему SPA так хороши для многих сайтов. В частности, SPA позволяют обеспечить такие скорость реакции и уровень интерактивности, какие и не снились традиционному сайту. Затем мы поговорили о некоторых концепциях программирования на JavaScript, которые необходимо отчетливо понимать для создания крупномасштабных SPA.

В части II мы продолжили проектировать и реализовывать SPA на основе проверенной на практике архитектуры. Мы не пользовались готовым «каркасом», потому что хотели продемонстрировать внутреннее устройство SPA. Вы можете положить в основу собственного SPA эту архитектуру, а можете приняться за изучение какой-нибудь из многочисленных библиотек – только теперь у вас имеется опыт, позволяющий судить, обладает библиотека нужными вам средствами или нет.

В части III мы настроили Node.js и сервер MongoDB, которыми воспользовались для реализации серверной части нашего SPA, в частности операций CRUD. Для организации быстрой полнодуплексной связи между клиентом и сервером мы использовали библиотеку Socket.IO. Нам также удалось обойтись без преобразования данных из одного формата в другой, столь характерного для традиционных сайтов.

И в конце концов у нас получился полный стек приложения, в котором в качестве языка используется JavaScript, а в качестве формата данных – JSON. Эта элегантная простота обладает достоинствами, проявляющимися на каждом шаге процесса разработки. Например, применение единого языка позволяет использовать общий код на стороне клиента и сервера, что существенно снижает размер и сложность программы. Заодно это экономит время и устраняет неразбериху, так как не приходится мысленно переключаться между разными языками и форматами данных. Преимущества находят выражение и на этапе тестирования: мало того что сокращается общий объем подлежащего тестированию кода, так еще мы можем почти везде пользоваться единой системой тестирования без накладных расходов и неудобств, связанных с применением браузера.

Мы надеемся, что вам понравилась эта книга и что, читая ее, вы узнали не меньше, чем мы, когда ее писали. Самый лучший способ продолжить изучение одностраничных приложений – разрабатывать их. Мы, как могли, старались познакомить вас с инструментами, которые могут понадобиться для программирования всех компонентов на JavaScript.

Приложение А

Стандарт кодирования на JavaScript

В этом приложении:

- ✧ Почему важен стандарт кодирования.
- ✧ Единообразное форматирование и документирование кода.
- ✧ Единообразное именование переменных.
- ✧ Изолирование кода с помощью пространств имен.
- ✧ Организация файлов и следование единообразному синтаксису.
- ✧ Валидация кода с помощью JSLint.
- ✧ Использование шаблона, воплощающего стандарт.

Стандарты кодирования – предмет вечных споров. Все согласны, что стандарт должен быть, но каждый представляет его по-своему. Поговорим о том, почему для JavaScript стандарт особенно важен.

А.1. Зачем нам стандарт кодирования?

Наличие четкого стандарта для слаботипизированного динамического языка, каким является JavaScript, почти наверняка важнее, чем для более строгих языков. Сама гибкость JavaScript может оказаться ящиком Пандоры, скрывающим кошмарные формы синтаксиса и безумные приемы кодирования. Если более строгие языки сами навязывают структуру и систематичность, то для достижения того же эффекта при программировании на JavaScript необходимы дисциплина и применение стандартов.

Ниже описывается стандарт, который мы использовали и шлифовали на протяжении многих лет. Он достаточно полный и логичный, и в этой книге мы последовательно придерживались его. Здесь мы не стремились к предельно сжатому изложению, а, напротив, добавили много примеров и объяснений. Но существо стандарта уложено в трехстраничную шпаргалку по адресу <https://github.com/mmikowski/spa>.

Мы не настолько самонадеянны, чтобы считать этот стандарт кодирования подходящим для всех и каждого; хотите – берите его на

вооружение, не хотите – игнорируйте. Но в любом случае мы надеемся, что изложенные здесь соображения побудят вас выработать собственный стиль. Мы настоятельно рекомендуем любой команде сначала прийти к согласию относительно стандарта и только потом приниматься за крупный проект – иначе она рискует разделить участь строителей Вавилонской башни.

Собственный опыт и объективные исследования говорят, что на сопровождение кода уходит больше времени, чем на его написание. Поэтому наш стандарт отдает предпочтение не скорости кодирования, а удобочитаемости. Мы считаем, что если код писался так, чтобы быть понятным, то, скорее всего, он окажется и более тщательно продуманным, и лучше структурированным.

Наша практика показала, что успешный стандарт кодирования должен обладать следующими характеристиками:

- минимизировать вероятность появления ошибок кодирования;
- порождать код, пригодный для крупномасштабных проектов и коллективной разработки, – единообразный, удобочитаемый, расширяемый и пригодный для сопровождения;
- способствовать эффективности, практичности и возможности повторного использования кода;
- поощрять использование сильных сторон JavaScript и препятствовать использованию его слабостей;
- применяться всеми членами команды разработчиков.

Мартин Фаулер однажды произнес ставшую знаменитой фразу: «Любой дурак может написать код, который будет понятен компьютеру. Хорошие программисты пишут код, понятный другим людям». И хотя даже самый четкий и полный стандарт не гарантирует, что код на JavaScript будет удобочитаемым, он все же может в этом помочь – как словари и грамматические справочники помогают писать текст на понятном человеку естественном языке.

А.2. Форматирование кода и комментарии

Единообразное и продуманное форматирование кода – один из лучших способов сделать его более понятным. И один из вопросов, вызывающих больше всего споров¹. Поэтому, читая данный раздел,

¹ Легионы разработчиков потратили бесчисленные часы в неистовых спорах только о том, как правильно использовать знаки табуляции. Если не верить, задайте поисковику запрос «tabs versus spaces».

расслабьтесь. Налейте чашечку латте без кофеина, опустите ноги в тазик с экстрактом чайного листа и мяты и постарайтесь сохранять непредубежденность. Будет интересно. Правда.

A.2.1. Форматирование кода с учетом удобства чтения

Что было бы, если бы из этой книги были убраны все заголовки, знаки препинания, разрядка и заглавные буквы? Наверное, книга вышла бы на несколько месяцев раньше, но читатели сочли бы ее совершенно неразборчивой. Быть может, именно потому наш редактор настаивал на соблюдении соглашений о форматировании, чтобы у тебя, любезный читатель, появился шанс разобраться, что здесь написано.

JavaScript-код должен быть понятен двум разным аудиториям – компьютерам, которые его выполняют, и людям, которые его сопровождают и развивают. Обычно глазам читателя код предстает гораздо больше раз, чем глазам его автора. Мы форматируем код в соответствии с принятыми соглашениями, для того чтобы дать коллегам-разработчикам (и, кстати, нам самим спустя несколько недель) шанс разобраться в нем.

Единообразные отступы и длины строк

Вероятно, от вашего внимания не укрылось, что ширина газетной колонки составляет от 50 до 80 знаков. Человеческому глазу трудно пробежать по строке, содержащей больше 80 знаков, и тем труднее, чем она длиннее. В авторитетной книге Bringham «The Elements of Typographic Style»¹ утверждается, что для комфортного чтения и понимания прочитанного в строке должно быть 45–75 знаков, причем оптимальная длина равна 66.

Длинные строки трудно читать и на экране монитора. В наши дни все больше веб-страниц верстаются в несколько колонок, хотя известно, что корректно реализовать это трудно. И пойти на такие хлопоты веб-дизайнер согласится, только если длинные строки вызывают проблемы (или если ему платят повременно).

Сторонники широкой табуляции (4–8 пробелов) говорят, что при этом код становится более разборчивым. Но они же выступают и в поддержку длинных строк как неизбежного следствия широкой табуляции. Наш подход иной: при узкой табуляции (2 пробела) и более коротких строках (78 знаков) документ получается уже и разборчи-

¹ Брингхерст. Основы стиля в типографике. – М.: Д. Аронов, 2006. – *Прим. перев.*

вее, а в каждой строке умещается больше полезной информации. Отдавая предпочтение узкой табуляции, мы также учитываем тот факт, что в событийно-управляемом языке типа JavaScript обычно гораздо больше отступов, чем в процедурном, – из-за обилия обратных вызовов и замыканий.

- Обозначайте каждый уровень вложенности кода **двумя пробелами**.
- **Пользуйтесь пробелами, а не знаками табуляции**, потому что позиции точек табуляции никак не стандартизованы.
- **Ограничивайте длину строки 78 знаками**.

Более узкий документ хорошо смотрится на любом экране. На двух мониторах с высоким разрешением можно будет одновременно открыть шесть файлов, но и на устройстве с небольшим экраном – ноутбуке, планшете или смартфоне – один документ прочитать будет легко. Узкий документ умещается и на экране электронной книги, и на печатной странице, что очень радует редактора¹.

Разбивайте код на абзацы

Тексты на русском и других естественных языках разбиваются на абзацы, чтобы читателю было понятно, где заканчивается одна смысловая единица и начинается следующая. Это соглашение не менее полезно и в языках программирования. Абзацы можно аннотировать как единое целое. При надлежащем использовании разрядки² JavaScript-код читается как хорошо отформатированная книга.

- **Организируйте код в виде совокупности логических абзацев**, отделяя их друг от друга пустыми строками.
- **В каждой строке должно быть не более одного предложения или присваивания**, хотя мы допускаем объявление нескольких переменных в одной строке.
- **Отделяйте операторы от переменных пробелами**, чтобы переменные отчетливо выделялись.
- **Ставьте пробелы после запятых**.
- **Выравнивайте однотипные операторы** внутри абзацев.
- **Располагайте комментарии с таким же отступом**, как код, к которому они относятся.

¹ В этой книге длина строки во всех листингах не превышает 72 знаков, и потеря шести знаков далась нам с большим трудом.

² Под разрядкой мы понимаем произвольную комбинацию пробелов, знаков табуляции и переходов на новую строку. Но знаками табуляции лучше не пользоваться.

- После каждого предложения ставьте точку с запятой.
- Закрывайте все предложения внутри управляющей конструкции в фигурные скобки. К управляющим конструкциям относятся, в частности, `for`, `if`, `while` и т. п. Пожалуй, самое распространенное нарушение этого правила – опускание скобок вокруг одиночного предложения внутри `if`. Не поступайте так. Всегда ставьте скобки, тогда при добавлении предложений в блок не возникнет случайных ошибок.

Листинг А.1 ❖ Так делать не надо

```
// инициализируем переменные
var first_name='sally';var rot_delta=1;
var x_delta=1;var y_delta=1; var coef=1;
var first_name = 'sally', x, y, r, print_msg, get_random;
// помещаем важный текст внутрь div с идентификатором sl_foo
print_msg = function ( msg_text ) {
// .text() предотвращает xss-атаку
$('#sl').text( msg_text )
};
// получаем случайное число
get_random = function ( num_arg ){
    return Math.random() * num_arg;
};
// инициализируем координаты
x=get_random( 10 );
y=get_random( 20 );
r=get_random( 360 );
// корректируем координаты
x+=x_delta*coef;
y+=y_delta*coef;
r+=rot_delta*coef;
if ( first_name === 'sally' ) print_msg('Hello Sally!')
```

Этот комментарий объясняет и без того очевидное.

В одной строке не должно быть несколько присваиваний.

Этот комментарий легко может стать неактуальным.

Комментарий следует располагать с таким же отступом, как код, к которому он относится.

Это предложение не завершается точкой с запятой.

Комментарий трудно визуально выделить из окружающего текста.

Эти уравнения плохо читаются.

Предложения в теле любого `if` должны быть заключены в фигурные скобки.

Листинг А.2 ❖ А надо так

```
var
    x, y, r, print_msg, get_random,
    coef    = 0.5,
    rot_delta = 1,
    x_delta  = 1,
    y_delta  = 1,
    first_name = 'sally'
;

// эта функция записывает текст в контейнер сообщений
print_msg = function ( msg_text ) {
    // .text() предотвращает xss-атаку
```

Удаляем очевидный комментарий.

В одной строке может быть несколько объявлений, но только одно присваивание.

Вставляем пустую строку перед следующим абзацем. Изменяем комментарий, так чтобы они описывал абзац.

Располагаем комментарий с таким же отступом, как код, к которому он относится.

```

$('#sl').text( msg_text );
};

// эта функция возвращает случайное число
get_random = function ( num_arg ) {
    return Math.random() * num_arg;
};

// инициализируем координаты
x = get_random( 10 );
y = get_random( 20 );
r = get_random( 360 );

// корректируем координаты
x += x_delta * coef;
y += y_delta * coef;
r += rot_delta * coef;
if ( first_name === 'sally' ) { print_msg('Hello Sally!'); }

```

Добавляем отсутствующую точку с запятой. Каждое предложение должно завершаться точкой с запятой.

Вставляем пустую строку перед следующим абзацем. Изменяем комментарий, так чтобы он описывал абзац.

Добавляем еще один абзац. Благодаря абзацам комментарии становятся значительно нагляднее.

Добавляем пробелы и выравниваем однотипные элементы, чтобы похожие предложения было проще воспринимать.

Во всех предложениях if и других управляющих конструкциях ставим фигурные скобки.

Целью форматирования является ясность кода, а не уменьшение количества байтов. Готовый JavaScript-код на пути к пользователям будет подвергнут конкатенации, минимизации и сжатию. Поэтому средства, призванные сделать программу понятной, – разрядка, комментарии и осмысленные имена переменных – почти никак не сказываются на производительности.

Единообразное разбиение строк

Если количество знаков в предложении не слишком велико, то его следует располагать в одной строке. Но часто это невозможно, поэтому мы разбиваем предложение на несколько строк. Следующие рекомендации помогут избежать при этом ошибок и не ухудшить понятности кода.

- **Разбивайте строку до операторов**, чтобы читатель знал, что операторы всегда находятся в левом столбце.
- **Все строки, занятые одним предложением, кроме первой**, располагайте с отступом на один уровень – в нашем случае отступ составляет два пробела.
- **Делайте разрыв строки после оператора «запятая»**.
- **Закрывающую скобку размещайте в отдельной строке**. Это наглядно показывает, где конец предложения, так что читателю не приходится бегать глазами в поисках точки с запятой.

Листинг А.3 ❖ Так делать не надо

```

long_quote = 'Восемь десятков и семь лет назад наши
    'отцы образовали на этом континенте новую
    'нацию, зачатую в свободе ' +

```

Когда концы строк не выровнены, легко не заметить, что в конце пропущен знак '+'.

```

    'и верящую в то, что ' +
    'все люди рождены равными.';

```

У расположения запятой в начале строки есть свои плюсы, но это не наш стандарт.

```

cat_breed_list = [ 'Абиссинская', 'Американский бобтейл',
    'Американский керл', 'Американская короткошерстная',
    'Американская жесткошерстная', 'Балийская',
    'Балийско-яванская', 'Бирманская', 'Бомбейская' ];

```

Где заканчивается предложение? Приходится искать глазами точку с запятой.

Листинг А.4 ❖ А надо так

```

long_quote = 'Восемь десятков и семь лет назад наши '
    + 'отцы образовали на этом континенте новую '
    + 'нацию, зачатую в свободе '
    + 'и верящую в то, что '
    + 'все люди рождены равными.';

```

Выравниваем операторы по левому краю.

```

cat_breed_list = [
    'Абиссинская',           'Американский бобтейл',
    'Американский керл',     'Американская короткошерстная',
    'Американская жесткошерстная', 'Балийская',
    'Балийско-яванская',     'Бирманская',
    'Бомбейская'
];

```

Запятой в конце строки упрощают сопровождение кода.

Закрывающая квадратная скобка занимает отдельную строку. Так проще понять, где начинается следующее предложение.

Чуть позже в этом приложении мы покажем, как установить программу JSLint, которая помогает проверять правильность синтаксиса.

Придерживайтесь стиля K&R при расстановке скобок

*Стиль расстановки скобок K&R*¹ – это компромисс между компактностью по вертикали и удобочитаемостью. Так следует форматировать объекты и хэши, массивы, составные предложения и вызовы. Составным называется предложение, которое содержит одно или несколько предложений, заключенных в фигурные скобки, например: `if`, `while` и `for`. Под «вызовом» понимается вызов функции или метода вида `alert('Меня вызвали!');`.

- По возможности **ограничивайтесь одной строкой**. Так, не стоит размещать объявление короткого массива в трех строках, если можно обойтись одной.
- **Ставьте открывающую скобку** – круглую, квадратную или фигурную – в конце первой из нескольких строк.
- **Внутри ограничителей (скобок) делайте отступ** на один уровень, например два пробела.

¹ Kernighan and Ritchie. Стиль форматирования, предложенный Керниганом и Ричи в книге «Язык программирования Си». – *Прим. перев.*

Ставьте закрывающую скобку – круглую, квадратную или фигурную – в отдельной строке с таким же отступом, как в первой из нескольких строк.

Листинг А.5 ❖ Так делать не надо

```
var invocation_count, full_name, top_fruit_list,
    full_fruit_list, print_string;

invocation_count = 2;
full_name = 'Фред Бернс';
top_fruit_list = ← Чрезмерно длинно и разреженно.
[
    'Апельсин',
    'Банан',
    'Яблоко'
];

full_fruit_list = ← Вообще ничего не разберешь! Попробуйте-ка
                    найти глазами нужный фрукт.
[ 'Абрикос', 'Агли', 'Ананас', 'Апельсин',
  'Банан', 'Виноград', 'Вишня', 'Грейпфрут',
  'Груша', 'Гуаява', 'Дыня', 'Ежевика',
  'Киви', 'Клубника', 'Кумкват', 'Лайм',
  'Лимон', 'Личи', 'Малина', 'Манго',
  'Мандарин', 'Нектарин', 'Персик',
  'Смородина', 'Финик', 'Черника', 'Яблоко'
];

print_string = function ( text_arg )
{ ← Расстановка скобок в стиле GNU приводит
  var char_list = text_arg.split(''), i;
  к увеличению количества страниц.

  for ( i = 0; i < char_list.length; i++ )
  {
      document.write( char_list[i] );
  }
  return true;
};

print_string( 'Мы насчитали '
+ String( invocation_count )
+ ' вызовов!'
);
```

Листинг А.6 ❖ А надо так

```
var
    run_count,      full_name,    top_fruit_list,
    full_fruit_list, print_string;

run_count = 2;
```



```

full_name = 'Фред Бернс';

top_fruit_list = [ 'Апельсин', 'Банан', 'Яблоко' ]; ← Все умещается в одной строке.

full_fruit_list = ← Выравнивание по вертикали творит чудеса в плане понятности.
[ 'Абрикос',    'Агли',      'Ананас',    'Апельсин',
  'Банан',      'Виноград',  'Вишня',     'Грейпфрут',
  'Груша',      'Гуаява',    'Дыня',     'Ежевика',
  'Киви',       'Клубника',  'Кумкват',   'Лайм',
  'Лимон',      'Личи',     'Малина',   'Манго',
  'Мандарин',   'Нектарин', 'Персик',    'Смородина',
  'Финик',     'Черника',  'Яблоко'
];

print_string = function ( text_arg ) { ← В стиле K&R открывающая скобка
    var char_list, i;                  ставится в конце строки.

    char_list = input_text.split('');

    for ( i = 0; i < char_list.length; i++ ) {
        document.write( char_list[i] );
    }
    return true;
};

print_string( 'Мы насчитали '
+ String( invocation_count )
+ ' вызовов!
);

```

Выравнивание элементов по вертикали очень способствует пониманию, но без мощного текстового редактора может отнимать много времени. Решению данной задачи помогает функция выделения вертикального блока, которая имеется, например, в редакторах Vim, Sublime, WebStorm и др. WebStorm даже позволяет автоматически выравнивать значения в хэше, что здорово экономит время. Если ваш редактор не поддерживает выделения по вертикали, настоятельно рекомендуем перейти на какой-нибудь другой.

Использование разрядки для выделения функций и ключевых слов

Во многих языках есть понятие артикля – в английском это *an*, *a* и *the*. Одна из функций артикля – сообщить читателю или слушателю, что следующее слово является именем существительным или именным словосочетанием. В случае функций и ключевых слов для той же цели вполне годится разрядка.

- После имени функции ставьте левую круглую скобку без пробела.
- После ключевого слова ставьте один пробел, а за ним левую круглую скобку.
- В предложении `for` ставьте один пробел после каждой точки с запятой.

Листинг А.7 ❖ Так делать не надо

```
mystery_text = get_mystery ('Hello JavaScript Denizens');
```

← `get_mystery` – ключевое слово или пользовательская функция?

```
for(x=1;x<10;x++){console.log(x);}
```

← Отсутствие пробелов превращает текст в неразборчивое нагромождение символов.

Листинг А.8 ❖ А надо так

```
mystery_text = get_mystery( 'Hello JavaScript Denizens' );
```

← Если скобка поставлена вплотную, значит, это функция.

```
for ( x = 1; x < 10; x++ ) { console.log( x ); }
```

← Дополнительные пробелы делают текст понятнее.

Такое соглашение принято и в других динамических языках, например в Python, Perl и PHP.

Единое использование кавычек

Мы предпочитаем использовать *одиночные, а не двойные кавычки* в качестве ограничителей строк, поскольку стандарт HTML гласит, что в двойные кавычки следует заключать значения атрибутов. В SPA очень часто встречается закавыченный HTML-код. Закрывая его в одиночные кавычки, мы можем обойтись без экранирования и кодирования. В результате программа получается короче и яснее, да и шансов допустить ошибку меньше.

Листинг А.9 ❖ Так делать не надо

```
html_snip = "<input name=\"alley_cat\" type=\"text\" value=\"bone\">";
```

Листинг А.10 ❖ А надо так

```
html_snip = '<input name="alley_cat" type="text" value="bone">';
```

Во многих языках, например Perl, PHP и Bash, существует понятие интерполирующих и неинтерполирующих кавычек. Если строка заключена в *интерполирующие кавычки*, то в ней производится подстановка переменных, а если в *неинтерполирующие* – то нет. Как правило, двойные кавычки (") являются интерполирующими, а одиночные (') – нет. В JavaScript интерполяция никогда не производится, а поведение одиночных и двойных кавычек ничем не отличается.

Тем не менее мы используем кавычки так, как принято в других популярных языках.

A.2.2. Комментарии как средство пояснения и документирования

Комментарии иногда важнее самого кода, потому что содержат существенные детали, которые без них были бы не очевидны. Особенно наглядно это проявляется в событийно-управляемом программировании, где из-за бесконечных обратных вызовов на прослеживание пути выполнения программы уходит очень много времени. Это не означает, что чем больше комментариев, тем лучше. Помещенные в нужное место, информативные и актуализируемые в процессе сопровождения комментарии – вещь очень ценная, тогда как беспорядочное нагромождение неточных комментариев может оказаться хуже полного отсутствия.

Объясняйте то, что нуждается в объяснении

Наш стандарт направлен на минимизацию количества комментариев и максимизацию их полезности. Первой цели мы достигаем, следуя соглашениям, которые делают код по возможности самоочевидным. Второй – располагая комментарии вровень с абзацами, к которым они относятся, и включая в них сведения, полезные читателю программы.

Листинг А.11 ❖ Так делать не надо

```
var
  welcome_to_the = '<h1>Welcome to Color Haus</h1>',
  houses_we_use  = [ 'yellow', 'green', 'little pink' ],
  the_results, make_it_happen, init;

// получить описание дома
var make_it_happen = function ( house ) {
  var
    sync = houses_we_use.length,
    spec = {},
    i;

  for ( i = 0; i < sync; i++ ) {
    ...
    // еще 30 строчек
  }
  return spec;
};

var init = function () {
```

```
// houses_we_use - массив цветов зданий.
// make_it_happen - функция, которая возвращает хэш описаний зданий.
//
var the_results = make_it_happen( houses_we_use );

// И помещаем приветственное сообщение в DOM
$('#welcome').text( welcome_to_the );
// А теперь наши описания
$('#specs').text( JSON.stringify( the_results ) );
};

init();
```

Листинг А.12 ❖ А надо так

```
var
  welcome_html    = '<h1>Welcome to Color Haus</h1>',
  house_color_list = [ 'yellow', 'green', 'little pink' ]
  spec_map, get_spec_map, run_init;

// Начало /get_spec_map/
// Получить хэш описаний, индексированный цветом
get_spec_map = function ( color_list_arg ) {
  var
    color_count = color_list_arg.length,
    spec_map    = {},
    i;
  for ( i = 0; i < color_count; i++ ) {
    // ... еще 30 строчек
  }
  return spec_map;
};
// Конец /get_spec_map/

run_init = function () {
  var spec_map = getSpecMap( house_color_list );

  $('#welcome').html( welcome_html );
  $('#specs').text( JSON.stringify( spec_map ) );
};

run_init();
```

Используем
ограничители
Начало
и Конец, чтобы
четко обозначить
границы длинных
участков кода.

Используем
единообразные
и осмысленные
имена переменных
вместо комментари-
ев, поясняющих их
назначение.

Единообразные осмысленные имена переменных могут нести *больше* информации при *меньшем* объеме комментариев. Ниже в этом приложении еще будет раздел об именовании переменных, но некоторые замечания сделаем уже сейчас. Имена переменных, относящихся к функциям, начинаются глаголом: `get_spec_map`, `run_init`. Имена прочих переменных выбираются так, чтобы было понятно, что

в них хранится: `welcome_html` – строка, содержащая HTML-разметку, `house_color_list` – массив названий цветов, а `спес_мар` – хэш описаний (спецификаций). Это делает код понятным при меньшем количестве комментариев, которые ведь еще сопровождать надо.

Документируйте API и TODO

Комментарии – это также средство формального документирования программы. Тут, правда, не надо перебарщивать – документацию по общей архитектуре следует не погребать в одном из десятков JavaScript-файлов, а оформлять в виде специального документа. Однако документацию по какой-то функции или API объекта можно и зачастую должно разместить прямо рядом с кодом.

Поясняйте все нетривиальные функции, описывая назначение, аргументы и используемые параметры, возвращаемое значение и возбуждаемые исключения.

Если какой-то код закомментирован, объясняйте, почему это сделано, добавляя комментарий вида `// TODO дата имя пользователя – примечание`. Имя пользователя и дата важны для оценки актуальности комментария и могут использоваться автоматизированными инструментами для формирования отчета о найденных в коде задачах на будущее (TODO).

Листинг А.13 ❖ Пример документации по API функции

```
// Начало метода DOM /toggleSlider/
// Назначение: сворачивает и раскрывает окно чата
// Обязательные аргументы:
// * do_extend (boolean) если true, то раскрывает окно, иначе сворачивает
// Необязательные аргументы:
// * callback (function) вызывается по завершении анимации
// Параметры:
// * chat_extend_time, chat_retract_time
// * chat_extend_height, chat_retract_height
// Возвращает: boolean
// * true – анимация окна чата начата
// * false – анимация окна чата не начата
// Исключения: нет
//
toggleSlider = function( do_extend, callback ) {
    // ...
};
// Конец метода DOM /toggleSlider/
```

Листинг А.14 ❖ Пример закомментированного кода

```
// Начало TODO 2012-12-29 mmikowski – отладочный код отключен
// alert( warning_text );
```

```
// ... (еще много строк) ...
//
// Конец TODO 2012-12-29 mmikowski - отладочный код отключен
```

Некоторые считают, что ненужный код следует сразу удалять, а при необходимости восстанавливать из системы управления версиями. Но наш опыт показывает, что закомментировать код, который *с большой вероятностью* снова понадобится, гораздо эффективнее, чем искать версию, в которой выброшенный код еще присутствовал, и выдергивать его оттуда. Вот если код остается закомментированным достаточно долго, то его можно и удалить.

А.3. Именованние переменных

Вы когда-нибудь обращали внимание на соглашения об именах в листингах, встречающихся в книгах? Например, можно увидеть строку типа `person_str = 'fred';`. Обычно автор так поступает, потому что не хочет впоследствии вставлять громоздкое отвлекающее напоминание о том, что хранится в переменной. Имя говорит само за себя.

Всякий, кто пишет программы, применяет какое-то соглашение об именовании, осознает он это или нет¹. Хорошее соглашение, которое признают и применяют все члены команды, имеет огромную ценность. В этом случае вместо скучного прослеживания кода и неустанного сопровождения комментариев разработчик может сосредоточиться на цели и логике программы.

А.3.1. Сокращение и повышение качества комментариев за счет соглашений об именовании

Единообразные и осмысленные имена исключительно важны для JavaScript-приложений корпоративного уровня, поскольку позволяют быстро понять, о чем идет речь, и избежать типичных ошибок. Взгляните на следующий абсолютно корректный JavaScript-код, который вполне можно встретить в реальной программе.

Листинг А.15 ❖ Пример А

```
var creator = maker( 'house' );
```

А теперь перепишем его, применяя наши соглашения об именовании, о которых расскажем чуть ниже.

¹ Что-то вроде строки из песни Раша «Freewill» (альбом 1980 года Permanent Waves): «если ты решил ничего не решать, то все равно решил».

Листинг А.16 ❖ Пример Б

```
var make_house = curry_build_item({ item_type : 'house' });
```

Пример Б выглядит более понятно. Из наших соглашений вытекает, что:

- `make_house` — конструктор объектов;
- вызываемая функция выполняет *карьерование*, то есть использует замыкание для хранения состояния и возвращает функцию;
- вызываемая функция принимает строковый аргумент, в имени которого имеется указание на его смысл;
- область видимости переменных локальна.

Все это можно было бы узнать и из примера А, изучив окружающий контекст. Для прослеживания всех функций и переменных пришлось бы потратить 5 или 30 или все 60 минут. А потом *мы должны были бы все это помнить*, работая с этим кодом или в обход него. Теряется не только время, но и сосредоточенность на задаче, которую мы решаем.

И эти затраты, которых вполне можно было бы избежать, несет *каждый* новый разработчик, встречающий этот код. А через несколько недель после написания кода любого разработчика, в том числе и его автора, можно считать новым. Очевидно, что это безумно неэффективно и чревато ошибками.

Посмотрим, как стал бы выглядеть пример А, если бы мы добавили комментарии, сообщающие ту же информацию, что в примере Б.

Листинг А.17 ❖ Пример А с комментариями

```
// 'creator' — это конструктор объекта, который мы
// получаем от 'maker'. Первый позиционный аргумент
// 'maker' должен быть строкой, которая говорит,
// конструктор каких объектов следует вернуть.
// В 'maker' используется замыкание для запоминания
// типа объекта, который должна создавать возвращенная
// функция.
```

```
var creator = maker( 'house' );
```

Мало того что пример А с комментариями получился гораздо многословнее примера Б, так его еще и писать куда дольше, главным образом потому, что мы пытались словами передать ту же информацию, которую несет в себе соглашение об именовании. Хуже того: комментарии со временем становятся неточными, потому что код изменяется, а программисты ленятся исправлять комментарии. Допустим, что спустя несколько недель мы решили изменить кое-какие имена.

Листинг А.18 ❖ Пример А с комментариями после изменения имен переменных

```
// 'creator' - это конструктор объекта, который мы
// получаем от 'maker'. Первый позиционный аргумент
// 'maker' должен быть строкой, которая говорит,
// конструктор каких объектов следует вернуть.
// В 'maker' используется замыкание для запоминания
// типа объекта, который должна создавать возвращенная
// функция.
```

On! Неправильное имя.

Ошибка – теперь это называется builder.

Блин... builder – это не maker, надо бы исправить.

Черт! Пусть кто-нибудь другой это правит – мне код писать надо!

```
var maker = builder( 'house' );
```

Вот беда, мы забыли исправить комментарий, в котором упоминаются изменившиеся имена переменных. Теперь комментарий абсолютно неверен и только вводит в заблуждение. Более того, комментарий еще и затуманивает код, потому что листинг получился *в девять раз* длиннее. Лучше бы его вообще не было. А теперь посмотрите, что было бы, если бы мы изменили имена переменных в примере Б.

Листинг А.19 ❖ Пример Б после изменения имен

```
var make_abode = curry_make_item({ item_type : 'abode' });
```

После внесения изменений новая версия автоматически останется правильной, потому что *нет никаких комментариев, нуждающихся в модификации*. Отсюда следует, что продуманное соглашение об именовании – отличный способ создания самодокументированного кода *его первым автором*, позволяющий добиться большей точности без громоздких комментариев, которые почти невозможно сопровождать. Это ускоряет разработку, повышает качество кода и упрощает сопровождение.

А.3.2. Рекомендации по именованию

Имя переменной может нести в себе массу информации, как мы только что убедились. Ниже перечислены некоторые рекомендации, которые нам показались наиболее полезными.

Используйте общепринятые символы

Возможно, некоторые члены вашей команды считают очень свежей идею назвать переменную `queensryche_album_name`. Однако те, кто тщетно пытается отыскать на своей клавиатуре клавишу `ÿ`, могут иметь на этот счет совершенно другое – и отнюдь не благожелательное – мнение. Лучше употреблять в именах переменных символы, имеющиеся на большинстве клавиатур в мире.

- Используйте в именах переменных только символы a–z, A–Z, 0–9, знак подчеркивания и \$.
- Имя переменной не должно начинаться цифрой.

Имя переменной должно нести информацию о ее области видимости

Между нашими JavaScript-файлами и модулями имеется взаимно однозначное соответствие, как в Node.js (мы подробнее остановимся на этом вопросе ниже). Опыт показывает, что полезно различать переменные, доступные в любом месте модуля, и имеющие более ограниченную область видимости.

- **Применяйте верблужью нотацию, когда переменная объявлена в области видимости модуля** (то есть доступна в любой точке пространства имен этого модуля).
- **Применяйте знаки подчеркивания, когда переменная объявлена не в области видимости модуля** (то есть для локальных переменных, объявленных внутри функций в пространстве имен модуля).

Имена всех переменных в области видимости модуля должны состоять по крайней мере из двух слогов, чтобы была понятна область видимости. Например, вместо имени `config` можно использовать более осмысленное имя `configMap`, из которого сразу видно, что переменная принадлежит области видимости модуля.

Не забывайте, что тип переменной важен

Из того, что JavaScript позволяет беспрепятственно манипулировать типами переменных, вовсе не следует, что так и нужно поступать. Рассмотрим следующий пример.

Листинг А.20 ❖ Неявное преобразование типа

```
var x = 10, y = '02', z = x + y;  
console.log ( z ); // '1002'
```

В данном случае JavaScript преобразует `x` в строку и конкатенирует ее с `y` (`02`), получая в результате строку `1002`. Скорее всего, вы хотели не этого. Преобразование типа может иметь и более трудные для понимания последствия.

Листинг А.21 ❖ Темная сторона преобразования типа

```
var  
  x = 10,  
  z = [ 03, 02, '01' ],
```

```

    i , p;

for ( i in z ) {
    p = x + z[ i ];
    console.log( p.toFixed( 2 ) );
}

// Выводится:
// 13.00
// 12.00
// TypeError: Object 1001 has no method 'toFixed'

```

Мы обнаружили, что *непреднамеренные* преобразования типов, подобные продемонстрированному выше, встречаются куда чаще *намеренных*, и это приводит к ошибкам, которые трудно найти и устранить. Вряд ли мы вообще когда-либо *намеренно* изменяем тип переменной, потому что это почти никогда не окупается из-за возникающей путаницы и трудности отслеживания типа (и это не единственная причина)¹. Поэтому, выбирая имя переменной, мы часто хотим сообщить в нем информацию о типе ее значения.

Именование булевых переменных

Если булева переменная представляет некое состояние, то мы включаем в ее имя слово *is*, например *is_retracted* или *is_stale*. Если мы хотим, чтобы булева переменная указывала на необходимость совершения действия, как, например, в аргументе функции, то употребляем слово *do*: *do_retract* или *do_extend*. А когда требуется обозначить владение, мы используем слово *has*, например *has_whiskers* или *has_wheels*. В табл. А.1 приведены примеры.

Таблица А. 1. Примеры имен булевых переменных

Индикатор	Локальная область видимости	Область видимости модуля
bool (общего вида)	bool_return	boolReturn
do (запрос действия)	do_retract	doRetract
has (обозначает обладание свойством)	has_whiskers	hasWhiskers
is (обозначает состояние)	is_retracted	isRetracted

¹ В последних версиях JIT-компилятора JavaScript в Firefox этот факт признан объективно существующим, вследствие чего стала применяться техника выведения типа, которая повышает производительность реального кода на 20–30%.

Именованние строковых переменных

Из приведенного выше примера видно, что иногда бывает полезно знать, что мы имеем дело со строковой переменной. В табл. А.2 перечислены типичные индикаторы, которыми мы пользуемся для строк.

Таблица А.2. Примеры имен строковых переменных

Индикатор	Локальная область видимости	Область видимости модуля
str (общего вида)	direction_str	directionStr
id (идентификатор)	email_id	emailId
date	email_date	emailDate
html	body_html	bodyHtml
msg (сообщение)	employee_msg	employeeMsg
name	employee_name	employeeName
text	email_text	emailText
type	item_type	itemType

Именованние целых переменных

В JavaScript целочисленный тип явно не поддерживается, но существует немало ситуаций, когда программа функционирует неправильно, если используется нецелое число. Например, перебор массива будет работать неожиданно, если индекс – число с плавающей точкой.

```
var color_list = [ 'red', 'green', 'blue' ];
```

```
color_list[1.5] = 'chartreuse';
```

```
console.log( color_list.pop() ); // 'blue'
console.log( color_list.pop() ); // 'green'
console.log( color_list.pop() ); // 'red'
console.log( color_list.pop() ); // undefined – а где же 'chartreuse'?
console.log( color_list[1.5] ); // а, вот он
console.log( color_list ); // shows [1.5: "chartreuse"]
```

Есть и другие встроенные функции, ожидающие на входе целое число, например метод `substr()`. Поэтому в тех случаях, когда важно подчеркнуть, что число целое, мы рекомендуем использовать индикаторы, как показано в табл. А.3.

Таблица А.3. Примеры имен целых переменных

Индикатор	Локальная область видимости	Область видимости модуля
int (общего вида)	size_int	sizeInt
отсутствует (соглашение)	i, j, k	Не разрешено в области видимости модуля
count	employee_count	employeeCount
index	employee_index	employeeIndex
time (миллисекунды)	retract_time	retractTime

Именованние числовых переменных

Если важно подчеркнуть, что мы имеем дело с нецелочисленными переменными, то можно использовать и другие индикаторы (см. табл. А.4).

Таблица А.4. Примеры имен числовых переменных

Индикатор	Локальная область видимости	Область видимости модуля
num (общего вида)	size_num	sizeNum
отсутствует (соглашение)	x, y, z	Не разрешено в области видимости модуля
coord (координата)	x_coord	xCoord
ratio	sales_ratio	salesRatio

Именованние регулярных выражений

Регулярным выражениям мы обычно присваиваем имена с префиксом `regex`, как показано в табл. А.5.

Таблица А.5. Примеры имен регулярных выражений

Индикатор	Локальная область видимости	Область видимости модуля
regex	regex_filter	regexFilter

Именованние массивов

Для именованния массивов нам представляются полезными следующие соглашения.

- Имя массива должно быть существительным в единственном числе, за которым следует слово «list».

- Для массивов в области видимости модуля рекомендуется использовать имена вида `существительноеList`.

Примеры приведены в табл. А.6.

Таблица А.6. Примеры имен массивов

Индикатор	Локальная область видимости	Область видимости модуля
list	timestamp_list	timestampList
list	color_list	colorList

Именование хэшей

В JavaScript официально нет типа данных «хэш» (`map`) – есть только объекты. Но мы считаем полезным отличать простые объекты, которые служат исключительно для хранения данных (хэши), от полноценных объектов. Хэш в JavaScript аналогичен структуре `map` в Java, `dict` в Python, *ассоциативному массиву* в PHP и `hash` в Perl.

При выборе имени для хэша мы обычно хотим подчеркнуть намерение разработчика и включаем в состав имени слово `map` после существительного в единственном числе. Примеры имен хэшей приведены в табл. А.7.

Таблица А.7. Примеры имен хэшей

Индикатор	Локальная область видимости	Область видимости модуля
map	employee_map	employeeMap
map	receipt_timestamp_map	receiptTimestampMap

Иногда ключ хэша чем-то необычен. В таком случае мы указываем ключ в имени, например `receipt_timestamp_map`.

Именование объектов

У объектов обычно имеется вполне конкретный аналог в «реальном мире», в соответствии с которым мы и выбираем имя.

- Имя объектной переменной – существительное, за которым следует необязательный модификатор, например `employee`, `receipt`.
- Имя объектной переменной в области видимости модуля должно состоять как минимум из двух слогов, чтобы была понятна область видимости, например `storeEmployee`, `salesReceipt`.
- Имена объектов jQuery должны начинаться знаком `$`. В настоящее время это соглашение общепринято, а в SPA объекты

jQuery (или, как их еще называют, коллекции) встречаются сплошь и рядом.

Примеры приведены в табл. А.8.

Таблица А.8. Примеры имен объектов

Индикатор	Локальная область видимости	Область видимости модуля
отсутствует (существительное в единственном числе)	Employee	storeEmployee
отсутствует (существительное в единственном числе)	Receipt	salesReceipt
\$	\$area_tabs	\$areaTabs

Если мы предполагаем, что в коллекции jQuery будет несколько элементов, то называем ее существительным во множественном числе.

Именованние функций

Функция почти всегда выполняет какое-то действие над объектом. Поэтому мы обычно начинаем имя функции с глагола.

- **Имя функции должно состоять из глагола, за которым следует существительное**, например `get_record`, `empty_cache_map`.
- **Имя функции в области видимости модуля должно состоять как минимум из двух частей**, чтобы была понятна область видимости, например `getRecord`, `emptyCacheMap`.
- **Одинаковые глаголы следует использовать для одинаковых целей**. В табл. А.9 описаны некоторые общеупотребительные глаголы.

Наш опыт показывает, что глагол `make` для конструктора, а также пары `fetch/get` и `store/save` с их семантическими различиями особенно ценны для распространения информации о намерениях в группе разработчиков. Соглашение о выборе имени `onEventname` для обработчиков событий также вошло в обиход и доказало свою полезность. Его общая форма – `on<eventname><modifier>`, где часть *modifier* не обязательна. Отметим, что имя события считается одним словом. Например, `onMouseover`, а не `onMouseOver`, `on_dragstart`, а не `on_drag_start`.

Именованние переменных неизвестного типа

Иногда мы действительно не знаем тип данных, хранящихся в переменной. Чаще всего такое бывает в двух случаях:

Таблица А.9. Примеры имен функций

Индикатор	Смысл индикатора	Локальная область видимости	Область видимости модуля
fn (общего вида)	Индикатор функции общего вида	fn_sync	fnSync
curry	Вернуть функцию, зависящую от аргументов	curry_make_user	curryMakeUser
destroy, remove	Удалить структуру данных, например массив. Предполагается, что ссылки на данные будут надлежащим образом очищены	destroy_entry, remove_element	destroyEntry, removeElement
empty	Удалить некоторые или все элементы структуры данных, не удаляя самого контейнера. Например, удалить все элементы массива, но оставить сам массив	empty_cache_map	emptyCacheMap
fetch	Вернуть данные, полученные из внешнего источника, например с помощью вызова AJAX или чтения из веб-сокета	fetch_user_list	fetchUserList
get	Вернуть данные, хранящиеся в объекте или другой внутренней структуре данных	get_user_list	getUserList
make	Вернуть вновь сконструированный объект (не используйте оператор new)	make_user	makeUser
on	Обработчик события. Имя события должно состоять из одного слова, как в HTML	on_mouseover	onMouseover
save	Сохранить данные в объекте или другой внутренней структуре данных	save_user_list	saveUserList
set	Инициализировать или обновить значения в соответствии с аргументами	set_user_name	setUserName
store	Отправить данные во внешний источник, например с помощью вызова AJAX	store_user_list	storeUserList
update	Аналогично set, но подразумевается семантика «ранее был инициализирован»	update_user_list	updateUserList

- при написании полиморфной функции, то есть принимающей данные разных типов;
- при получении данных из внешнего источника, например в случае вызова AJAX или чтения из веб-сокета.

В таких случаях определяющей характеристикой переменной является неопределенность ее типа. Мы решили, что это следует подчеркнуть наличием слова `data` в имени (см. табл. А.10).

Таблица А.10. Примеры имен переменных неизвестного типа

Локальная область видимости	Область видимости модуля	Примечание
<code>http_data</code> , <code>socket_data</code>	<code>httpData</code> , <code>socketData</code>	Данные неизвестного типа, полученные по протоколу HTTP или из веб-сокета
<code>arg_data</code> , <code>data</code>	---	Данные неизвестного типа, полученные в качестве аргумента

Теперь посмотрим, как эти рекомендации по выбору имен применяются на практике.

А.3.3. Практическое применение рекомендаций

Сравним прототип объекта до и после применения рекомендаций по выбору имен.

Листинг А.22 ❖ Так делать не надо

```

doggy = {
  temperature : 36.5,
  name        : 'Guido',
  greeting    : 'Grrrr',
  speech      : 'I am a dog',
  height      : 1.0,
  legs        : 4,
  ok          : check,
  remove      : destroy,
  greet_people : greet_people,
  say_something : say_something,
  speak_to_us : speak,
  colorify    : flash,
  show        : render
};

```

Мы понятия не имеем, что такое `temperature`: метод, строка или объект? Если это число, то по какой шкале измеряется: Фаренгейта или Цельсия?

Это свойство также непонятно. Приходится гадать, строка это или метод.

Множественное число `legs` наводит на мысль о коллекции типа массива или хэша. Но здесь оно используется для хранения целочисленного счетчика (число лап).

Именование методов просто ужасно — нет никакого соответствия между ключом и функцией, на которую он ссылается, поэтому прослеживание кода станет сущим кошмаром. Кроме того, имена функций не всегда обозначают действие. А самое худшее — это имя `ok`, которое подразумевает булево состояние, однако таким не является.

Листинг А.23 ❖ А надо так

```

dogPrototype = {
  body_temp_c    : 36.5,
  dog_name       : 'Guido',
  greet_text     : 'Grrrr',
  speak_text    : 'I am a dog',
  height_in_m    : 1.0,
  leg_count      : 4,
  check_destroy  : checkDestroy,
  destroy_dog    : destroyDog,
  print_greet    : printGreet,
  print_name     : printName,
  print_speak    : printSpeak,
  show_flash     : showFlash,
  redraw_dog     : redrawDog
};

```

Указание единицы измерения в имени говорит, что это число, и одновременно сообщает о температурной шкале.

Индикатор внутри имени говорит, что это строка, что подтверждается и присвоенным значением.

Слово count показывает, что это целое число.

Глаголы, обозначающие действия, говорят, что это методы. Обратите внимание на выравнивание имен, это помогает проследить выполнение кода.

Данные примеры взяты из файлов двух страниц – listings/apx0A/bad_dog.html и listings/apx0A/good_dog.html, которые входят в состав прилагаемых к книге ресурсов. Предлагаем вам скачать их, сравнить и самостоятельно решить, какой код более понятен и удобен для сопровождения.

А.4. Объявление и присваивание переменным

Переменной можно присвоить указатель на функцию, указатель на объект, указатель на массив, строку, число, null или undefined. В некоторых реализациях JavaScript имеются внутренние различия между целыми, 32-разрядными и 64-разрядными числами с плавающей точкой, но формального интерфейса для объявления соответствующих типов не существует.

- **Используйте {} или []** вместо new Object() или new Array() для создания нового объекта, хэша или массива. Напомним, что хэш – это простой объект для хранения данных, не имеющий методов. Если требуется наследование объектов, используйте вспомогательную функцию createObject, представленную в главе 2 и в разделе А.5 настоящего приложения.
- **Используйте вспомогательные функции для копирования объектов и массивов.** Простые переменные – булевы, строковые и числовые – копируются в момент присваивания. Например, предложение new_str = this_str копирует данные (в данном

случае строку) в `new_str`. Составные переменные, например массивы и объекты, в JavaScript *не* копируются при присваивании; копируется лишь указатель на соответствующую структуру данных. Например, после выполнения предложения `second_map = first_map` переменная `second_map` будет указывать на те же данные, что и `first_map`, а любые манипуляции с `second_map` будут отражаться в `first_map`. Корректное копирование массива или объекта не всегда легко реализовать. Мы настоятельно рекомендуем использовать для этой цели хорошо протестированные функции, например из библиотеки jQuery.

- **Явно объявляйте все переменные в самом начале области видимости функции** в одном предложении `var`. В JavaScript нет области видимости блока, объявленные в любом месте функции переменные видны из любой точки внутри этой функции. Поэтому сразу после вызова функции все переменные получают начальное значение `undefined`. Помещая все объявления переменных в начало функции, вы признаете и учитываете такое поведение. Кроме того, подобный код легче читать и проще обнаружить в нем необъявленные переменные (каковых вообще не должно быть).

```
var getMapCopy = function ( arg_map ) {
    var key_name, result_map, val_data; ◀ Только объявления – несколько в одной строке.

    result_map = {}; ◀ Только присваивание – единственное в строке.

    for ( key_name in arg_map ) {
        if ( arg_map.hasOwnProperty( key_name ) ) {
            val_data = arg_map[ key_name ]; ◀ Условное присваивание.
            if ( val_data ) { result_map[ key_name ] = val_data; }
        }
    }

    return result_map;
};
```

Объявление переменной – *не* то же самое, что присваивание ей значения. *Объявление* сообщает интерпретатору JavaScript, что переменная существует в области видимости. Присваивание связывает с переменной значение (отличное от `undefined`). Для удобства разрешено совмещать объявление с присваиванием в одном предложении `var`, но это необязательно.

- **Не пользуйтесь блоками**, потому что в JavaScript нет понятия области видимости блока¹. Определение переменной внутри блока может ввести в заблуждение программистов, знакомых с другими языками, берущими истоки в С. Определяйте переменные в области видимости функции.
- **Присваивайте все функции переменным**. Тем самым подчеркивается тот факт, что в JavaScript функции являются полноправными объектами.

```
// Плохо
function getMapCopy( arg_map ) { ... };
// Хорошо
var getMapCopy = function ( arg_map ) { ... };
```

- **Используйте именованные аргументы** в случаях, когда у функции три или более аргументов, так как позиционные аргументы легко забываются и не являются самодокументированными.

```
// Плохо
var coord_map = refactorCoords( 22, 28, 32, 48 );
// Лучше
var coord_map = refactorCoords({ x1:22, y1:28, x2:32, y2:48 });
```

- **Размещайте не более одного предложения присваивания в строке**. Располагайте их в алфавитном порядке или, если это разумно, объединяйте в логические группы. В одной строке допустимо размещать *несколько объявлений*.

```
// переменные, используемые в функции охвата и перетаскивания
var
  $cursor = null,           // текущий выделенный элемент списка
  scroll_up_intid = null,    // ид интервала для прокрутки вверх
  index, length, ratio     ← Несколько объявлений в одной строке.
  ;
```

Объявление
с присваиванием. ←

А.5. Функции

Функции играют важнейшую роль в JavaScript: они помогают организовывать код, являются контейнером, определяющим область видимости переменных, и предоставляют контекст выполнения, который можно использовать для конструирования основанных на прототипе

¹ Это верно лишь отчасти, потому что в версии JavaScript 1.7 в Firefox появилось предложение `let`, с помощью которого можно организовать область видимости блока. Однако оно не поддерживается основными браузерами, так что пользоваться им не следует.

объектов. Поэтому хотя рекомендаций, относящихся к функциям, у нас немного, они нам очень дороги.

- **Применяйте паттерн фабрики для конструкторов объектов**, поскольку он точнее соответствует тому, как на самом деле устроены объекты в JavaScript, работает быстро и может использоваться для имитации характерных для классов возможностей, например счетчика созданных объектов.

```
var createObject, extendObject,
    sayHello, sayText, makeMammal,
    catPrototype, makeCat, garfieldCat;

// ** Вспомогательная функция для определения наследования.
// Кросс-браузерный метод для наследования Object.create()
// В современных версиях js (v1.8.5+) поддержка встроена.
var objectCreate = function ( arg ) {
    if ( ! arg ) { return {}; }
    function obj() {};
    obj.prototype = arg;
    return new obj;
};

Object.create = Object.create || objectCreate;

// ** Вспомогательная функция для расширения объекта.
extendObject = function ( orig_obj, ext_obj ) {
    var key_name;
    for ( key_name in ext_obj ) {
        if ( ext_obj.hasOwnProperty( key_name ) ) {
            orig_obj[ key_name ] = ext_obj[ key_name ];
        }
    }
};

// ** методы объекта...
sayHello = function () {
    console.warn( this.hello_text + ' says ' + this.name );
};

sayText = function ( text ) {
    console.warn( this.name + ' says ' + text );
};

// ** конструктор makeMammal
makeMammal = function ( arg_map ) {
    var mammal = {
        is_warm_blooded : true,
        has_fur          : true,
        leg_count        : 4,
```

```
    has_live_birth : true,
    hello_text    : 'grunt',
    name         : 'anonymous',
    say_hello     : sayHello,
    say_text      : sayText
  };
  extendObject( mammal, arg_map );
  return mammal;
};

// ** используем конструктор млекопитающего для создания
// прототипа кошки
catPrototype = makeMammal({
  has_whiskers : true,
  hello_text : 'meow'
});

// ** конструктор кошки
makeCat = function( arg_map ) {
  var cat = Object.create( catPrototype );
  extendObject( cat, arg_map );
  return cat;
};

// ** экземпляр кошки
garfieldCat = makeCat({
  name : 'Garfield',
  weight_lbs : 8.6
});

// ** вызовы методов экземпляра кошки
garfieldCat.say_hello();
garfieldCat.say_text('Purr...');
```

- ❶ **Избегайте псевдоклассических конструкторов объектов** – тех, в которых используется ключевое слово `new`. Если вызвать такой конструктор без ключевого слова `new`, то будет повреждено глобальное пространство имен. Если вы никак не можете обойтись без такого конструктора, то хотя бы начинайте его имя с большой буквы, чтобы сразу было ясно, что это псевдоклассический конструктор.
- ❷ **Объявляйте все функции до первого использования** – помните, что объявление функции – *не* то же самое, что *присваивание ей значения*.
- ❸ **Если функцию необходимо сразу же вызвать**, то окружайте ее круглыми скобками, чтобы было понятно, что речь идет о результате, возвращенном функцией, а не о самой функции: `spa.shell = (function () { ... })()`;

А.6. Пространства имен

Поначалу объем JavaScript-кода был сравнительно невелик, и весь он размещался на одной веб-странице. В таких скриптах могли использоваться (и часто *использовались*) глобальные переменные, что не вызывало трагических последствий. Но со временем JavaScript-приложения становились все более масштабными, в обиход начали входить сторонние библиотеки, и шансы, что еще кто-то захочет назвать глобальную переменную `i`, резко возросли. А когда код из разных источников работает с одной и той же глобальной переменной, разверзаются врата ада¹.

Вероятность возникновения такой проблемы можно свести к минимуму, если использовать только одну глобальную функцию и объявить все наши переменные внутри нее:

```
var spa = (function () {
    // прочий код

    var initModule = function () {
        console.log( 'люди, привет' );
    };

    return { initModule : initModule };
})();
```

Эту единственную глобальную переменную (в данном случае `spa`) мы называем *пространством имен*. Функция, которая ей присвоена, выполняется на этапе загрузки, и, разумеется, никакие объявленные внутри нее локальные переменные недоступны в глобальном пространстве имен. Отметим, что метод `initModule` сделан доступным извне. Поэтому внешняя программа может вызвать эту функцию инициализации, но все остальное ей недоступно. Да и для вызова данной функции необходимо указывать префикс `spa`:

```
// вызываем функцию инициализации spa из другой библиотеки
spa.initModule();
```

¹ Автор однажды работал над приложением, в котором сторонняя библиотека совершенно неожиданно и, очевидно, по ошибке предъявила свои права на глобальную переменную `util` (а что им мешало использовать JSLint...). И хотя в нашем приложении было всего три пространства имен, одно из них как раз называлось `util`. Из-за конфликта имен наше приложение не работало. На то, чтобы найти ошибку и придумать, как ее обойти, ушло четыре часа. Нельзя сказать, что мы были счастливы.

Пространство имен можно разбить на части, чтобы не втискивать все приложение размером 50 Кб в один файл. Например, можно создать пространства имен `spa`, `spa.shell` и `spa.slider`:

```
// В файле spa.js:
var spa = (function () {
    // какой-то код
})();

// В файле spa.shell.js:
var spa.shell = (function () {
    // какой-то код
})();

// В файле spa.slider.js:
var spa.slider = (function () {
    // какой-то код
})();
```

А.7. Имена и структура дерева файлов

Пространства имен лежат в основе схемы именования и размещения файлов. Приведем несколько общих рекомендаций.

- **Используйте jQuery** для манипуляций моделью DOM.
- **Поищите готовый код**, например подключаемые к jQuery модули, прежде чем писать свой собственный, — соизмеряйте стоимость интеграции и разбухание кода с выгодами от стандартизации и единообразия.
- **Старайтесь не включать JavaScript-код** непосредственно в HTML-разметку; применяйте внешние библиотеки.
- **Минимизируйте, запутывайте и сжимайте JavaScript и CSS** перед запуском в эксплуатацию. Например, для минимизации и запутывания JavaScript-кода на этапе сборки можно воспользоваться программой Uglify, а для сжатия файлов во время доставки — модулем Apache2 `mod_gzip`.

Далее следуют рекомендации, касающиеся собственно JavaScript-файлов.

- **Включайте сторонние JavaScript-файлы в HTML-страницу сначала**, чтобы их функции уже были подготовлены до того, как начнет работать ваше приложение.
- **Включайте свои JavaScript-файлы** в порядке возрастания числа компонентов пространства имен. Например, нельзя загружать пространство имен `spa.shell` раньше, чем будет загружено корневое пространство имен `spa`.

- **Все JavaScript-файлы должны иметь расширение .js.**
- **Помещайте все статические JavaScript-файлы** в каталог `js`.
- **Называйте JavaScript-файлы** в соответствии с тем, какое пространство имен в них содержится; в каждом файле должно быть одно пространство имен. Например:

```
spa.js           // пространство имен spa.*
spa.shell.js    // пространство имен spa.shell.*
spa.slider.js   // пространство имен spa.slider.*
```

- **Используйте шаблон** для запуска файла, содержащего модуль JavaScript. Один такой шаблон приведен в конце этого приложения.

Мы организуем параллельные структуры JavaScript и CSS-файлов и имен классов.

- **Создавайте по одному CSS-файлу для каждого JavaScript-файла**, который порождает HTML-разметку. Например:

```
spa.css          // пространство имен spa.*
spa.shell.css    // spa.shell.*
spa.slider.css   // пространство имен spa.slider.*
```

Все CSS-файлы должны иметь расширение `.css`.

- **Помещайте все статические CSS-файлы** в каталог `css`.
- **Начинайте имена CSS-селекторов с префикса**, совпадающего с именем соответствующего им модуля. Это помогает избежать нежелательной интерференции с классами, принадлежащими сторонним модулям. Например:

```
spa.css defines #spa, .spa-x-clearall
spa.shell.css defines
    #spa-shell-header, #spa-shell-footer, .spa-shell-main
```

- **Используйте формат <пространство_имен>-х-<дескриптор>** для имен индикаторов состояния и других общих классов, например `spa-x-select` и `spa-x-disabled`. Помещайте их в корневую таблицу стилей, например `spa.css`.

Эти рекомендации легко запомнить и применять на практике. А получающуюся в результате организацию взаимосвязанных CSS и JavaScript-файлов понять гораздо проще.

А.8. Синтаксис

В этом разделе приведен обзор синтаксиса JavaScript и даны некоторые рекомендации.

A.8.1. Метки

Любое предложение может быть помечено. Но мы не рекомендуем помечать какие-либо предложения, кроме **while**, **do**, **for**, **switch**. Имя метки должно быть существительным в единственном числе, набранным заглавными буквами:

```
var
  horseList = [ 'Anglo-Arabian', 'Arabian', 'Azteca', 'Clydsedale' ],
  horseCount = horseList.length,
  breedName, i
;

HORSE:
for ( i = 0; i < horseCount; i++ ) {
  breedName = horseList[ i ];
  if ( breedName === 'Clydsedale' ) { continue HORSE; }
  // обработка всех лошадей, кроме клейдесдалей
  // ...
}
```

A.8.2. Предложения

Ниже перечислены наиболее употребительные предложения JavaScript с нашими советами по их использованию.

continue

Мы стараемся не использовать предложения **continue** без метки, поскольку из-за них труднее разобраться в потоке выполнения программы. Включение метки в какой-то мере смягчает этот недостаток.

```
// не рекомендуется
continue;
```

```
// рекомендуется
continue HORSE;
```

do

Предложение **do** должно выглядеть следующим образом:

```
do {
  // предложения
} while ( условие );
```

Всегда ставьте в конце **do** точку с запятой.

for

Предложение **for** должно быть записано в одной из следующих форм:

```

for ( инициализация; условие; обновление ) {
    // предложения
}

for ( переменная in объект ) {
    if ( фильтр ) {
        // предложения
    }
}

```

Первая форма применяется для обхода массивов и для организации циклов с известным количеством итераций.

Вторая форма применяется для объектов и хэшей. Имейте в виду, что атрибуты и методы, добавленные в прототип объекта, также включаются. Чтобы отфильтровать только собственные свойства, пользуйтесь методом `hasOwnProperty`:

```

for ( переменная in объект ) {
    if ( объект.hasOwnProperty( переменная ) ) {
        // предложения
    }
}

```

if

Формы предложения `if` перечислены ниже. Ключевое слово `else` должно располагаться в отдельной строке:

```

if ( условие ) {
    // предложения
}

if ( условие ) {
    // предложения
}
else {
    // предложения
}

if ( условие ) {
    // предложения
}
else if ( условие ) {
    // предложения
}
else {
    // предложения
}

```

return

В предложении `return` не следует заключать возвращаемое значение в скобки. Возвращаемое выражение должно начинаться в той же строке, что ключевое слово `return`, чтобы по ошибке не вставить точку с запятой.

switch

Предложение `switch` рекомендуется записывать в такой форме:

```
switch ( expression ) {  
    case expression:  
        // предложения  
        break;  
    case expression:  
        // предложения  
        break;  
    default:  
        // предложения  
}
```

Каждая группа предложений (кроме следующих после `default`) должна заканчиваться предложением `break`, `return` или `throw`; «проваливанием» следует пользоваться с большой осторожностью и только сопроводив комментариями, но и в этом случае дважды подумайте, так ли это нужно. Окупается ли краткость потерей понятности? Наверное, все-таки нет.

try

Предложение `try` должно быть записано в одной из следующих форм:

```
try {  
    // предложения  
}  
catch ( переменная ) {  
    // предложения  
}
```

```
try {  
    // предложения  
}  
catch ( переменная ) {  
    // statements  
}  
finally {  
    // предложения  
}
```

while

Предложение **while** рекомендуется записывать в такой форме:

```
while ( условие ) {  
    // предложения  
}
```

Предложений **while** лучше избегать, потому что они чреваты возникновением бесконечных циклов. Отдавайте предпочтение предложению **for** там, где это возможно.

with

Предложений **with** лучше вообще избегать. Если требуется изменить значение **this** во время вызова функции, пользуйтесь методом **object.call()** и родственными ему.

A.8.3. Прочие замечания о синтаксисе

Разумеется, JavaScript не исчерпывается метками и предложениями. Вот еще несколько советов.

Избегайте оператора «запятая»

Старайтесь не пользоваться оператором «запятая» (например, в некоторых вариантах цикла **for**). Это не относится к употреблению запятой в качестве разделителя в литеральных объектах, литеральных массивах, предложениях **var** и списках параметров.

Избегайте выражений присваивания

Не пользуйтесь присваиванием в части «условие» предложений **if** и **while**, то есть избегайте конструкций вида **if (a = b) { ... }**, поскольку непонятно, что вы хотели сделать: сравнить на равенство или выполнить присваивание.

*Всегда используйте для сравнения операторы **===** и **!==***

Почти всегда операторы **===** и **!==** предпочтительнее. Операторы **==** и **!=** производят приведение типа. В частности, не следует использовать **==** для сравнения значений, которые могут быть приведены к **false**. Мы настраиваем JSLint так, чтобы приведение типа считалось ошибкой. Если вы хотите узнать, «похоже» ли значение на **true** или **false**, пользуйтесь такой конструкцией:

```
if ( is_drag_mode ) { // is_drag_mode похоже на true!  
    runReport();  
}
```

Избегайте неоднозначностей при использовании плюса и минуса

Следите за тем, чтобы за знаком + не следовал + или ++. Такая практика только вносит путаницу. Вставляйте скобки, чтобы четко обозначить свое намерение.

```
// неудачно  
total = total_count + +arg_map.cost_dollars;  
// лучше  
total = total_count + (+arg_map.cost_dollars);
```

При такой записи никто по ошибке не прочтет + + как ++. Те же замечания относятся к знаку минус.

Не пользуйтесь eval

Будьте осторожны – у `eval` есть и другие личины. Не пользуйтесь конструктором `Function`. Не передавайте строку в качестве аргумента функции `setTimeout` или `setInterval`. Используйте вместо `eval` анализатор, когда требуется преобразовать строку из формата JSON во внутреннюю структуру данных.

А.9. Валидация кода

JSLint – это весьма популярный инструмент валидации JavaScript-кода, который разработал и сопровождает Дуглас Крокфорд (Douglas Crockford). Он очень полезен для выявления ошибок кодирования и контроля соблюдения принципиальных рекомендаций. Если вы занимаетесь созданием JavaScript-кода профессионального качества, то просто обязаны использовать JSLint или аналогичный валидатор. Нам он помогает избежать самых разных ошибок и сократить время разработки.

А.9.1. Установка JSLint

1. Скачайте последний дистрибутив `jslint4java`, например `jslint4java-2.0.2.zip`, с сайта <http://code.google.com/p/jslint4java/>.
2. Распакуйте файл и установите программу, следуя инструкциям для своей платформы.

Если вы работаете с OS X или Linux

Можете переместить jar-файл в другое место, например командой `sudo mv jslint4java-2.0.2.jar /usr/local/lib/`, а затем создать файл `/usr/local/bin/jslint`, содержащий следующий командный скрипт:

```
#!/bin/bash
# См. http://code.google.com/p/jslint4java/

for jsfile in $@;
do /usr/bin/java \
  -jar /usr/local/lib/jslint4java-2.0.1.jar \
  «$jsfile»;
done
```

Не забудьте сделать файл `jslint` исполняемым – `sudo chmod 755 /usr/local/bin/jslint`.

Если на вашей машине установлен Node.js, то можете установить другую версию JSLint: `npm install -g jslint`. Она работает быстрее, хотя для примеров, приведенных в этой книге, не тестировалась.

A.9.2. Настройка JSLint

В наш шаблон модуля включены конфигурационные параметры JSLint, соответствующие нашему стандарту кодирования:

```
/*jslint      browser : true, continue : true,
   devel     : true, indent : 2,    maxerr : 50,
   newcap    : true, nomen  : true, plusplus : true,
   regexp    : true, sloppy : true, vars    : false,
   white     : true
*/
/*global $, spa, <прочие внешние переменные> */
```

- `browser : true` – разрешаются ключевые слова, имеющие отношение к браузеру: `document`, `history`, `clearInterval` и т. д.
- `continue : true` – разрешается использовать предложение `continue`.
- `devel : true` – разрешается использовать ключевые слова, применяемые на этапе разработки: `alert`, `console` и т. д.
- `indent : 2` – ожидается, что отступ составляет два пробела.
- `maxerr : 50` – выйти из JSLint после обнаружения 50 ошибок.
- `newcap : true` – не считать знак подчеркивания в начале идентификатора ошибкой.
- `nomen : true` – разрешать имена конструкторов, не начинающиеся с большой буквы.
- `plusplus : true` – разрешать операторы `++` и `--`.

- `regex : true` – разрешать полезные, но потенциально опасные конструкции в регулярных выражениях.
- `sloppy : true` – не требовать использования прагмы `strict`.
- `vars : false` – не разрешать несколько предложений `var` в одной области видимости функции.
- `white : true` – отключить встроенные в JSLint проверки форматирования.

A.9.3. Использование JSLint

Для валидации программы мы можем в любой момент запустить JSLint из командной строки:

```
jslint filepath1 [filepath2, ... filepathN]
# пример: jslint spa.js
# пример: jslint *.js
```

Мы написали скрипт для `git`, подключаемый в точке сохранения, который проверяет все измененные JavaScript-файлы перед записью их в репозиторий. Показанный ниже командный скрипт можно поместить в файл `repo/.git/hooks/pre-commit`.

```
#!/bin/bash
# См. www.davidpashley.com/articles/writing-robust-shell-scripts.html
# unset var check
set -u;
# exit on error check
# set -e;

BAIL=0;
TMP_FILE=»/tmp/git-pre-commit.tmp»;
echo;
echo «JSLint test of updated or new *.js files ...»;
echo « We ignore third_party libraries in ../js/third_party/...»;
git status \
| grep '\.js$' \
| grep -v '/js/third_party/' \
| grep '#\s\+(\modified|new file\)' \
| sed -e 's/#\s\+(\modified|new file\):\s\+//g' \
| sed -e 's/\s\+$/g' \
| while read LINE; do
    echo -en « Check ${LINE}: ... «
    CHECK=$(jslint $LINE);
    if [ «${CHECK}» != «» ]; then
        echo «FAIL»;
    else
        echo «pass»;
    fi;
done \
| tee «${TMP_FILE}»;
```

```

echo «JSLint test complete»;
if grep -s 'FAIL' «${TMP_FILE}»; then
    echo «JSLint testing FAILED»;
    echo " Please use jslint to test the failed files and ";
    echo " commit again once they pass the check.";
    exit 1;
fi

echo;
exit 0;

```

Можете адаптировать под себя, если хотите. И не забудьте сделать файл исполняемым (в Mac и Linux это делается командой `chmod 755 pre-commit`).

A.10. Шаблон модуля

Опыт показывает, что полезно разбивать модуль на стандартные секции. Это помогает понять содержащийся в нем код, упрощает навигацию и напоминает о необходимости соблюдать рекомендации по кодированию. Ниже приведен шаблон, на котором мы остановились, написав сотни модулей для разных проектов.

Листинг A.24 ❖ Рекомендуемый шаблон модуля

```

/* ←
 * module_template.js
 * Шаблон функционального модуля для браузера.
 */

/*jslint      browser : true, continue : true,
 * devel : true, indent : 2, maxerr : 50,
 * newcap : true, nomen : true, plusplus : true,
 * regexp : true, sloppy : true, vars : false,
 * white : true
 */

/*global $, spa */

spa.module = (function () { ←
    //----- НАЧАЛО ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ ----- ←
    var
        configMap = {
            settable_map : { color_name: true },
            color_name : 'blue'
        },
        stateMap = { $container : null },

```

В заголовке включаем сведения о назначении, авторе и авторских правах. В результате эта информация не потеряется при любом способе передачи файла.

В заголовке размещаем параметры JSLint. Мы рекомендуем подключать скрипт в точке сохранения системы управления версиями, чтобы в репозиторий попадали только файлы, прошедшие контроль JSLint.

Создаем пространство имен для модуля с помощью самовыполняющейся функции. Это предотвращает случайное создание глобальных переменных. В одном файле должно быть только одно пространство имен, и его имя должно точно соответствовать имени файла. Например, если модуль содержит пространство имен `spa.shell`, то файл должен называться `spa.shell.js`.

Объявляем и инициализируем переменные в области видимости модуля. Обычно мы храним конфигурационные параметры модуля в хэше `configMap`, переменные состояния, изменяющиеся во время выполнения, — в хэше `stateMap`, а кэш коллекций jQuery — в хэше `jqueryMap`.


```

jqueryMap = {},

setJqueryMap, configModule, initModule;
//----- КОНЕЦ ПЕРЕМЕННЫХ В ОБЛАСТИ ВИДИМОСТИ МОДУЛЯ -----

```

В этой секции находятся все служебные методы. Они не обращаются к объектной модели документа (DOM) и потому не нуждаются в браузерере. Если полезность метода не ограничивается одним модулем, то его следует поместить в общую библиотеку, например `spa.util.js`.

```

//----- НАЧАЛО СЛУЖЕБНЫХ МЕТОДОВ -----
// пример : getTrimmedString
//----- КОНЕЦ СЛУЖЕБНЫХ МЕТОДОВ -----

```

```

//----- НАЧАЛО МЕТОДОВ DOM -----
// Начало метода DOM /setJqueryMap/
setJqueryMap = function () {
    var $container = stateMap.$container;

    jqueryMap = { $container : $container };
};
// Конец метода DOM /setJqueryMap/
//----- КОНЕЦ МЕТОДОВ DOM -----

```

В этой секции находятся все закрытые методы, обращающиеся к DOM и, следовательно, нуждающиеся в браузерере. Например, может существовать метод DOM для перемещения спрайта с помощью стилей CSS. Для кэширования коллекций jQuery следует использовать метод `setJqueryMap`.

В этой секции находятся все закрытые обработчики событий: нажатия кнопки, нажатия клавиши, изменения размера окна браузера, получения сообщения из веб-сокета и т. д. Для изменения DOM обработчики событий, как правило, вызывают методы DOM, а не занимаются этим самостоятельно.

```

//----- НАЧАЛО ОБРАБОТЧИКОВ СОБЫТИЙ -----
// пример: onClickButton = ...
//----- КОНЕЦ ОБРАБОТЧИКОВ СОБЫТИЙ -----

```

Здесь могут находиться методы обратного вызова. Если наш модуль предоставляет обратные вызовы, то обычно мы помещаем их между обработчиками событий и открытыми методами. Это квазиоткрытые методы, так как вызываются внешними модулями, которым они и предоставлены.

```

//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого модуля /configModule/
// Назначение: задает конфигурацию допустимых ключей.
// Аргументы: хэш допускающих установку ключей и значений
// * color_name - какой цвет использовать
// Настройки :
// * В configMap.settable_map объявлены все допустимые ключи
// Возвращает: true
// Исключения: нет
//
configModule = function ( input_map ) {
    spa.butil.setConfigMap({
        input_map      : input_map,
        settable_map   : configMap.settable_map,
        config_map     : configMap
    });
};

```

В этой секции находятся все открытые методы, составляющие открытый интерфейс модуля. Если в модуле имеются методы `configModule` и `initModule`, то они должны находиться именно здесь.

```

    return true;
};
// Конец открытого модуля /configModule/

// Начало открытого модуля /initModule/
// Назначение: инициализирует модуль
// Аргументы:
// * $container - элемент jquery, используемый этим функциональным модулем
// Возвращает: true
// Исключения: по-разному
//
initModule = function ( $container ) {
    stateMap.$container = $container;
    setJqueryMap();

    return true;
};
// Конец открытого модуля /initModule/

// возвращаем открытые методы ← Возвращаем объект, содержащий
//                                     все открытые методы.
return {
    configModule : configModule,
    initModule   : initModule
};
//----- КОНЕЦ ОТКРЫТЫХ МЕТОДОВ -----
})();
```

A.11. Резюме

Хороший стандарт кодирования необходим, чтобы один или несколько программистов работали максимально эффективно. Представленный нами стандарт полный и логичный, но мы понимаем, что он подойдет не каждой команде. Как бы то ни было, мы надеемся, что он побудит читателей к размышлениям о типичных проблемах и о том, как их можно разрешить или хотя бы сгладить за счет применения соглашений. Мы настоятельно рекомендуем любой команде выработать свой стандарт, прежде чем браться за крупный проект.

Код пишется один раз, а читается многократно, поэтому мы за оптимизацию ради удобочитаемости. Мы ограничиваем длину строк 78 знаками и используем отступ шириной в два пробела. Мы не пользуемся точками табуляции. Мы объединяем строки в логические абзацы, чтобы читателю было проще понять наши намерения, и применяем единообразные правила разбиения длинных строк. Для расстановки скобок мы предпочитаем стиль K&R, а для различения ключевых слов и функций используем разрядку. При определении

строковых литералов мы отдаем предпочтение одиночным кавычкам. Для пояснения того, что делает программа, мы предпочитаем использовать соглашения, а не комментарии. Осмысленные и построенные по общему принципу имена переменных – ключ к передаче информации о наших намерениях, это позволяет обойтись без чрезмерно обширных комментариев. Если мы все-таки включаем комментарии, то помещаем их в начале абзаца. Нетривиальные интерфейсы комментируются в обязательном порядке и следуя общей схеме.

Мы защищаем свой код от нежелательного взаимодействия с другими скриптами, применяя пространства имен, которые реализуются в виде самовыполняющихся функций. Мы подразделяем корневое пространство имен на части, чтобы не порождать слишком громоздкие файлы и чересчур широкие области видимости. В каждом JavaScript-файле мы определяем только одно пространство имен, имя которого согласовано с именем самого файла. Для CSS-селекторов и JavaScript-файлов мы организуем параллельные пространства имен.

Мы установили и настроили JSLint. Мы всегда прогоняем свой код через JSLint, прежде чем поместить его в репозиторий исходных кодов. Для валидации всех файлов мы используем одни и те же параметры. Мы предложили шаблон модуля, вобравший в себя многие описанные выше соглашения и включающий в начале конфигурационные параметры JSLint.

Смысл стандарта кодирования – в том, чтобы освободить программистов от черной работы, предложив им общий диалект и единообразную структуру. Это позволяет направить весь свой творческий потенциал на решение действительно важных задач. Хороший стандарт позволяет прояснить намерения, а это чрезвычайно важно в крупномасштабных проектах.

Приложение 6

Тестирование SPA

В этом приложении:

- ✧ Режимы тестирования.
- ✧ Выбор каркаса тестирования.
- ✧ Настройка `nodeunit`.
- ✧ Создание комплекта тестов.
- ✧ Адаптация модулей SPA для тестирования.

Это приложение опирается на код, который мы закончили писать в главе 8. У вас должны быть все созданные в главе 8 файлы, потому что мы будем развивать их. Рекомендуем скопировать эти файлы в новый каталог «`appendix_B`» и уже в нем вносить изменения.

Мы большие поклонники разработки через тестирование и даже работали над экстравагантными проектами, в которых была автоматизирована *генерация* тестов. Использовался инструмент порождения перестановок, который автоматически генерировал тысячи регрессионных тестов, имея на входе всего лишь описание API и ожидаемое поведение. После любой модификации кода разработчик должен был прогнать все регрессионные тесты и лишь в случае успеха имел право поместить новую версию в репозиторий. А в случае появления нового API разработчик добавлял его описание в конфигурационный файл, после чего автоматически генерировались сотни или тысячи новых тестов. Такой подход позволял создавать код отменного качества, поскольку покрытие было достаточно полным, и нам редко приходилось возвращаться назад из-за внесенной ошибки.

Хотя нам нравятся такие регрессионные тесты, в этом приложении мы все же умерим свои амбиции. Времени и места хватит только на то, чтобы помочить ножки, а не принять ванну. Но мы все же опишем режимы тестирования, обсудим, как их использовать, и создадим комплект тестов с помощью `jQuery` и каркаса тестирования. Мы начинаем тестирование позже, чем стали бы в реальном проекте, — вообще-то мы предпочитаем писать тесты вместе с кодом, потому что это дает возможность прояснить назначение самого кода. И будто

специально для того, чтобы подтвердить этот момент, мы обнаружили и исправили две ошибки, когда писали данное приложение¹. Ну а теперь поговорим о том, какие режимы тестирования необходимы для нашего SPA.

Б.1. Режимы тестирования

При разработке SPA мы используем по меньшей мере четыре разных режима тестирования, которые обычно применяются в следующем порядке:

1. Тестирование модели без браузера с подставными данными (режим 1).
2. Тестирование пользовательского интерфейса с подставными данными (режим 2).
3. Тестирование модели без браузера с реальными данными (режим 3).
4. Тестирование модели пользовательского интерфейса с реальными данными (режим 4).

Мы должны максимально облегчить переключение из одного режима в другой, чтобы можно было быстро обнаруживать, изолировать и устранять ошибки. Отсюда, в частности, вытекает, что тестированию в любом режиме должен подвергаться один и тот же код. Мы хотим запускать тесты без браузера (режимы 1 и 3) и с браузером (режимы 2 и 4).

На рис. Б.1 показано, какие модули используются при тестировании модели без браузера с подставными данными (режим 1). Обычно сначала код тестируется именно в этом режиме, чтобы убедиться, что API модели работает, как задумано.

На рис. Б.2 показано, какие модули используются при тестировании пользовательского интерфейса с подставными данными (режим 2). Этот режим очень удобен для изоляции ошибок представления и контроллера уже после того, как модель протестирована.

На рис. Б.3 показано, какие модули используются при тестировании модели без браузера с реальными данными (режим 3). Это позволяет изолировать ошибки в серверном API.

¹ Если вам интересно, то ошибки состояли в следующем: 1) список людей в онлайн некорректно очищался при завершении сеанса и 2) обращение к методу `sra.model.chat.get_chatee()` возвращало неактуальный объект, после того как был обновлен аватар участника чата. В главе 6 обе ошибки уже исправлены.

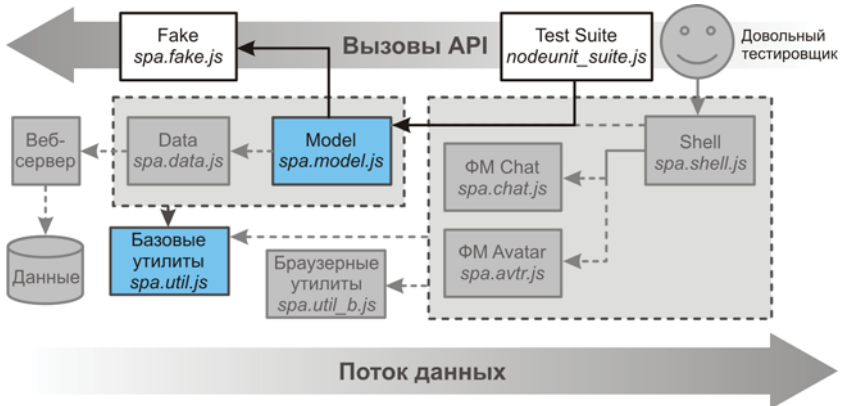


Рис. Б.1 ❖ Тестирование модели без браузера с подставными данными (режим 1)

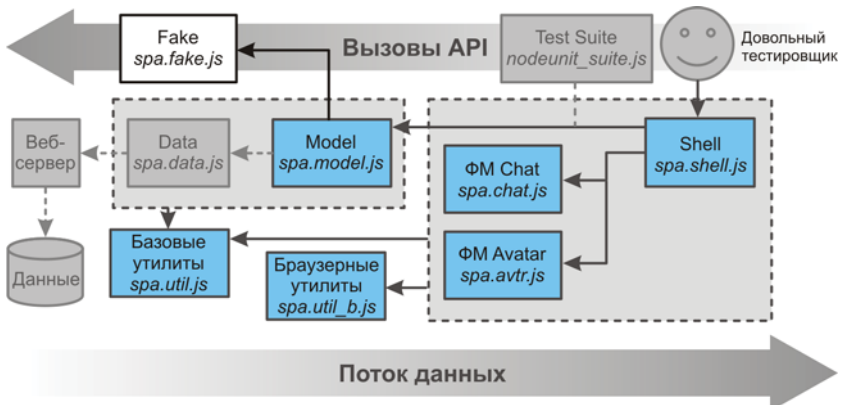


Рис. Б.2 ❖ Тестирование представления и контроллера с подставными данными (режим 1)

На рис. Б.4 показано, какие модули используются при тестировании модели и пользовательского интерфейса с реальными данными (режим 4). Это позволяет проверить все приложение целиком. «Повернутые» на тестировании (или стремящиеся к этому, как мы) называют этот режим *интеграционным тестированием*, или *тестированием сопряжений*.

Если тестирование в первых трех режимах было проведено качественно, то в режиме 4 не должно быть найдено много ошибок. А если

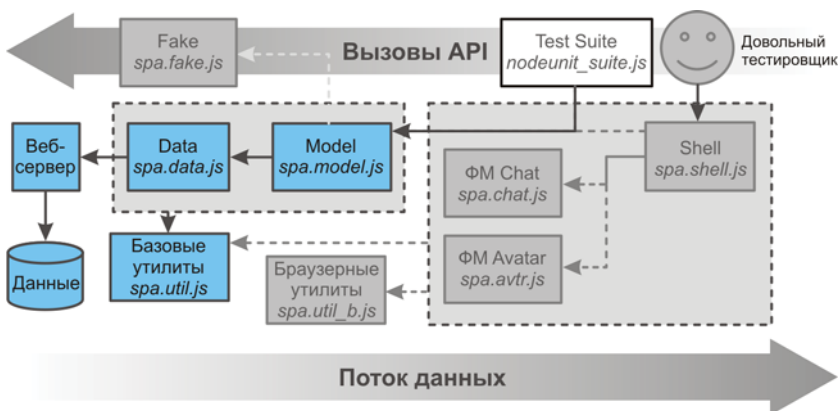


Рис. Б.3 ❖ Тестирование модели с реальными данными (режим 3)

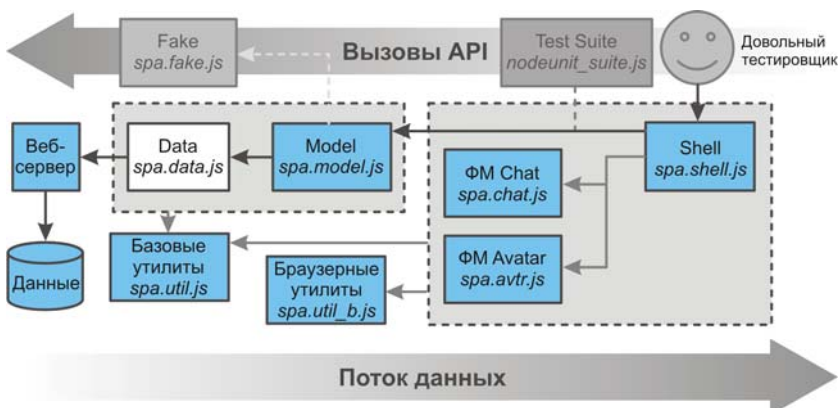


Рис. Б.4 ❖ Интеграционное тестирование с реальными данными (режим 4)

ошибка все-таки обнаруживается, то нужно попытаться воспроизвести ее в более простом режиме, начав с самого первого. Если говорить об эффективном разрешении проблем, то режим 4 можно сравнить с луной – посмотреть интересно, но жить вряд ли кому захочется.

В этом разделе мы внесем необходимые изменения, чтобы можно было тестировать пользовательский интерфейс в браузере с подставными и с реальными данными (режимы 2 и 4). Вот что мы собираемся сделать:

- добавить в модель метод `spa.model.setDataMode`, который будет переключаться с подставных данных на реальные;
- модифицировать модуль `Shell` так, чтобы на этапе инициализации он смотрел, есть ли в строке запроса параметр `fake`, и в зависимости от этого устанавливал тот или иной режим с помощью метода `spa.model.setDataMode`.

Добавить метод `spa.model.setDataMode` нетрудно: всего-то и нужно, что изменить переменную `isFakeData` в области видимости модуля. Изменения выделены в следующем листинге **полужирным** шрифтом.

Листинг Б.1 ❖ Добавление метода `spa.model.setDataMode` в модель – `webapp/public/js/spa.model.js`

```
...
spa.model = (function () {
  'use strict';
  var
    configMap = { anon_id : 'a0' },
    stateMap = { ...
    },

    isFakeData = true, ← По умолчанию подставные данные.

    personProto, makeCid, clearPeopleDb, completeLogin,
    makePerson, removePerson, people, chat, initModule,
    setDataMode;
...
  setDataMode = function ( arg_str ) { ← Устанавливаем переменную isFakeData
    isFakeData = arg_str === 'fake'      в области видимости модуля.
      ? true : false;
  };

  return {
    initModule : initModule,
    chat       : chat,
    people     : people,
    setDataMode : setDataMode ← Добавляем в список экспорта.
  };
})();
```

Следующий шаг – изменить модуль `Shell`, так чтобы он анализировал параметры строки запроса на этапе инициализации и вызывал метод `spa.model.setDataMode` (тот самый, что мы только что добавили). Это чисто косметическое изменение показано **полужирным** шрифтом в листинге ниже.

Листинг Б.2 ❖ Установка вида данных в Shell – webapp/public/js/spa.shell.js

```

...
//----- НАЧАЛО ОТКРЫТЫХ МЕТОДОВ -----
// Начало открытого метода /initModule/
...
//
initModule = function ( $container ) {
    var data_mode_str;

    // установить вид данных в зависимости от наличия параметра
    // fake в строке запроса
    data_mode_str
        = window.location.search === '?fake'
        ? 'fake' : 'live';
    spa.model.setDataMode( data_mode_str );

    // загрузить HTML и кэшировать коллекции jQuery
    stateMap.$container = $container;
    $container.html( configMap.main_html );
    setJqueryMap();
    ...

```

Сначала зайдём в каталог webapp, установим модули (`npm install`) и запустим приложение Node (`node app.js`). Если теперь открыть в браузере документ с флагом `fake` (`http://localhost:3000/spa.html?fake`), то мы будем тестировать интерфейс с подставными данными (режим 2)¹. Если же открыть в браузере документ без флага `fake` (`http://localhost:3000/spa.html`), то будет тестироваться интерфейс с реальными данными (режим 4). В следующих разделах мы обсудим, как тестировать SPA без браузера (режимы 1 и 3). Но сначала выберем каркас тестирования.

Б.2. Выбор каркаса тестирования

Мы проектировали архитектуру SPA так, чтобы облегчить тестирование модели без браузера. Опыт показывает, что если модель работает правильно, то устранить ошибки в пользовательском интерфейсе тривиально. Кроме того, мы пришли к выводу, что люди часто (но не всегда) тестируют интерфейс более эффективно, чем скрипты.

Вместо браузера мы будем использовать для тестирования модели Node.js. Это позволит без труда автоматизировать прогон комплекта

¹ Да, мы знаем, что поиск параметра в запросе сделан небрежно. В производственном приложении мы воспользовались бы библиотечной функцией надежнее.

тестов на этапе разработки, не выполняя развертывания. А поскольку браузер не нужен, то тесты проще писать, сопровождать и развивать.

Для Node.js существует много каркасов тестирования, которые использовались и совершенствовались годами. Поступим мудро – возьмем один из них и не будем изобретать свой собственный. Вот перечень каркасов, которые нам по тем или иным причинам показались особенно интересными¹:

- *jasmine-jquery* – умеет «наблюдать» за событиями jQuery;
- *mocha* – популярный каркас, аналогичный nodeunit, но с более качественными средствами отчетности;
- *nodeunit* – популярный каркас, обладающий простыми и вместе с тем мощными средствами;
- *patr* – использует обещания (аналоги объектов `$.Deferred` в jQuery) для асинхронного тестирования;
- *vows* – популярный асинхронный каркас, основанный на идеях разработки через поведение (BDD);
- *zombie* – популярный каркас для автоматизированного тестирования полного стека с использованием движка WebKit.

Zombie – всеобъемлющий каркас, предназначенный для тестирования как модели, так и пользовательского интерфейса. Он даже включает в себя движок отрисовки WebKit, чтобы можно было проверять, как отрисовываются элементы. Мы не пойдем по этому пути, потому что устанавливать, настраивать и сопровождать Zombie дорого и утомительно, а это всего лишь приложение, а не отдельная книга. Каркасы *jasmine-jquery* и *patr* кажутся нам интересными, но, на наш взгляд, уровень их поддержки оставляет желать лучшего. Mocha и *vows* популярны, но нам хотелось бы начать с чего-нибудь попроще.

Остается, стало быть, только *nodeunit* – популярный, мощный, простой каркас, который к тому же отлично интегрируется с нашей IDE. Вот его и настроим.

Б.3. Настройка nodeunit

Прежде чем устанавливать *nodeunit*, необходимо установить Node.js, как описано в главе 7. Затем нужно установить два npm-пакета:

- *jquery* – мы должны установить версию jQuery для Node.js, потому что в нашей модели используются глобальные пользова-

¹ Полный перечень см. по адресу <https://github.com/joyent/node/wiki/modules#testing>.

тельские события, а для этого нужны jQuery и подключаемый модуль `jquery.event.gevent`. В придачу мы получаем имитацию браузерного окружения. Так что если мы захотим протестировать манипуляции с DOM, то такая возможность имеется;

- `nodeunit` – этот пакет дает нам командный инструмент `nodeunit`. При прогоне комплекта тестов мы будем использовать `nodeunit` вместо `node`.

Желательно установить эти пакеты на уровне системы, чтобы их можно было использовать во всех проектах с участием Node.js. Это можно сделать, указав флаг `-g` и выполнив установку от имени пользователя `root` (или администратора, если вы работаете в Windows). В Linux и Mac нужно выполнить такие команды:

Листинг Б.3 ❖ Установка пакетов jQuery и nodeunit на уровне системы

```
$ sudo npm install -g jquery
$ sudo npm install -g nodeunit
```

Быть может, понадобится сообщить среде исполнения, где искать системные библиотеки Node.js, установив переменную окружения `NODE_PATH`. В Linux и Mac для этого следует включить следующую строку в свой файл `~/.bashrc`:

```
$ echo 'export NODE_PATH=/usr/lib/node_modules' >> ~/.bashrc
```

В этом случае переменная `NODE_PATH` будет устанавливаться всякий раз при открытии нового сеанса работы с терминалом¹. Установив Node.js, jQuery и nodeunit, подготовим наши модули к тестированию.

Б.4. Создание комплекта тестов

Начиная с главы 6, мы имеем все ингредиенты, необходимые для тестирования нашей модели: четко определенный API и известные данные (благодаря модулю Fake). На рис. Б.5 показано, как мы планируем тестировать модель².

Прежде чем приступать к тестированию, нужно заставить Node.js загрузить наши модули. Сделаем это.

¹ В уже открытом сеансе выполните команду `export PATH=/usr/lib/node_modules`. Путь может зависеть от того, как был установлен Node.js. В Mac можно попробовать путь `/usr/local/share/npm/lib/node_modules`.

² Придирчивый читатель наверняка заметил, что это точная – пиксельная – копия одного из предыдущих рисунков. Эх, если бы нам платили за дюйм колонки...

Б.4.1. Инструктируем Node.js загрузить наши модули

Node.js обрабатывает глобальные переменные иначе, чем браузеры. В отличие от браузеров, переменные по умолчанию имеют область видимости файла. По существу, Node.js оборачивает каждый библиотечный файл анонимной функцией. Чтобы переменная оказалась доступна всем модулям, ее нужно сделать свойством объекта верхнего уровня. А таковым в Node.js является не `window`, как в браузерах, а – как бы вы думали? – `global`.

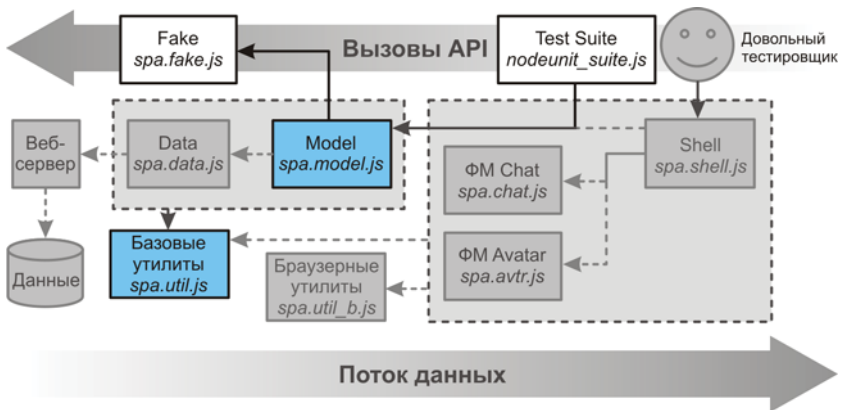


Рис. Б.5 ❖ Тестирование модели без браузера с подставными данными (режим 1)

Мы проектировали модули для работы в браузере. Но, проявив некоторую изобретательность, можно чуть-чуть модифицировать их и использовать в Node.js. Вот что нужно сделать. Наше приложение работает в едином пространстве имен (объекте) `spa`. Поэтому если объявить в тестовом скрипте Node.js атрибут `global.spa` до загрузки наших модулей, то все будет работать, как и предполагалось.

Пока все это не улетучилось из нашей кратковременной памяти, давайте запустим комплект тестов `webapp/public/nodeunit_suite.js`, как показано в следующем листинге.

Листинг Б.4 ❖ Объявление пространства имен в комплекте тестов – `webapp/public/nodeunit_suite.js`

```
/*
 * nodeunit_suite.js
 * Комплект автономных тестов для SPA
```

```

*
* Запускать командой /nodeunit <этот_файл>/
*/

/*jslint      node : true,  continue : true,
   devel  : true, indent : 2,    maxerr : 50,
   newcap : true, nomen  : true, plusplus : true,
   regexp : true, sloppy : true, vars    : false,
   white  : true
*/

/*global spa */

// наши модули и глобальные переменные
global.spa = null;

```

Добавляем параметр `node : true`, чтобы JSLint предполагал, что скрипт работает в окружении Node.js.

Создаем атрибут `global.spa`, чтобы модули SPA могли использовать пространство имен `spa` при загрузке.

Чтобы покончить с загрузкой наших модулей, нам осталось только подправить корневой JavaScript-файл (`webapp/public/js/spa.js`). В результате во время тестирования будет использоваться правильная глобальная переменная `spa`. Изменения выделены **полужирным** шрифтом в следующем листинге.

Листинг Б.5 ❖ Модификация корневого JavaScript-файла SPA – `webapp/public/js/spa.js`

```

/*
 * spa.js
 * Модуль корневого пространства имен
 */
...
/*global $, spa:true */

spa = (function () {
    'use strict';
    var initModule = function ( $container ) {
        spa.data.initModule();
        spa.model.initModule();

        if ( spa.shell && $container ) {
            spa.shell.initModule( $container );
        }
    };

    return { initModule: initModule };
})();

```

Добавляем в конфигурацию `spa:true`, чтобы JSLint не возражал против присваивания значения глобальной переменной `spa`.

Удаляем объявление `var`.

Модифицируем приложение так, чтобы оно могло работать без пользовательского интерфейса (Shell).

После того как мы завели переменную `global.spa`, можно загрузить модули почти так же, как это делалось в головном HTML-документе (`webapp/public/spa.html`). Сначала загружаем сторонние модули – jQuery, TaffyDB и т. п. – и обеспечиваем доступность *их* глобальных

переменных тоже (jQuery, \$ и TAFFY, если вам интересно). Затем можно загружать подключаемые модули jQuery и наши собственные модули SPA. Мы не загружаем ни Shell, ни функциональные модули, потому что для тестирования модели они не нужны. Подправим наш файл с автономными тестами, пока не забыли. Изменения выделены **полужирным** шрифтом.

Листинг Б.6 ❖ Добавление библиотек и наших модулей – webapp/public/nodeunit_suite.js

```
...
/*global $, spa */

// сторонние модули и глобальные переменные
global.jQuery = require( 'jquery' );
global.TAFFY = require( './js/jq/taffydb-2.6.2.js' ).taffy;
global.$      = global.jQuery;
require( './js/jq/jquery.event.gevent-0.1.9.js' );

// наши модули и глобальные переменные
global.spa = null;
require( './js/spa.js' );
require( './js/spa.util.js' );
require( './js/spa.fake.js' );
require( './js/spa.data.js' );
require( './js/spa.model.js' );

// пример кода
spa.initModule();
spa.model.setDataMode( 'fake' );

var $t = $( '<div/>' );
$.gevent.subscribe(
    $t, 'spa-login',
    function ( event, user ){
        console.log( 'Login user is:', user );
    }
);

spa.model.people.login( 'Fred' );
```

Да, кстати, мы тут немного забежали вперед и добавили в конец листинга коротенький тестовый скрипт. И хотя в конечном итоге мы будем запускать этот файл с помощью nodeunit, пока воспользуемся Node.js и убедимся, что библиотеки загружаются правильно. Да, действительно, при прогоне комплекта тестов с помощью Node.js печатается вот что:

```
$ node nodeunit_suite.js
Login user is: { cid: 'id_5',
  name: 'Fred',
  css_map: { top: 25, left: 25, 'background-color': '#8f8' },
  ___id: 'T000002R000003',
  ___s: true,
  id: 'id_5' }
```

Если вы прорабатываете примеры дома, наберитесь терпения. До появления первых результатов пройдет три секунды, потому что именно столько времени модуль ждет, прежде чем выполнить запрос на аутентификацию. И еще восемь секунд потребуется на полное завершение скрипта. Это также объясняется тем, что модуль Fake эмулирует работу сервера с помощью таймеров (создаваемых методами `setTimeout` и `setInterval`). Пока все таймеры не сработают, Node.js будет считать, что программа «работает», и не завершится. К этому вопросу мы еще вернемся позже, а пока познакомимся с nodeunit.

Б.4.2. Подготовка одного теста в nodeunit

Итак, мы научили Node.js загружать наши библиотеки и можем сосредоточиться на подготовке тестов к прогону с помощью nodeunit. Но сначала освоимся с самим каркасом nodeunit. Для прогона тестов нужно выполнить следующие действия:

- объявить тестовые функции;
- в каждой тестовой функции сообщить объекту `test`, сколько предполагается утверждений, воспользовавшись методом `test.expect(<count>);`;
- в каждом тесте выполнить утверждения, например: `test.ok(true);`;
- в конце каждого теста сообщить объекту `test`, что тест завершен, вызвав метод `test.done()`;
- экспортировать список тестов в порядке прогона. Каждый последующий тест запускается только после завершения предыдущего;
- прогнать комплект тестов командой `nodeunit <filename>`.

В листинге Б.7 показан скрипт nodeunit, в котором все эти шаги выполнены для одного теста. В аннотациях содержится полезная дополнительная информация.

Листинг Б.7 ❖ Первый пример теста для nodeunit – webapp/public/nodeunit_test.js

```
/*jslint node : true, sloppy : true, white : true */
```

```
// Тривиальный пример использования nodeunit
```

```
// Начало /testAcct/
var testAcct = function ( test ) {
  test.expect( 1 );
  test.ok( true, 'this passes' );
  test.done();
};
// Конец /testAcct/

module.exports = { testAcct : testAcct };
```

Объявляем тестовую функцию testAcct. Тест может называться как угодно, но это должна быть функция, которая принимает в качестве единственного аргумента объект test.

Сообщаем объекту test, что планируется выполнить только одно утверждение.

Вызываем метод test.done(), чтобы nodeunit мог перейти к следующему тесту (или выйти).

Вызываем первое (и единственное в этом примере) утверждение.

Экспортируем тесты в том порядке, в котором nodeunit должен их выполнить.

Выполнив команду `nodeunit nodeunit_test.js`, мы увидим на экране следующую информацию:

```
$ nodeunit_test.js
✓ testAcct
OK: 1 assertions (3ms)
```

Теперь применим полученные знания о nodeunit к коду, который хотим протестировать.

Б.4.3. Создание первого настоящего теста

Преобразуем предыдущий пример в настоящий тест. Чтобы обойти ловушки, подстерегающие при тестировании событийно-управляемого кода, мы можем воспользоваться nodeunit и отложенными объектами jQuery. Во-первых, мы полагаемся на тот факт, что nodeunit не переходит к следующему тесту, пока предыдущий не объявлен законченным с помощью метода `test.done()`. Благодаря этому тесты становятся проще и понятнее. Во-вторых, мы можем воспользоваться отложенным объектом jQuery (Deferred), чтобы вызывать `test.done()` только после опубликования обязательного события `spa-login`. Это даст возможность перейти к следующему тесту. Модифицируем комплект тестов, как показано в листинге Б.8. Изменения выделены полужирным шрифтом.

Листинг Б.8 ❖ Первый настоящий тест – `webapp/public/nodeunit_suite.js`

```
...
// наши модули и глобальные переменные
global.spa = null;
require( './js/spa.js' );
require( './js/spa.util.js' );
require( './js/spa.fake.js' );
require( './js/spa.data.js' );
require( './js/spa.model.js' );

// Начало /testAcct/. Инициализация и аутентификация
```



```
var testAcct = function ( test ) {
  var $t, test_str, user, on_login,
      $defer = $.Deferred();

  // задаем ожидаемое количество тестов
  test.expect( 1 );

  // определяем обработчик события 'spa-login'
  on_login = function () { $defer.resolve(); };

  // инициализация
  spa.initModule( null );
  spa.model.setDataMode( 'fake' );

  // создаем и подписываем объект jQuery
  $t = $( '<div/>' );
  $.gevent.subscribe( $t, 'spa-login', on_login );

  spa.model.people.login( 'Fred' );

  // проверяем, что пользователь больше не анонимный
  user = spa.model.people.get_user();
  test_str = 'user is no longer anonymous';
  test.ok( ! user.get_is_anon(), test_str );

  // объявляем, что тест завершен, после того как выполнена аутентификация
  $defer.done( test.done );
};

// Конец /testAcct/. Инициализация и аутентификация

module.exports = { testAcct : testAcct };
```

Прогнав этот комплект тестов командой `nodeunit ./nodeunit_suite.js`, мы увидим такой результат:

```
$ nodeunit nodeunit_test.js
✓ testAcct
OK: 1 assertions (3320ms)
```

Успешно реализовав первый тест, решим, какие тесты должны войти в комплект, и обсудим, как выполнить их в нужном порядке.

Б.4.4. План событий и тестов

Когда в главах 5 и 6 мы тестировали модель вручную, то ждали завершения процесса, прежде чем начинать ввод нового теста, — и это было естественно. Человеку кажется очевидным, что нужно дождаться завершения аутентификации и только потом переходить к тестированию.

нию обмена сообщениями. Но для комплекта тестов ничего очевидного здесь нет.

Мы должны продумать порядок следования событий и тестов. У работы над комплектом тестов есть, в частности, то преимущество, что она заставляет тщательно анализировать код, в результате чего мы начинаем лучше понимать его. Иногда мы находим больше ошибок в процессе самого написания тестов, чем при их прогоне.

Поэтому начнем с составления плана тестирования. Мы хотим проверить, как поведет себя модель, когда воображаемый пользователь Фред проходит во всем функциям нашего SPA. Вот каких действий мы ожидаем от Фреда:

- `testInitialState` – проверить начальное состояние модели;
- `loginAsFred` – аутентифицироваться от имени «Fred» и проверить объект пользователя до завершения процесса;
- `testUserAndPeople` – проверить список людей в онлайн и сведения о них;
- `testWilmaMsg` – получить сообщение от Wilma и проверить данные в нем;
- `sendPebblesMsg` – сделать собеседником Pebbles и послать ей сообщение;
- `testMsgToPebbles` – проверить содержимое сообщения, отправленного Pebbles;
- `testPebblesResponse` – проверить содержимое ответного сообщения, полученного от Pebbles;
- `updatePebblesAvtr` – изменить данные в аватаре Pebbles;
- `testPebblesAvtr` – проверить, правильно ли обновился аватар Pebbles;
- `logoutAsFred` – завершить сеанс Фреда;
- `testLogoutState` – проверить состояние модели после завершения сеанса.

Каркас `nodedunit` прогоняет тесты в указанном ему порядке и не переходит к следующему тесту, пока предыдущий не будет объявлен завершившимся. И это нам на руку, потому что мы хотим, чтобы определенные тесты начинались только после возникновения определенных событий. Например, проверять список людей в онлайн следует лишь после того, как произойдет событие аутентификации. В листинге Б.9 представлен план тестирования с указанием того, какие события должны произойти до начала каждого теста. Отметим, что имена тестов точно соответствуют названиям пунктов в приведенном выше плане и что они абсолютно понятны человеку.

Листинг Б.9 ❖ План тестирования с указанием блокирующих событий

```
// Начало /testInitialState/
// инициализировать SPA
// проверить пользователя в начальном состоянии
// проверить список людей в онлайн
// перейти к следующему тесту без блокировки
// Конец /testInitialState/

// Начало /loginAsFred/
// аутентифицироваться как 'Fred'
// проверить атрибуты пользователя до завершения аутентификации
// перейти к следующему тесту, когда выполнены два условия:
// + аутентификация завершена (событие spa-login)
// + список людей в онлайн обновлен (событие spa-listchange)
// Конец /loginAsFred/

// Начало /testUserAndPeople/
// проверить атрибуты пользователя
// проверить список людей в онлайн
// перейти к следующему тесту, когда выполнены два условия:
// + получено первое сообщение (событие spa-updatechat)
// (это сообщение от пользователя 'Wilma')
// + изменен собеседник (событие spa-setchatee)
// Конец /testUserAndPeople/

// Начало /testWilmaMsg/
// проверить сообщение, полученное от 'Wilma'
// проверить атрибуты собеседника
// перейти к следующему тесту без блокировки
// Конец /testWilmaMsg/

// Начало /sendPebblesMsg/
// сделать собеседником пользователя 'Pebbles'
// отправить 'Pebbles' сообщение методом send_msg
// проверить результаты get_chatee()
// перейти к следующему тесту, когда выполнены два условия:
// + собеседник установлен (событие spa-setchatee)
// + сообщение отправлено (событие spa-updatechat)
// Конец /sendPebblesMsg/

// Начало /testMsgToPebbles/
// проверить атрибуты собеседника
// проверить отправленное сообщение
// перейти к следующему тесту, когда выполнено условие:
// + получен ответ от 'Pebbles' (событие spa-updatechat)
// Конец /testMsgToPebbles/

// Начало /testPebblesResponse/
// проверить сообщение, полученное от 'Pebbles'
// перейти к следующему тесту без блокировки
```

```
// Конец /testPebblesResponse/

// Начало /updatePebblesAvtr/
// вызвать метод update_avatar
// перейти к следующему тесту, когда выполнено условие:
// + список людей в онлайн обновлен (событие spa-listchange)
// Конец /updatePebblesAvtr/

// Начало /testPebblesAvtr/
// получить объект пользователя 'Pebbles' с помощью метода get_chatee
// проверить данные об аватаре 'Pebbles'
// перейти к следующему тесту без блокировки
// Конец /testPebblesAvtr/

// Начало /logoutAsFred/
// закончить сеанс Фреда
// перейти к следующему тесту, когда выполнено условие:
// + выход из сеанса завершен (событие spa-logout)
// Конец /logoutAsFred/

// Начало /testLogoutState/
// проверить список людей в онлайн
// проверить атрибуты пользователя
// продолжить без блокировки
// Конец /testLogoutState/
```

Б.4.5. Создание комплекта тестов

Теперь можно добавить кое-какие служебные методы и постепенно расширять комплект тестов. На каждом шаге мы будем прогонять весь комплект, проверяя, все ли правильно.

Тесты начального состояния и аутентификации

Для начала напомним несколько служебных методов и добавим первые три теста: проверка начального состояния модели, аутентификация от имени Фреда и последующая проверка атрибутов пользователя и списка людей в онлайн. Наш опыт показывает, что тесты обычно можно отнести к одной из двух категорий:

1. Проверочные тесты, в которых много утверждений (типа `user.name === 'Fred'`), служат для проверки правильности данных программы. Зачастую они неблокирующие.
2. Тесты управления, которые выполняют различные действия: аутентификацию, отправку сообщения или обновление аватара. В них редко бывает много утверждений, зато они часто блокируют выполнение, пока не будет удовлетворено условие, зависящее от некоторого события.

Мы считаем, что имеет смысл отразить это естественное деление, и называем тесты соответственно. Проверочные тесты имеют имена вида `test<something>`, а имена тестов управления отражают выполняемое в них действие, например `loginAsFred`.

Тест `loginAsFred` требует, чтобы аутентификация завершилась и список людей в онлайн обновился, и только после этого позволяет `nodeunit` перейти к тесту `testUserAndPeople`. Для реализации этой задумки мы подписываем коллекцию jQuery `$t` на события `spa-login` и `spa-listchange`. Затем мы используем отложенные объекты jQuery, которые гарантируют, что эти события произойдут до вызова `test.done()` из теста `loginAsFred`.

Изменения выделены в листинге Б.10 полужирным шрифтом. Как обычно, читайте аннотации, потому что в них содержится полезная дополнительная информация.

Листинг Б.10 ❖ Добавление первых двух тестов – `webapp/public/nodeunit_suite.js`

```
...
/*global $, spa */

// сторонние модули и глобальные переменные
...
// наши модули и глобальные переменные
...
var
  // служебные методы и обработчики
  makePeopleStr, onLogin, onListchange,

  // тестовые функции
  testInitialState, loginAsFred, testUserAndPeople,

  // обработчики событий
  loginEvent, changeEvent, loginData, changeData,

  // индексы
  changeIdx = 0,

  // отложенные объекты
  $deferLogin = $.Deferred(),
  $deferChangeList = [ $.Deferred() ];

// служебный метод для построения строки имен людей в онлайн
```

Объявляем первые три тестовых метода, назвав их так, чтобы сгенерированный отчет был понятен.

Добавляем служебный метод `makePeopleStr`. Он формирует строку из имен пользователей в коллекции `TaffyDB`. Это позволит проверить список людей в онлайн, просто сравнив строки.

```

  makePeopleStr = function ( people_db ) {
    var people_list = [];
    people_db().each(function( person, idx ) {
      people_list.push( person.name );
    });
  };
}
```

```

    });
    return people_list.sort().join( ', ' );
};

// обработчик события 'spa-login' ←
onLogin = function ( event, arg ) {
    loginEvent = event;
    loginData = arg;
    $deferLogin.resolve();
};

// обработчик события 'spa-listchange' ←
onListChange = function ( event, arg ) {
    changeEvent = event;
    changeData = arg;
    $deferChangeList[ changeIdx ].resolve();
    changeIdx++;
    $deferChangeList[ changeIdx ] = $.Deferred();
};

// Начало /testInitialState/
testInitialState = function ( test ) {
    var $t, user, people_db, people_str, test_str;
    test.expect( 2 );

    // инициализировать SPA
    spa.initModule( null );
    spa.model.setDataMode( 'fake' );

    // создать объект jQuery
    $t = $( '<div/>' ); ←
    // подписаться на глобальные пользовательские события
    $.gevent.subscribe( $t, 'spa-login', onLogin );
    $.gevent.subscribe( $t, 'spa-listchange', onListchange );

    // проверить пользователя в начальном состоянии
    user = spa.model.people.get_user();
    test_str = 'user is anonymous';
    test.ok( user.get_is_anon(), test_str );

    // проверить список пользователей в онлайн
    test_str = 'expected user only contains anonymous';
    people_db = spa.model.people.get_db();
    people_str = makePeopleStr( people_db );
    test.ok( people_str === 'anonymous', test_str );

    // перейти к следующему тесту без блокировки
    test.done(); ←
};

```

Добавляем метод для обработки глобального пользовательского события spa-login. Он вызывает метод \$deferLogin.resolve().

Добавляем метод для обработки глобального пользовательского события spa-listchange. Он вызовет \$deferChangeList[idxChange].resolve(), а затем поместит новый отложенный объект jQuery в конец списка \$deferChangeList, соответствующего будущим событиям spa-listchange.

Создаем коллекцию jQuery \$t, которую подпишем на глобальные пользовательские события.

Подписываемся на глобальные пользовательские события jQuery, чтобы узнать о завершении теста loginAsFred. Событие spa-login обрабатывается методом onLogin, а событие spa-listchange – методом onListchange.

Тест testInitialState не блокирует выполнение, поскольку вызывает метод test.done() безусловно.

```

// Конец /testInitialState/

// Начало /loginAsFred/
loginAsFred = function ( test ) {
    var user, people_db, people_str, test_str;
    test.expect( 6 );

    // аутентифицироваться как 'Fred'
    spa.model.people.login( 'Fred' );
    test_str = 'log in as Fred';
    test.ok( true, test_str );

    // проверить атрибуты пользователя до завершения аутентификации
    user = spa.model.people.get_user();
    test_str = 'user is no longer anonymous';
    test.ok( ! user.get_is_anon(), test_str );

    test_str = 'usr name is "Fred"';
    test.ok( user.name === 'Fred', test_str );

    test_str = 'user id is undefined as login is incomplete';
    test.ok( ! user.id, test_str );

    test_str = 'user cid is c0';
    test.ok( user.cid === 'c0', test_str );

    test_str = 'user list is as expected';
    people_db = spa.model.people.get_db();
    people_str = makePeopleStr( people_db );
    test.ok( people_str === 'Fred,anonymous', test_str );

    // перейти к следующему тесту, когда выполнены два условия:
    // + аутентификация завершена (событие spa-login)
    // + список людей в онлайн обновлен (событие spa-listchange)
    $.when( $deferLogin, $deferChangeList[ 0 ] ) <-
        .then( test.done );
};
// Конец /loginAsFred/

// Начало /testUserAndPeople/
testUserAndPeople = function ( test ) {
    var
        user, cloned_user,
        people_db, people_str,
        user_str, test_str;
    test.expect( 4 );

    // проверить атрибуты пользователя
    test_str = 'login as Fred complete';
    test.ok( true, test_str );

```

В тесте loginAsFred используются отложенные объекты jQuery, гарантирующие, что ожидаемые события произойдут до вызова test.done(). Должен завершиться процесс аутентификации (\$deferLogin.is_resolved() === true) и обновиться список людей в онлайн (\$deferChangeList[0].is_resolved === true). Это требование реализуется предложением \$.when(<deferred objects>).then(<function>).

Проверяем атрибуты аутентифицированного пользователя и списка людей в онлайн.

```

user      = spa.model.people.get_user();
test_str  = 'Fred has expected attributes';
cloned_user = $.extend( true, {}, user );

delete cloned_user.__id;
delete cloned_user.__s;
delete cloned_user.get_is_anon;
delete cloned_user.get_is_user;

test.deepEqual(
  cloned_user,
  { cid      : 'id_5',
    css_map : { top: 25, left: 25, 'background-color': '#8f8' },
    id      : 'id_5',
    name    : 'Fred'
  },
  test_str
);

// проверить список людей в онлайнe
test_str = 'receipt of listchange complete';
test.ok( true, test_str );

people_db = spa.model.people.get_db();
people_str = makePeopleStr( people_db );
user_str = 'Betty,Fred,Mike,Pebbles,Wilma';
test_str = 'user list provided is expected - ' + user_str;

test.ok( people_str === user_str, test_str );

test.done(); ←
};
// Конец /testUserAndPeople/

module.exports = {
  testInitialState : testInitialState, ←
  loginAsFred      : loginAsFred,
  testUserAndPeople : testUserAndPeople
};
// Конец комплекта тестов

```

Пока не блокируем тест testUserAndPeople, потому что это последний тест в цепочке. Когда будут добавлены новые тесты, мы внесем сюда изменения.

Экспортируем тесты в том порядке, в котором они должны выполняться. При прогоне комплекта тестов с помощью nodeunit печатаются их имена.

Прогнав этот комплект тестов с помощью команды `nodeunit nodeunit_suite.js`, мы увидим такую картину:

```

$ nodeunit nodeunit_suite.js
✓ testInitialState
✓ loginAsFred
✓ testUserAndPeople

```

OK: 12 assertions (4223ms)


```

onUpdatechat = function ( event, arg ) {
    msgEvent = event;
    msgData = arg;
    $deferMsgList[ msgIdx ].resolve();
    msgIdx++;
    $deferMsgList[ msgIdx ] = $.Deferred();
};

// обработчик события 'spa-setchatee' ←
onSetchatee = function ( event, arg ) {
    chateeEvent = event;
    chateeData = arg;
    $deferChateeList[ chateeIdx ].resolve();
    chateeIdx++;
    $deferChateeList[ chateeIdx ] = $.Deferred();
};

// Начало /testInitialState/
testInitialState = function ( test ) {
    ...
    // подписаться на глобальные пользовательские события
    $.gevent.subscribe( $t, 'spa-login', onLogin );
    $.gevent.subscribe( $t, 'spa-listchange', onListchange );
    $.gevent.subscribe( $t, 'spa-setchatee', onSetchatee ); ←
    $.gevent.subscribe( $t, 'spa-updatechat', onUpdatechat ); ←
    ...
};
// Конец /testInitialState/

...
// Начало /testUserAndPeople/
testUserAndPeople = function ( test ) {
    ...
    test.ok( people_str === user_str, test_str );

    // перейти к следующему тесту, когда выполнены два условия: ←
    // + получено первое сообщение (событие spa-updatechat)
    // (это сообщение от пользователя 'Wilma')
    // + изменен собеседник (событие spa-setchatee)
    $.when($deferMsgList[ 0 ], $deferChateeList[ 0 ] )
        .then( test.done );
};
// Конец /testUserAndPeople/

// Начало /testWilmaMsg/ ←
testWilmaMsg = function ( test ) {
    var test_str;
    test.expect( 4 );

    // проверить сообщение, полученное от 'Wilma'

```

Добавляем обработчик глобального пользовательского события spa-setchatee. Он вызывается после изменения собеседника по любой причине. Это может произойти, когда пользователь явно выбирает нового собеседника, или если текущий собеседник выходит из чата, или в результате получения сообщения от кого-то, кроме текущего собеседника.

Подписываем коллекцию jQuery \$t на глобальное пользовательское событие spa-setchatee, указывая обработчик onSetchatee.

Подписываем коллекцию jQuery \$t на глобальное пользовательское событие spa-updatechat, указывая обработчик onUpdatechat.

Невыходим из теста testUserAndPeople, пока не будет обработано первое сообщение и не произойдет первая смена собеседника. Для реализации блокировки используем отложенными объектами jQuery и конструкцией \$.when().then().

Добавляем тест для проверки сообщения от Wilma и атрибутов нового собеседника.

```

test_str = 'Message is as expected';
test.deepEqual(
  msgData,
  { dest_id: 'id_5',
    dest_name: 'Fred',
    sender_id: 'id_04',
    msg_text: 'Hi there Fred! Wilma here.'
  },
  test_str
);

// проверить атрибуты собеседника
test.ok( chateeData.new_chatee.cid === 'id_04' );
test.ok( chateeData.new_chatee.id === 'id_04' );
test.ok( chateeData.new_chatee.name === 'Wilma' );

// перейти к следующему тесту без блокировки
test.done();
};
// Конец /testWilmaMsg/

// Начало /sendPebblesMsg/
sendPebblesMsg = function ( test ) {
  var test_str, chatee;
  test.expect( 1 );

  // сделать собеседником пользователя 'Pebbles'
  spa.model.chat.set_chatee( 'id_03' );

  // отправить 'Pebbles' сообщение методом send_msg
  spa.model.chat.send_msg( 'whats up, tricks?' );

  // проверить результаты get_chatee()
  chatee = spa.model.chat.get_chatee();
  test_str = 'Chatee is as expected';
  test.ok( chatee.name === 'Pebbles', test_str );

  // перейти к следующему тесту, когда выполнены два условия:
  // + собеседник установлен (событие spa-setchatee)
  // + сообщение отправлено (событие spa-updatechat)
  $.when( $deferMsgList[ 1 ], $deferChateeList[ 1 ] )
    .then( test.done );
};
// Конец /sendPebblesMsg/

// Начало /testMsgToPebbles/
testMsgToPebbles = function ( test ) {
  var test_str;
  test.expect( 2 );

```

Переходим к следующему тесту без блокировки.

Добавляем тест sendPebblesMsg, в котором Fred делает собеседником Pebbles и отправляет ей сообщение. В отличие от большинства тестов, выполняющих действия, здесь есть несколько утверждений, и выполнение блокируется до возникновения событий.

Не выходим из теста sendPebblesMsg, пока не будет обработано второе сообщение и не произойдет вторая смена собеседника. Для реализации блокировки пользуемся отложенными объектами JQuery и конструкцией \$.when().then().

Добавляем тест для проверки сообщения, отправленного в адрес Pebbles.

```

// проверить атрибуты собеседника
test_str = 'Pebbles is the chatee name';
test.ok(
  chateeData.new_chatee.name === 'Pebbles',
  test_str
);

// проверить отправленное сообщение
test_str = 'message change is as expected';
test.ok( msgData.msg_text === 'whats up, tricks?', test_str );

// перейти к следующему тесту, когда выполнено условие:
// + получен ответ от 'Pebbles' (событие spa-updatechat)
$deferMsgList[ 2 ].done( test.done );
};
// Конец /testMsgToPebbles/

// Начало /testPebblesResponse/
testPebblesResponse = function ( test ) {
  var test_str;
  test.expect( 1 );

  // проверить сообщение, полученное от 'Pebbles'
  test_str = 'Message is as expected';
  test.deepEqual(
    msgData,
    { dest_id: 'id_5',
      dest_name: 'Fred',
      sender_id: 'id_03',
      msg_text: 'Thanks for the note, Fred'
    },
    test_str
  );

  // перейти к следующему тесту без блокировки
  test.done();
};
// Конец /testPebblesResponse/

module.exports = {
  testInitialState : testInitialState,
  loginAsFred      : loginAsFred,
  testUserAndPeople : testUserAndPeople,
  testWilmaMsg      : testWilmaMsg,
  sendPebblesMsg     : sendPebblesMsg,
  testMsgToPebbles   : testMsgToPebbles,
  testPebblesResponse : testPebblesResponse
};
// Конец комплекта тестов

```

Невыходим из теста testMsgToPebbles, пока не будет обработано третье сообщение (ответ от Pebbles).

Добавляем тест testPebblesResponse для проверки сообщения, полученного от Pebbles.

Выходим из testPebblesResponse без блокировки.

Добавляем в комплект новые тесты.

Прогнав этот комплект тестов с помощью команды `nodeunit nodeunit_suite.js`, мы увидим такую картину:

```
$ nodeunit nodeunit_suite.js
✓ testInitialState
✓ loginAsFred
✓ testUserAndPeople
✓ testWilmaMsg
✓ sendPebblesMsg
✓ testMsgToPebbles
✓ testPebblesResponse
```

```
OK: 20 assertions (14233ms)
```

Выполнение этого комплекта занимает столько же времени, сколько и раньше, но мы видим новые тесты. Теперь мы ждем сообщения от Wilma и проверяем его. Для завершения комплекта осталось добавить еще несколько тестов.

Добавление тестов аватаров, завершения сеанса и состояния после завершения сеанса

Пора добавить оставшиеся четыре теста из нашего плана. Как и раньше, для ожидания событий мы пользуемся отложенными объектами. Новые тесты приведены в листинге Б.12. Изменения выделены **полужирным** шрифтом.

Листинг Б.12 ❖ Дополнительные тесты – webapp/public/nodeunit_suite.js

```
...
var
  // служебные методы и обработчики
  makePeopleStr, onLogin,      onListchange,
  onSetchatee,   onUpdatechat, onLogout,

  // тестовые функции
  testInitialState,    loginAsFred,      testUserAndPeople,
  testWilmaMsg,        sendPebblesMsg,   testMsgToPebbles,
  testPebblesResponse, updatePebblesAvtr, testPebblesAvtr,
  logoutAsFred,        testLogoutState,

  // обработчики событий
  loginEvent, changeEvent, chateeEvent, msgEvent, logoutEvent,
  loginData,  changeData, msgData, chateeData, logoutData,

  ...
  $deferMsgList = [ $.Deferred() ],
  $deferLogout = $.Deferred();
```

```

...
// обработчик события 'spa-setchatee'
...
// обработчик события 'spa-logout'
onLogout = function ( event, arg ) {
    logoutEvent = event;
    logoutData = arg;
    $deferLogout.resolve();
};

// Начало /testInitialState/
testInitialState = function ( test ) {
    ...
    $.gevent.subscribe( $t, 'spa-updatechat', onUpdatechat );
    $.gevent.subscribe( $t, 'spa-logout', onLogout );

    // проверяем пользователя в начальном состоянии
    ...
// Конец /testPebblesResponse/

// Начало /updatePebblesAvtr/
updatePebblesAvtr = function ( test ) {
    test.expect( 0 );

    // вызвать метод update_avatar
    spa.model.chat.update_avatar({
        person_id : 'id_03',
        css_map : {
            'top' : 10, 'left' : 100,
            'background-color' : '#ff0'
        }
    });

    // перейти к следующему тесту, когда выполнено условие:
    // + список людей в онлайн обновлен (событие spa-listchange)
    $deferChangeList[ 1 ].done( test.done );
};
// Конец /updatePebblesAvtr/

// Начало /testPebblesAvtr/
testPebblesAvtr = function ( test ) {
    var chatee, test_str;
    test.expect( 1 );

    // получить объект пользователя 'Pebbles' с помощью метода get_chatee
    chatee = spa.model.chat.get_chatee();

    // проверить данные об аватаре 'Pebbles'
    test_str = 'avatar details updated';
    test.deepEqual(

```

```
    chatee.css_map,
    { top : 10, left : 100,
      'background-color' : '#ff0'
    },
    test_str
  );

  // перейти к следующему тесту без блокировки
  test.done();
};
// Конец /testPebblesAvtr/

// Начало /logoutAsFred/
logoutAsFred = function( test ) {
test.expect( 0 );

  // закончить сеанс Фреда
  spa.model.people.logout();

  // перейти к следующему тесту, когда выполнено условие:
  // + выход из сеанса завершен (событие spa-logout)
  $deferLogout.done( test.done );
};
// Конец /logoutAsFred/

// Начало /testLogoutState/
testLogoutState = function ( test ) {
  var user, people_db, people_str, user_str, test_str;
  test.expect( 4 );

  test_str = 'logout as Fred complete';
  test.ok( true, test_str );

  // проверить список людей в онлайн
  people_db = spa.model.people.get_db();
  people_str = makePeopleStr( people_db );
  user_str = 'anonymous';
  test_str = 'user list provided is expected - ' + user_str;

  test.ok( people_str === 'anonymous', test_str );

  // проверить атрибуты пользователя
  user = spa.model.people.get_user();
  test_str = 'current user is anonymous after logout';
  test.ok( user.get_is_anon(), test_str );
  test.ok( true, 'test complete' );

  // продолжить без блокировки
  test.done();
};
```

```
// Конец /testLogoutState/

module.exports = {
  testInitialState : testInitialState,
  loginAsFred      : loginAsFred,
  testUserAndPeople : testUserAndPeople
  testWilmaMsg      : testWilmaMsg,
  sendPebblesMsg     : sendPebblesMsg,
  testMsgToPebbles   : testMsgToPebbles,
  testPebblesResponse : testPebblesResponse,
  updatePebblesAvtr   : updatePebblesAvtr,
  testPebblesAvtr     : testPebblesAvtr,
  logoutAsFred       : logoutAsFred,
  testLogoutState    : testLogoutState
};
// Конец комплекта тестов
```

Прогнав этот комплект тестов с помощью команды `nodeunit nodeunit_suite.js`, мы увидим такую картину:

```
$ nodeunit nodeunit_suite.js
✓ testInitialState
✓ loginAsFred
✓ testUserAndPeople
✓ testWilmaMsg
✓ sendPebblesMsg
✓ testMsgToPebbles
✓ testPebblesResponse
✓ updatePebblesAvtr
✓ testPebblesAvtr
✓ logoutAsFred
✓ testLogoutState
```

```
OK: 25 assertions (14234ms)
```

Мы закончили запланированный комплект тестов. Теперь мы можем автоматически прогонять его перед сохранением обновлений в репозитории (подключив скрипт к точке фиксации). Такой подход не замедляет, а, наоборот, *ускоряет* разработку, так как предотвращает возврат назад и гарантирует качество кода. Это пример *встраивания* качества в продукт на стадии проектирования в противоположность тестированию продукта после его «завершения».

Увы, осталась одна вопиющая проблема: этот комплект тестов никогда не завершается. На экране-то мы видим сообщение о 25 проведенных утверждениях, но управление не возвращается ни терминалу, ни любому другому вызывающему процессу. Из-за этого автоматизировать прогон тестов не получится. В следующем разделе мы обсудим, почему так происходит и что с этим можно сделать.

Б.5. Адаптация модулей SPA для тестирования

Node.js (а следовательно, и nodeunit) должны ответить на один животрепещущий вопрос: *как узнать, что выполнение комплекта тестов завершилось?* Это пример классической *проблемы останова* в информатике. В любом событийно-управляемом языке она нетривиальна. Вообще говоря, Node.js считает приложение завершившимся, когда не осталось подлежащего выполнению кода и нет ожидающих операций.

До сих пор мы предполагали, что программа продолжает работать, пока пользователь не закроет вкладку браузера, и больше никаких условий выхода не рассматривали. Если тестировщик работает в режиме 2 (тестирование в браузере с подставными данными) и завершает сеанс, то модуль Fake вызывает метод `setTimeout` в ожидании следующей аутентификации.

Наш комплект тестов, как зрители некоторых фильмов, ожидает явно увидеть титры со словом «Конец». Поэтому если мы *вообще* хотим, чтобы прогон завершался, не дожидаясь сигнала `SIGTERM` или `SIGKILL`, необходима какая-то *тестовая оправка*¹. Так называется конфигурация или директива, необходимые для тестирования, но не нужные в производственном режиме.

Естественно, мы хотели бы минимизировать количество тестовых оправок, чтобы не вносить вместе с ними новых ошибок. Но иногда без них не обойтись. В данном случае нам нужна тестовая оправка, которая не давала бы модулю Fake постоянно перезапускать таймеры. Тогда наш комплект тестов сможет завершиться, и у нас появится возможность автоматизировать его запуск и интерпретировать результаты.

Чтобы предотвратить переустановку таймеров после завершения сеанса в модуле Fake, мы можем предпринять следующие действия.

- В комплекте тестов передавать аргумент `true` методу завершения сеанса: `spa.model.people.logout(true)`. Этот аргумент (который мы будем называть флагом *do_not_reset* – не восстанавливать) говорит модели, что после завершения сеанса она не

¹ Уточним ситуацию: нам нужно, чтобы эта программа завершалась, потому что система управления версиями анализирует результат скрипта, вызванного из точки подключения, по коду завершения. Раз нет завершения, то нет и кода завершения, а значит, нет и автоматизации, что, конечно, неприемлемо.

должна восстанавливать начальное состояние для подготовки к следующей аутентификации.

- Метод модели `spa.model.people.logout` должен принимать обязательный аргумент `do_not_reset`. Он передается без изменения методу `chat._leave`.
- Метод модели `spa.model.chat._leave` также должен принимать необязательный аргумент `do_not_reset`. Он передается серверу в качестве данных вместе с сообщением `leavechat`.
- Изменить модуль Fake (`webapp/public/js/spa.fake.js`), так чтобы функция обратного вызова `leavechat` интерпретировала полученные данные как флаг `do_not_reset`. Функция `leavechat`, видя, что ей передано значение `true`, *не должна* перезапускать таймеры после завершения сеанса.

Работы оказалось немного больше, чем хотелось бы (хотелось бы вообще ничего дополнительно не делать), но все равно это лишь мелкие изменения в трех файлах. В листинге Б.13 показано, как добавить флаг `do_not_reset` при вызове метода `logout` в комплекте тестов.

Листинг Б.13 ❖ Добавление флага `do_not_reset` в комплект тестов – `webapp/public/nodeunit_suite.js`

```
...
// Начало /logoutAsFred/
logoutAsFred = function( test ) {
test.expect( 0 );

    // закончить сеанс Фреда
    spa.model.people.logout( true );

    // перейти к следующему тесту, когда выполнено условие:
    // + выход из сеанса завершен (событие spa-logout)
    $deferLogout.done( test.done );
};
// Конец /logoutAsFred/
...
```

Теперь добавим аргумент `do_not_reset` в модель, как показано в следующем листинге.

Листинг Б.14 ❖ Добавление флага `do_not_reset` в модель – `webapp/public/nodeunit_suite.js`

```
...
people = (function () {
...
logout = function ( do_not_reset ) {
    var user = stateMap.user;
    chat._leave( do_not_reset );
}
```

```
stateMap.user = stateMap.anon_user;
clearPeopleDb();

$.gevent.publish( 'spa-logout', [ user ] );
};
...
}());
...
chat = (function () {
...
  _leave_chat = function ( do_not_reset ) {
    var sio = isFakeData ? spa.fake.mockSio : spa.data.getSio();
    chatee = null;
    stateMap.is_connected = false;
    if ( sio ) { sio.emit( 'leavechat', do_not_reset ); }
  };
  ...
}());
...
```

И наконец, изменим модуль Fake так, чтобы он учитывал флаг `do_not_reset` при отправке сообщения `leavechat`.

Листинг Б.15 ❖ Добавление флага `do_not_reset` в модуль Fake – `webapp/public/nodeunit_suite.js`

```
...
mockSio = (function () {
...
  emit_sio = function ( msg_type, data ) {
    ...
    if ( msg_type === 'leavechat' ) {
      // восстанавливаем состояние "не аутентифицирован"
      delete callback_map.listchange;
      delete callback_map.updatechat;

      if ( listchange_idto ) {
        clearTimeout( listchange_idto );
        listchange_idto = undefined;
      }
      if ( ! data ) { send_listchange(); }
    }
    ...
  }
}
```

Если после этих изменений мы выполним команду `nodeunit nodeunit_suite.js`, то увидим, что все тесты прошли и комплект *завершился*:

```
$ nodeunit nodeunit_suite.js
✓ testInitialState
✓ loginAsFred
```

- ✓ testUserAndPeople
- ✓ testWilmaMsg
- ✓ sendPebblesMsg
- ✓ testMsgToPebbles
- ✓ testPebblesResponse
- ✓ updatePebblesAvtr
- ✓ testPebblesAvtr
- ✓ logoutAsFred
- ✓ testLogoutState

OK: 25 assertions (14234ms)

\$

Код выхода комплекта тестов – это количество оказавшихся неверными утверждений. Поэтому если все тесты прошли, то на выходе мы получим код 0 (в Linux и Mac распечатать код выхода позволяет команда `echo $?`). Скрипт может в зависимости от кода выхода (и другой выходной информации) выполнять различные действия: запретить развертывание сборки или отправить по электронной почте сообщение ответственному разработчику или руководителю проекта.

Б.6. Резюме

Тестирование помогает нам быстрее разрабатывать более качественные программы. В правильно управляемом проекте с самого начала предусматривается несколько режимов тестирования, а тесты пишутся одновременно с кодом, чтобы выявлять и устранять ошибки быстро и эффективно. Наверное, почти каждому доводилось когда-то работать над проектом, где чуть ли не каждый шаг вперед сопровождается ошибкой в части программы, которая *раньше* работала. Систематический прогон продуманных тестов на ранних стадиях разработки позволяет устранить регрессию и способствует быстрому продвижению.

В этом приложении мы продемонстрировали четыре режима тестирования и обсудили, как их настроить и когда использовать. В качестве каркаса тестирования мы выбрали `nodeunit`. Затем мы сумели протестировать модель без участия браузера. В комплекте тестов мы использовали отложенные объекты `jQuery` и предоставляемые каркасом методы, чтобы тесты выполнялись в правильном порядке. Наконец, мы показали, как подправить модули, чтобы комплект тестов корректно работал в тестовой среде.

Надеемся, что после этой презентации в голове у вас прояснилось и появились новые идеи. Удачного тестирования!

Предметный указатель

Символы

\, 433
 focus, псевдокласс, 164
 hover, псевдокласс, 165
 id, параметр, 315
#!, 398
\$ переменная, 82
\$ символ, 198, 444
? (вопросительный знак), 319
, (запятая), 58
* (звездочка), 319
% (знак процента), 162
[] квадратные скобки, 430, 448
() круглые скобки, 430, 433
- (минус), 460
' (одиночные кавычки), 433
!= (оператор), 459
!== (оператор), 459
== (оператор), 459
=== (оператор), 459
+ (плюс), 460
_gaq, объект, 401
_leave_chat, метод, 242
__proto__, свойство, 73
_publish_listchange, метод, 242
_publish_updatechat, метод, 250
_trackEvent, метод, 401
_trackPageView, метод, 401
_update_list, метод, 242
_ (подчерк), 440
(решетка), 127
& символ, 198
<> символы, 198
{ } фигурные скобки, 430, 448

A

ActionScript, 31
action, параметр, 402
additionalProperties, атрибут, 361
AddThis, 139
adduser, обработчик сообщения,
модуль Chat, 382

Airbrake, 403
AJAX, метод, 45
Akamai, 406
alert, метод, 68
all, метод Express, 318
Amazon Cloudfront, 406
anon_user, ключ, 215
Apache/Apache2, 297, 333
Avatar, функциональный модуль
 JavaScript-код, 278
 добавление поддержки в модель
 изменения в объекте chat, 256
 использование модуля Fake, 258
 тестирование, 259
 модификация модуля Shell, 285
 назначение, 191
 обзор, 277
 таблица стилей, 284

B

basicAuth, метод, 328
bodyParser, метод, 309
body, элемент, 39
break, предложение, 458
Bugsense, 403

C

C, язык программирования, 85
Cache-Control, заголовок, 411
Cappuccino, 33
Cassandra, 340
Catalyst, 141
category, параметр, 402
CDN Planet, сайт, 406
CDN (сеть доставки
содержимого), 42, 406
chat, объект модели
 добавление средств работы
 с сообщениями, 247
 документирование, 239
 использование модуля
 Fake, 243, 252

- методы, 239
 - обзор, 235
 - реализация, 240
 - события, 238
 - тестирование, 254
 - тестирование метода join, 246
 - Chat, функциональный модуль
 - adduser сообщение,
 - обработчик, 382
 - API инициализации, 156
 - API конфигурирования, 155
 - handleResize, метод, 183
 - removeSlider, метод, 181
 - setSliderPosition, метод, 157
 - updateavatar сообщение,
 - обработчик, 390
 - updatechat сообщение,
 - обработчик, 386
 - добавление файлов, 146
 - каскадное конфигурирование
 - и инициализация, 158
 - модификация JavaScript-кода, 263
 - модификация модуля Shell, 172
 - модификация таблиц стилей, 271
 - назначение, 190
 - обзор, 262
 - отключение, обработчик
 - сообщений, 388
 - паттерн якорного интерфейса, 153
 - реализация, 378
 - структура каталогов и файлов, 145
 - таблицы стилей, 160
 - тестирование, 276
 - cid, свойство объекта person, 203
 - clearChat, метод, 263
 - close, метод, 68
 - Cloudfront, 406
 - CoffeeScript, 33
 - collection_name, MongoDB, 346
 - completeLogin(), метод, 219
 - configMap,
 - переменная, 48, 111, 120, 146
 - configure, метод, 308
 - Connect, каркас для Node.js, 302
 - connect, метод, 333
 - console.log(), функция, 59
 - Content-Type, свойство, 300, 313, 317
 - continue, предложения, 456
 - CouchDB, 32
 - countUp, функция, 331
 - createServer, метод, 300
 - create, метод, 71
 - CRUD, операции
 - маршруты
 - для обновления объекта
 - пользователя, 316
 - для получения списка
 - пользователей, 313
 - для создания объекта
 - пользователя, 313
 - для удаления объекта
 - пользователя, 316
 - для чтения объекта
 - пользователя, 315
 - обобщенный
 - идентификатор, 319
 - перенос в отдельный
 - модуль, 324
 - методы драйвера MongoDB, 350
 - модуль базы данных
 - обзор, 368
 - перенос кода, 372
 - структура файлов, 369
 - определение, 312
 - CSS3, 32
 - css_map, свойство, 203
 - CSS (каскадные таблицы стилей), 31
 - em, единица измерения, 162
 - для модуля Shell, 106, 113
 - для функционального модуля
 - Chat, 160
 - css, каталог, 99
 - curl, программа, 314
 - curry, префикс, 446
- D**
- Data, модуль, разработка, 288
 - decodeHtml, служебная
 - функция, 198
 - DELETE, глагол, 312
 - DisQus, 139, 144
 - Django, 141
 - document, 68
 - document, элемент и модель, 193

DOM (объектная модель документа), 40

- использование jQuery, 454
- методы, 464
- хранилище, 408

DoubleClick, 139

do, предложения, 456

DTD (определение типа документа), 360

Dust, система шаблонов, 271

E

Edgecast, 406

em, единица измерения, 162, 166, 184

encodeURIComponent, служебная функция, 198

end, метод, 300

Errorception, 404

errorHandler, метод, 310

eval, функция, 460

exports, атрибут, 323

Express, каркас для Node.js

- добавление промежуточного уровня, 308
- обслуживание статических файлов, 310
- общие сведения, 305
- окружения, 309

extend(), метод, 130

F

Facebook

- аутентификация, 295
- кнопка \, 139

Fake модуль, использование в модели, 212, 243, 252

find, метод, MongoDB, 346

Firebug, 38

Firefox

- и быстроедействие JavaScript, 32
- метод Object.create(), 72

Flash, 30

FMVC (фрактальный MVC), паттерн, 142

for, цикл, 56, 456

fs, модуль, 335

G

get_by_cid(), метод, 205

get_chatee(), метод, 238

get_cid_map(), метод, 214

get_db(), метод, 205, 214

getEmSize, служебная функция, 198

get_is_anon(), метод, 204

get_is_user(), метод, 204

getPeopleList(), метод, 214

get_user(), метод, 205, 218

GET, глагол, 312

get, метод, Express, 307

Google

- индексирование SPA, 396
- кнопка \, 139

Google Analytics

- на стороне сервера, 402
- события, 401

Googlebot

- определение, 396
- строка пользовательского агента, 398
- тестирование, 399

Google Chat, 333

Google Chrome

- и быстроедействие JavaScript, 32
- инструменты разработчика
 - изучение приложения, 46
- обзор, 38
- метод Object.create(), 72

- протоколирование

- AJAX-запросов, 399

Google V8, движок JavaScript, 35

GWT (Google Web Toolkit), 33

H

Handlebars, система шаблонов, 271

handleResize, метод, 183

hashchange, обработчик события, 129, 135, 154, 170, 180

headers, свойство, 301

head, элемент, 101

height, свойство, 48

HTML, 31

- преобразование в JavaScript-код, 110
- шаблон функциональных контейнеров, 111

HTML5, 32

HTTPS, 329

HTTP-кэширование
 last-modified, атрибут, 413
 max-age, атрибут, 411
 no-cache, атрибут, 412
 no-store, атрибут, 413
 http, модуль Node.js, 299

I

id, свойство объекта person, 203
 if, предложение, 47, 457
 initModule, функция, 44, 103, 112, 147, 154, 171
 Internet Explorer
 инструменты разработчика, 38
 метод Object.create(), 72
 is_chatee_online, флаг, 249
 isFakeData, флаг, 212

J

Jabber, 333
 jasmine-jquery, каркас, 473
 JavaScript
 Node.js, 297
 аргументы, 155
 быстродействие, 32
 два прохода, 60
 ипотечный калькулятор, 29
 использование на всех уровнях
 SPA, 28
 контекст выполнения, 62
 обратные вызовы, 155
 переменные
 область видимости, 55, 66
 поднятие, 58
 преобразование HTML в, 110
 прототипическое
 наследование, 69
 сгенерированный, 33
 функции. См. функции
 цепочка прототипов, 73
 мутации, 77
 эволюция, 31
 Java-апплеты, 29
 join(), метод, 238, 246
 jQuery, 40
 анимация, 45
 кросс-браузерная
 совместимость, 32

манипуляция объектами DOM,
 35, 454
 переменная \$, 82
 преимущества, 45
 скачивание, 99
 унифицированная библиотека
 ввода, 202

jquery.event.gevent-0.1.9.js, файл, 196
 jquery.event.ue-0.3.2.js, файл, 196
 JSLint, 40, 44
 использование, 462
 настройка, 461
 установка, 460
 JSON, 31, 35
 просмотр в браузере, 354
 схемы
 загрузка, 362
 создание, 360
 JSONovitch, расширение Firefox, 354
 JSONView, расширение Chrome, 354
 JSV, модуль, установка, 360

K

K&R, стиль расстановки скобок, 430

L

last-modified, атрибут, 413
 leavechat, сообщение, 252
 let, предложение, 56
 listchange, сообщение, 244
 listen, метод, 300
 LiveFyre, 139, 144
 logger(), функция, 304
 login(), метод, 206, 218
 logout(), метод, 206, 218
 log(), функция, 59
 LRU, алгоритм вытеснения
 из кэша, 414

M

makeCid(), метод, 219
 makeError(), метод, 148
 makePerson(), метод, 213
 max-age, атрибут, 411
 memcached, 340
 и Node.js, 415
 использование для кэширования
 на сервере, 414

methodOverride(), метод, 309
method, свойство, 302
MicroMVC, 141
mocha, каркас, 473
Model, модуль
 включение унифицированной
 библиотеки ввода, 202
 добавление поддержки аватаров
 изменения в объекте chat, 256
 использование модуля Fake, 258
 исправление ошибок,
 найденных во время
 тестирования, 260
 тестирование, 259
 обзор, 190
 объект chat. См. объект chat
 объект people. См. people, объект
 модели
 содержимое файлов, 196
 структура каталогов и файлов, 194
mod_perl, 297
MongoDB, 32
 валидация данных. См. Валидация
 данных
 динамическая структура
 документа, 343
 документоориентированное
 хранилище, 342
 драйвер. См. Драйвер MongoDB
 команды, 346
 кэширование, 421
 модуль CRUD. См. CRUD,
 модуль базы данных
 преимущества, 340
Mongolia, 348
Mongoose, 348
Mongoskin, 348
mongo, команда, 350
Mustache, система шаблонов, 271
MySQL, 340

N

name, свойство объекта person, 203
Neo4J, 340
New Relic, 404
new, ключевое слово, 71, 448, 452
no-cache, атрибут, 412
nodealitics, проект, 403

node-googleanalytics, проект, 403
Node.js
 движок JavaScript, 35
 загрузка модулей, 323, 475
 использование Connect, 302
 использование Express. См. Express,
 каркас для Node.js
 использование Redis, 415
 общие сведения, 297
 преимущества, 297
 приложение Hello World, 298
 скачивание, 298
NODE_PATH, переменная
 окружения, 474
nodeunit, каркас
 настройка, 473
 определение, 473
NoSQL, базы данных, 340
no-store, атрибут, 413
npm (Node Package Manager)
 использование манифеста, 305
 определение, 298
 флаг -save, 306

O

Object.create, метод, 71, 215
ODM (объектно-документное
отображение), 348
onClick, обработчик события, 170
odata, свойство, 301
onerror, событие, 404
onHeldendNav, обработчик
события, 281
onListchange, обработчик
события, 264, 269, 282
onload, метод, 68
onload, событие, 45
onLogin, обработчик
события, 231, 264, 270
onLogout, обработчик
события, 231, 264, 270, 282
onresize, метод, 68
onSetchatee, обработчик
события, 264, 268, 281
onSubmitMsg, обработчик
события, 263, 268
onTapAcct, обработчик события, 230
onTapList, обработчик
события, 263, 268

onTapNav, обработчик события, 280
 onTapToggle, обработчик события, 268
 onUpdatechat, обработчик события, 264, 269
 on, метод, 333
 Openfire, 333
 Opera и быстродействие JavaScript, 32
 opt_label, параметр, 402
 opt_noninteraction, параметр, 402
 opt_value, параметр, 402
 ORM (объектно-реляционное отображение), 348
 Overture, 139

P

Passenger, 297
 PATCH, глагол, 312
 patr, каркас, 473
 people_cid_map, ключ, 215
 people_db, ключ, 215
 people, объект модели
 документирование, 209
 изменение модели, 218
 методы, 205
 обзор, 202
 подставной объект соединения
 Socket.IO, 223
 подставной список людей, 212
 реализация, 213
 события, 206
 тестирование, 225
 PhoneGap, 32
 POST, глагол, 312
 post, метод, Express, 313
 preventDefault(), метод, 122
 preventImmediatePropagation(), метод, 122
 PUT, глагол, 312

R

Rackspace, 406
 Redis, 414, 415
 removePerson, метод, 219
 removeSlider(), метод, 182
 remove, метод, MongoDB, 346
 request, объект, Node.js, 300, 301

resize, событие, 185
 response, объект, Node.js, 300
 REST (передача представимых состояний), 312
 return, предложение, 458
 Ruby on Rails, 141

S

Safari
 инструменты разработчика, 38
 метод Object.create(), 72
 safe, параметр, 355
 script, элемент
 местоположение на странице, 101
 общие сведения, 37
 scrollChat, метод, 263
 send_listchange, функция, 244
 send_message(), метод, 238
 set_chatee(), метод, 238
 setDataMode(), метод, 471
 setInterval, функция, 331
 setSliderPosition(), метод, 157
 ShareThis, 139
 Shell, модуль
 HTML-файл приложения, 100
 аутентификация и завершение сеанса
 модификация
 JavaScript-кода, 229
 модификация таблицы стилей, 231
 проектирование пользовательского интерфейса, 229
 тестирование, 233
 координация коммуникаций между функциональными модулями, 153
 модификация для поддержки модуля Avatar, 285
 модификация для поддержки функциональных модулей, 172
 общие сведения, 96
 пространства имен.
 См. Пространства имен
 структура каталогов и файлов, 98
 управление состоянием приложения
 использование якоря, 126

- общие сведения, 124
- элементы управления
 - в настольных
 - и веб-приложениях, 124
 - элементы управления историей, 125
- файлы jQuery, 99
- функциональные контейнеры. *См.* Функциональные контейнеры
- show dbs, команда, 346
- socket.io.js, файл, 332
- Socket.IO, модуль, 35
 - и сервер обмена сообщениями, 333
 - общие сведения, 329
 - отправка сообщения
 - об обновлении приложения, 334
 - подставной объект, 223
- spa.avtr.css, файл, 196, 284
- spa.avtr.js, файл, 196, 278
- spa.chat.css, файл, 273
- spa.chat.js, файл, 264
- spa.css, файл, 272
- spa.data.js, файл, 196, 290
- spa.fake.js, файл, 196, 212, 224, 244, 253, 258
- spa.js, файл, 216, 289
- spa-listchange, событие, 206, 239
- spa-login, событие, 206
- spa-logout, событие, 206
- spa.model.js, файл, 214, 219, 239, 248, 257, 260, 291
- spa-setchatee, событие, 238, 239
- spa.shell.css, файл, 232
- spa.shell.js, файл, 230, 285
- spa-slider, класс, 39
- spa-updatechat, событие, 239
- spa.util_b.js, файл, 199
- spa, каталог, 98
- SPA (одностраничное приложение)
 - история, 29
 - и эволюция JavaScript, 30
 - преимущества, 49
- Spring MVC, 141
- SQL, 31
- static, метод, 322
- stopPropagation(), метод, 122
- strict, прагма, 213
- style, элемент, 37

- Sublime, 432
- SVG, 32
- switch, предложение, 458

T

- TaffyDB, 214
- taffydb-2.6.2.js, файл, 196
- this, ключевое слово, 87
- Tidy, программа, 106
- TODO, комментарии, 436
- Tomcat, 297
- Tornado, 297
- try/catch, блок, 404, 458
- Twisted, 297
- type, атрибут, 56

U

- Uglify, 454
- undefined, значение, 58, 61, 73, 449
- underscore.js, 271
- update_avatar(), метод, 238, 256
- updateavatar, обработчик сообщения, 257, 258, 390
- updatechat, обработчик сообщения, 252, 386
- update, метод, MongoDB, 346
- uriAnchor, подключаемый модуль, 99, 127, 133
- url, свойство, 302
- userupdate, сообщение, 219
- use, команда, 346

V

- ValueClick, 139
- var, ключевое слово, 54, 56, 58, 323, 449
- vim, редактор, 111, 432
- vows, каркас, 473

W

- watchFile, метод, 338
- Webmaster Tools, 399
- WebStorm, 432
- wget, программа, 314
- while, предложение, 459
- window объект
 - и глобальные переменные, 68
 - событие onload, 45, 68

событие `resize`, 185
 with, предложение, 459
`writeAlert`, метод, 263
`writeChat`, метод, 263

Х

XML, 31
 XMPP (Extensible Messaging and Presence Protocol), 333

Y

Yahoo, аутентификация, 295

Z

zombie, каркас, 473

А

Абзацы, разбиение кода на, 427
 Автоматизированное тестирование, 193
 Автоматическая двусторонняя привязка к данным, 287
 Автономное тестирование, 193
 Авторизация, 295
 Анимация с помощью jQuery, 45
 Анонимные функции
 локальные переменные, 80
 общие сведения, 78
 определение, 79
 самовыполняющиеся, 79
 Аргументы в JavaScript, 155
 Асинхронные каналы, 237
 Аутентификация, 294

Б

Базовая аутентификация на сервере, 327
 Базы данных
 MongoDB. См. MongoDB
 и бизнес-логика, 341
 исключение преобразования данных, 340
 кэширование запросов, 420
 Библиотеки, загрузка в последнюю очередь, 150
 Бизнес-логика, 190
 Большой вектор атаки, 329

Булевы переменные, именование, 441

В

Валидация данных
 загрузка JSON-схем, 362
 общие сведения, 364
 создание JSON-схемы, 360
 тип объекта, 357
 установка модуля JSV, 360
 функция валидации, 363
 Веб-сокеты
 общие сведения, 329
 преимущества, 381
 Веб-хранилище, 408
 Верблюжья нотация, 440
 Внедряемая система шаблонов, 271
 Выплывающий чат, пример
 CSS, 39
 HTML, 39
 JavaScript-код, 40
 добавление обработчика щелчка мышью, 119
 изучение с помощью инструментов разработчика в Chrome, 46
 метод сворачивания и раскрытия окна, 117
 постановка задачи, 36
 структура файла, 37
 Выражения присваивания, 459

Г

Глобальная область видимости, 66
 Глобальное пространство имен, засорение, 81
 Глобальные переменные
 и объект `window`, 68
 общие сведения, 55

Д

Динамический язык, 54
 Документирование
 в виде комментариев, 436
 объект `chat`, 239
 объект `people`, 209
 Документоориентированное хранилище, 342

Долгое нажатие, жест, 202

Драйвер MongoDB

добавление в серверное

приложение, 353

использование методов CRUD, 350

подготовка файлов проекта, 347

установка, 348

Е

Естественные JavaScript SPA, 33

З

Зависимые пары ключ–значение, 133

Загрузка

библиотек, 150

модулей в Node.js, 323

Закладки в браузере, 124

Замыкания в JavaScript, 85

Запросы, кэширование, 420

Запятая, оператор, 58, 459

Засорение глобального пространства имен, 81

И

Именованые переменных

булевых, 441

информация об области

видимости, 440

использование общепринятых

символов, 439

и тип переменной, 440

массивов, 443

неизвестного типа, 445

обзор, 437

объектов, 444

регулярных выражений, 443

строковых, 442

функций, 445

хэшей, 444

целых, 442

числовых, 443

Именованные аргументы, 450

Инициализация функциональных модулей, 156

Инструментальная система шаблонов, 271

Инструменты разработчика в Chrome, 399

Интерполяция внутри кавычек, 433

История

развития SPA, 29

управление состоянием

приложения, 125

К

Каррирование, 438

Каскадная инициализация

функционального модуля Chat, 158

Клиентские идентификаторы, 204

Код

единообразное использование

кавычек, 433

единообразное разбиение

строк, 429

единообразные отступы, 426

использование разрядки, 432

разбиение на абзацы, 427

расстановка скобок в стиле

K&R, 430

сопровождение, 141

стандарты, 54

Команды MongoDB, 346

Комментарии

документирование API, 436

пояснение важных мест

программы, 434

Контейнеры, отрисовка в, 139

Контекст выполнения

в JavaScript, 62

объект, 62

цепочка ссылок, 90

Контроллер (паттерн MVC), 142

Конфигурирование модулей, 155

Конфиденциальность и сторонние модули, 140

Крокфорд Дуглас, 460

Кросс-платформенная разработка, 32, 50

Кэширование

HTTP

last-modified, 413

max-age, 411

no-cache, 412

no-store, 413

в MongoDB, 421

веб-хранилище, 408

запросов к базе данных, 420
на сервере, 414
обзор, 406

Л

Литеральные объекты, 55
Локальные переменные
в анонимных функциях, 80
общие сведения, 55

М

Маршруты обобщенные, 320
Массивы, именование, 443
Метки, 456
Модель (паттерн MVC), 142
Модули
адаптация для тестирования, 496
загрузка в Node.js, 323, 475
шаблон, 463

Н

Назад, кнопка, 124
Независимые пары
ключ–значение, 133
Неизвестные типы, именование, 445

О

Область видимости
в новых версиях JavaScript, 56
информация в имени
переменной, 440
переменных, 55, 66
функции, 62
Область видимости блока, 56
Обновление приложений, 334
Обработчики событий, 464
в примере чата, 119
именование, 264
Обработчик событий щелчка
мышью, 119
Обратные вызовы, 155, 464
Объекты
именование, 444
функции как, 78
Объекты на основе классов
и прототипов, сравнение, 69
Объявление переменных, 54, 448
Открытые методы, 464

Отступы в коде, 426
Очередь событий в Node.js, 298

П

Память
управление, 85
утечки, 90
Паттерн модуля
в JavaScript, 82
определение, 80
Передача по ссылке, 155
Переменные
undefined, 61
глобальные и объект window, 68
именование. См. Именование
переменных
не более одного присваивания
в строке, 450
область видимости, 55, 66
объявление, 54, 448
поднятие, 58
присваивание функций, 450
Переходная кривая, 45
Планшеты
и SPA, 51
и единица измерения px, 162
и унифицированная библиотека
ввода, 202
Повторное использование кода, 141
Подключаемые модули, 30
Поднятие переменных, 58
Подписка на событие, 206
Поисковая оптимизация, 396
Правило ограничения домена, 30
Префиксы имен классов CSS, 161
Привязка к данным, 287
Проблема остановки, 496
Промежуточного уровня
функциональность, 302
Пространства имен
в модуле Shell
корневое пространство имен
CSS, 101
корневое пространство имен
JavaScript, 103
обзор, 98
имена и структура дерева
файлов, 454

- использование префикса, 161
- стандарты кодирования, 453
- Протоколирование ошибок
 - на стороне клиента
 - вручную, 404
 - с помощью сторонних служб, 403
- Прототипическое наследование, 69

Р

- Разбиение строк, 429
- Развертывание, 31, 50
- Разработка через тестирование (TDD), 260
- Разрядка, использование в коде, 432
- Регрессионное тестирование, 193
- Регулярные выражения, именование, 443
- Реляционные базы данных, 340
- Роботы-обходчики, 396

С

- Сборщик мусора, 85
- Сгенерированный JavaScript-код, 33
- Серверы
 - CRUD, маршруты
 - обновление пользователя, 316
 - обобщенные, 319
 - перенос в отдельный модуль, 324
 - получение списка пользователей, 313
 - создание объекта пользователя, 313
 - удаление пользователя, 316
 - чтение объекта пользователя, 315
 - Node.js. См. Node.js
 - Socket.IO. См. Socket.IO
 - веб-сокеты, 329
 - аутентификация и авторизация, 294
 - базовая аутентификация, 327
 - валидация, 295
 - кэширование, 414
 - сохранение и синхронизация данных, 296
- Синхронные каналы, 237
- Слаботипизированный язык, 54

- Служебные методы, 464
- События
 - в Google Analytics, 401
 - и обратные вызовы, 207
 - обработка в jQuery, 207
 - очередь в Node.js, 298
- Соглашения при кодировании, 36
- Сопровождение кода, 141
- Состояние приложения
 - определение, 124
 - управление с помощью якоря `urlAnchor`, подключаемый модуль, 133
 - модификация Shell, 129
 - общие сведения, 126, 134
 - элементы управления в браузере и персональном приложении, 124
 - элементы управления историей, 125
- Среда выполнения, 31
- Статические файлы, обслуживание с помощью Express, 310
- Сторонние библиотеки
 - вопросы безопасности, 30
 - загрузка SPA в последнюю очередь, 150
 - и самовыполняющиеся анонимные функции, 81
 - недостатки, 139
 - примеры, 139
 - сравнение с функциональными модулями, 139
- Схемы и MongoDB, 344

Т

- Текстовые редакторы, 111
- Тестирование
 - адаптация модулей для, 496
 - выбор каркаса, 472
 - настройка nodeunit, 473
 - режимы тестирования, 468
 - создание комплекта тестов
 - загрузка модулей с помощью Node.js, 475
 - план событий и тестов, 480
 - подготовка теста в nodeunit, 478
 - создание первого теста, 479

тестирование обмена
сообщениями, 488
тесты аутентификации, 483
тесты завершения сеанса, 492
Типы и имена переменных, 440

У

Унифицированная библиотека
ввода, 202
Установка
 JSLint, 460
 драйвера MongoDB, 348

Ф

Фабрика, паттерн, 451
Фрактал, определение, 142
Функции
 анонимные, 78
 замыкания, 85
 именование, 445
 контекст выполнения, 64
 немедленный вызов, 452
 область видимости, 62
 паттерн модуля, 82
 полноправные объекты, 78
 присваивание переменным, 450
 самовыполняющиеся
 анонимные, 79
 стандарты кодирования, 450
Функциональные контейнеры
 CSS-стили модуля Shell, 106, 113
 HTML-код модуля Shell, 105
 добавление HTML-шаблона, 111
 настройка приложения
 для использования Shell, 115
 общие сведения, 104
 определение, 116
Функциональные модули
 Chat. См. Chat, функциональный
 модуль

инициализация, 156
и паттерн MVC, 141
коммуникация, 153
 сравнение со сторонними
 модулями, 139
общие сведения, 138
определение, 137

Х

Хранилища ключей
и значений, 340, 408
Хэши, именование, 444

Ц

Целые переменные,
именование, 442
Цепочка прототипов, 73
 мутации, 77

Ш

Шаблон модуля, 463

Э

Эволюция JavaScript, 30
Эластичная верстка, 106

Я

Якорного интерфейса паттерн, 126
Якорь
 в функциональном модуле
 Chat, 155
 управление состоянием
 приложения
 uriAnchor, подключаемый
 модуль, 133
 модификация Shell, 129
 общие сведения, 126, 134
 перезагрузка страницы, 127

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЪЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Майкл С. Миковски, Джош К. Пауэлл

Разработка одностраничных веб-приложений

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Слинкин А. А.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 13.12.2013. Формат 60×90 1/16 .

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 32. Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru

Если ваш сайт представляет собой набор дергающихся страниц, связанных ссылками, то вы отстали от жизни. Следующей ступенью вашей карьеры должны стать одностраничные приложения (SPA). В таком приложении отрисовка пользовательского интерфейса и бизнес-логика перенесены в браузер, а взаимодействие с сервером сводится к синхронизации данных. Пользователь работает с таким сайтом, как с персональным приложением на рабочем столе, что гораздо удобнее и приятнее. Однако разрабатывать, сопровождать и тестировать SPA нелегко.

В этой книге показано как организуется командная разработка передовых SPA — проектирование, тестирование, сопровождение и развитие — с применением JavaScript на всех уровнях и без привязки к какому-то конкретному каркасу. Попутно вы отточите навыки работы с HTML5, CSS3 и JavaScript и узнаете об использовании JavaScript не только в браузере, но также на сервере и в базе данных.

Предполагается, что читатель знаком с основами веб-разработки. Опыта разработки SPA не требуется.

Краткое содержание:

- Проектирование, реализация и тестирование всех компонентов SPA;
- Лучшие в своем классе инструменты типа jQuery, TaffyDB, Node.js и MongoDB;
- Построение веб-приложений, работающих в реальном масштабе времени, с помощью веб-сокетов и библиотеки Socket.IO;
- Адаптация к сенсорным устройствам: планшета и смартфонам;
- Типичные ошибки при проектировании одностраничных приложений.

Авторы являются архитекторами и техническими руководителями.

Майкл Миковски работал над многими коммерческими SPA и платформой, обрабатывающей свыше 100 миллиардов запросов в код.

Джош Пауэлл принимал участие в создании ряда сайтов из числа самых нагруженных в веб.

Разработка одностраничных веб-приложений

Опыт, отточенный в ходе разработки многих поколений SPA.

Из предисловия
Грегори Д. Бенсона

Основательно, полно и методично.

Марк Райалл,
ThoughtWorks

Читать обязательно, даже если пользуетесь каким-то каркасом.

Кен Римпл,
автор «Spring Roo in Action»

Горячо рекомендую изложенные в этой книге приемы.

Джейсон Качор,
SharePoint MVP

Блистательное руководство!
Майк ГринХалл,
NHS Wales

Интернет-магазин: www.dmkpress.com

Книга — почтой: orders@aliants-kniga.ru

Оптовая продажа: "Альянс-книга"

тел. (499) 725-54-09

books@aliants-kniga.ru



ISBN 978-5-97060-072-6



9 785970 600726 >