# Разработка веб-приложений с использованием AngularJS

Создавайте одностраничные веб-приложения, реализуя всю мощь AngularJS

Павел Козловский
Питер Бэкон Дарвин

ДМК ИЗДАТЕЛЬСТВО

Павел Козловский
Питер Бэкон Дарвин

# Разработка веб-приложений с использованием AngularJS

---

Создавайте одностраничные веб-приложения, реализуя всю мощь AngularJS

# Mastering Web Application Development with AngularJS

Build single-page web applications using the power of AngularJS

**Pawel Kozlowski**
**Peter Bacon Darwin**

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

# Разработка веб-приложений с использованием AngularJS

Создавайте одностраничные веб-приложения, реализуя всю мощь AngularJS

**Павел Козловский**
**Питер Бэкон Дарвин**

ДМК Пресс
издательство
Москва, 2014

**УДК 004.738.5:004.4AngularJS**
**ББК 32.973.26-018.2**
**К59**

К59 Павел Козловский, Питер Бэкон Дарвин

Разработка веб-приложений с использованием AngularJS. – Пер. с англ. Киселева А. Н., М.: ДМК Пресс, 2014. – 394с.: ил.

**ISBN 978-5-97060-064-1**

С появлением HTML5 и CSS3 разработка клиентских веб-приложений на языке JavaScript приобрела особую популярность. Создатели фреймворка AngularJS предприняли революционный подход к решению вопроса превращения браузера в самую лучшую платформу для разработки веб-приложений.

Книга проведет вас через основные этапы конструирования типичного одностраничного веб-приложения. В ней обсуждаются такие темы, как организация структуры приложения, взаимодействие с различными серверными технологиями, безопасность, производительность и развертывание. После представления AngularJS и обзора перспектив клиентских веб-приложений, книга шаг за шагом проведет вас через создание достаточно сложного приложения.

Издание будет наиболее полезно веб-разработчикам, желающим оценить или решившим применить фреймворк AngularJS для создания своих приложений. Предполагается, что читатель имеет некоторое знакомство с AngularJS, хотя бы на уровне понимания простейших примеров. Мы надеемся также, что вы обладаете знанием HTML, CSS и JavaScript.



ISBN 978-1-78216-182-0 (англ.)
ISBN 978-5-97060-064-1 (рус.)

# ОГЛАВЛЕНИЕ

**Об авторах ..... 11**
**Благодарности ..... 12**
**О рецензентах ..... 14**
**Предисловие ..... 15**
О чем рассказывается в этой книге ..... 16
Что потребуется при чтении этой книги ..... 17
Кому адресована эта книга ..... 18
Соглашения ..... 18
Отзывы и пожелания ..... 19
Поддержка клиентов ..... 19
Загружаемые примеры программного кода ..... 19
Ошибки и опечатки ..... 20
Нарушение авторских прав ..... 20
Вопросы ..... 20

**Глава 1. Дзен Angular ..... 21**
Знакомьтесь, AngularJS ..... 22
Общие сведения о фреймворке ..... 22
Найдите свой путь в проект ..... 23
Сообщество ..... 23
Обучающие ресурсы в Интернете ..... 23
Библиотеки и расширения ..... 24
Инструменты ..... 24
Batarang ..... 24
Plunker и jsFiddle ..... 25
Расширения и дополнения для IDE ..... 25
Ускоренное введение в AngularJS ..... 25
Hello World – пример приложения на AngularJS ..... 25
Двунаправленное связывание данных ..... 27
Шаблон MVC в AngularJS ..... 27
С высоты птичьего полета ..... 28
Подробнее о контекстах ..... 30
Представление ..... 37
Модули и внедрение зависимостей ..... 42
AngularJS и остальной мир ..... 56
jQuery и AngularJS ..... 58
Взгляд в будущее ..... 59
В заключение ..... 60

**Глава 2. Сборка и тестирование ..... 62**
Введение в пример приложения ..... 63
Область применения ..... 63
Стек технологий ..... 65
Хранилище данных ..... 65
Система сборки ..... 68
Принципы построения систем сборки ..... 69
Инструменты ..... 71
Организация файлов и каталогов ..... 73
Каталоги верхнего уровня ..... 74
Каталог с исходным кодом ..... 75
Соглашения по именованию файлов ..... 79
Модули и файлы AngularJS ..... 80
Один файл, один модуль ..... 80
Внутри модуля ..... 81
Автоматическое тестирование ..... 84
Модульные тесты ..... 86
Интеграционные тесты ..... 93
В заключение ..... 97

**Глава 3. Взаимодействие с сервером ..... 99**
Выполнение запросов XHR и JSONP с помощью $http ..... 99
Модель данных и адреса URL в MongoLab ..... 100
Краткий обзор $http ..... 100
Ограничения политики общего происхождения ..... 104
Promise API и служба $q ..... 109
Получение отложенных результатов с помощью службы $q ..... 110
Интеграция службы $q в AngularJS ..... 119
Promise API и служба $http ..... 120
Взаимодействие с конечными точками RESTful ..... 121
Служба $resource ..... 122
Взаимодействия с веб-службами REST с помощью $http ..... 128
Дополнительные возможности $http ..... 132
Обработка ответов ..... 132
Тестирование кода, осуществляющего взаимодействия с помощью $http ..... 133
В заключение ..... 136

**Глава 4. Отображение и форматирование данных ... 137**
Знакомство с директивами ..... 137
Отображение результатов вычисления выражений ..... 138
Директива интерполяции ..... 138
Отображение значений с помощью ngBind ..... 139
Включение разметки HTML в выражения ..... 139
Отображение по условию ..... 141
Включение блоков содержимого по условию ..... 142
Отображение коллекций с помощью директивы ngRepeat ..... 143
Знакомство с директивой ngRepeat ..... 143
Специальные переменные ..... 144

Итерации по свойствам объекта ... 145
Приемы использования директивы ngRepeat ... 145
Обработчики событий DOM ... 149
Увеличение эффективности с помощью шаблонов на основе DOM ... 150
Избыточный синтаксис ... 150
Применение директивы ngRepeat к множеству элементов DOM ... 151
Элементы и атрибуты не могут изменяться во время выполнения ... 152
Нестандартные элементы HTML и старые версии IE ... 153
Преобразование моделей с помощью фильтров ... 153
Применение встроенных фильтров ... 154
Создание собственных фильтров – реализация постраничного вывода ... 161
Доступ к фильтрам из кода на JavaScript ... 163
Правила использования фильтров ... 164
В заключение ... 168

**Глава 5. Создание улучшенных форм ... 169**
Сравнение традиционных форм с формами AngularJS ... 169
Введение в директиву ngModel ... 171
Создание формы с информацией о пользователе ... 172
Директивы ввода ... 173
Добавление проверки обязательного наличия значения ... 174
Текстовые элементы ввода ... 174
Кнопки-флажки ... 175
Радиокнопки ... 175
Элементы выбора из списка ... 176
Использование скрытых полей ввода ... 181
Устройство механизма связывания данных в ngModel ... 182
ngModelController ... 182
Проверка форм в AngularJS ... 184
ngFormController ... 184
Добавление динамического поведения в форму с информацией о пользователе ... 185
Вывод сообщений об ошибках ... 186
Отключение процедуры проверки, встроенной в браузер ... 188
Вложенные формы ... 188
Вложенные формы как компоненты многократного пользования ... 189
Повторение вложенных форм ... 189
Проверка повторяющихся полей ввода ... 191
Отправка традиционной формы HTML ... 192
Непосредственная отправка форм на сервер ... 192
Обработка события отправки формы ... 193
Сброс формы в исходное состояние ... 194
В заключение ... 195

**Глава 6. Организация навигации ... 196**
Адреса URL в одностраничных веб-приложениях ... 197

Адреса URL с решеткой до появления HTML5 ..... 198
HTML5 и интерфейс истории посещений ..... 199
Служба $location ..... 200
Знакомство с интерфейсом службы $location и адресами URL ..... 201
Адреса фрагментов, навигация внутри страницы и $anchorScroll ..... 202
Настройка режима HTML5 интерпретации адресов URL ..... 203
Навигация вручную с помощью службы $location ..... 205
Служба $route ..... 208
Определение основных маршрутов ..... 208
Гибкое сопоставление маршрутов ..... 210
Повторное использование шаблонов разметки с разными контроллерами ..... 212
Предотвращение «мерцания» пользовательского интерфейса при изменении маршрута ..... 213
Предотвращение изменения маршрута ..... 215
Ограничения службы $route ..... 216
Один маршрут соответствует одной области на экране ..... 217
Распространенные приемы использования, советы и рекомендации ..... 220
Обработка ссылок ..... 220
Организация определений маршрутов ..... 222
В заключение ..... 224

**Глава 7. Безопасность приложений ..... 226**

Аутентификация и авторизация на стороне сервера ..... 227
Обработка неавторизованного доступа ..... 227
Реализация прикладного интерфейса аутентификации на стороне сервера ..... 228
Безопасность шаблонов разметки ..... 228
Противостояние нападениям ..... 229
Предотвращение перехвата cookie («атака через посредника») ..... 230
Предотвращение нападений вида «межсайтовый скриптинг» ..... 231
Предотвращение внедрения данных в формате JSON ..... 233
Предотвращение подделки межсайтовых запросов ..... 234
Обеспечение безопасности на стороне клиента ..... 235
Служба security ..... 236
Отображение формы аутентификации ..... 236
Создание меню и панелей инструментов с поддержкой системы безопасности ..... 238
Поддержка аутентификации и авторизации на стороне клиента ..... 240
Обработка ошибок авторизации ..... 241
Перехват ответов ..... 242
Создание службы securityInterceptor ..... 242
Создание службы securityRetryQueue ..... 244
Предотвращение переходов по защищенным маршрутам ..... 246
Использование функций в свойстве resolve маршрутов ..... 247
Создание службы authorization ..... 248
В заключение ..... 250

**Глава 8. Создание собственных директив ..... 251**
Что такое директива AngularJS? ..... 251
Встроенные директивы ..... 252
Использование директив в разметке HTML ..... 253
Тестирование директив ..... 255
Определение директивы ..... 257
Оформление кнопок с помощью директив ..... 258
Создание директивы button ..... 258
Директивы-виджеты ..... 261
Создание директивы постраничного просмотра ..... 261
Тест для директивы постраничного просмотра списков ..... 262
Использование шаблонов с разметкой HTML в директивах ..... 263
Изолирование директивы от родительского контекста ..... 264
Реализация виджета ..... 267
Добавление в директиву функции обратного вызова selectPage ..... 268
Создание директивы проверки ..... 269
Внедрение контроллера другой директивы ..... 270
Взаимодействие с контроллером ngModelController ..... 271
Тестирование директивы проверки ..... 272
Реализация директивы проверки ..... 274
Асинхронная проверка модели ..... 275
Имитация службы Users ..... 275
Тестирование директивы асинхронной проверки ..... 276
Реализация директивы асинхронной проверки ..... 278
Директива-обертка для виджета выбора даты из библиотеки jQueryUI ..... 279
Тестирование директив-оберток ..... 281
Реализация директивы datepicker ..... 282
В заключение ..... 283

**Глава 9. Создание продвинутых директив ..... 285**
Включение ..... 285
Использование включения в директивах ..... 286
Включение в директивах с изолированным контекстом ..... 286
Директива вывода предупреждения на основе приема включения ..... 286
Контекст включения ..... 288
Создание и использование функций включения ..... 291
Создание функции включения с помощью службы $compile ..... 291
Использование функций включения в директивах ..... 292
Создание директивы if, использующей включение ..... 294
Контроллеры директив ..... 296
Внедрение специальных зависимостей в контроллеры директив ..... 297
Создание директивы постраничного просмотра на основе контроллера ..... 298
Различия между контроллерами директив и функциями связывания ..... 299
Комплект директив виджета «аккордеон» ..... 301
Управление процессом компиляции ..... 305
Создание директивы field ..... 306

Использование службы $interpolate ..... 308
Динамическая загрузка шаблонов ..... 310
Настройка шаблона директивы field ..... 310
В заключение ..... 312

**Глава 10. Создание интернациональных веб-приложений ..... 313**
Использование национальных наборов символов и настроек ..... 314
Модули с национальными настройками ..... 314
Использование доступных национальных настроек ..... 315
Поддержка переводов ..... 317
Перевод строк в шаблонах AngularJS ..... 318
Перевод строк в коде JavaScript ..... 321
Шаблоны проектирования, советы и рекомендации ..... 322
Инициализация приложений с учетом выбранных национальных настроек ..... 322
Переключение между национальными настройками ..... 325
Нестандартное форматирование дат, чисел и валют ..... 326
В заключение ..... 328

**Глава 11. Создание надежных веб-приложений на основе AngularJS ..... 330**
Внутренние механизмы AngularJS ..... 331
Это не механизм строковых шаблонов ..... 331
Настройка производительности – определить требования, измерить, настроить и повторить ..... 343
Настройка производительности приложений на основе AngularJS ..... 345
Оптимизация использования процессора ..... 346
Оптимизация потреблении памяти ..... 356
Директива ng-repeat ..... 358
В заключение ..... 360

**Глава 12. Подготовка и развертывание веб-приложений на основе AngularJS ..... 362**
Повышение производительности сетевых операций ..... 363
Минификация статических ресурсов ..... 363
Предварительная загрузка шаблонов ..... 368
Оптимизация начальной страницы ..... 373
Избегайте отображения шаблонов в необработанном виде ..... 373
AngularJS и подключение прикладных сценариев ..... 376
Поддержка браузеров ..... 379
Поддержка Internet Explorer ..... 380
В заключение ..... 381

**Предметный указатель ..... 383**

# ОБ АВТОРАХ

**Павел Козловский** (Pawel Kozlowski) обладает более чем 15-летним профессиональным опытом веб-разработки и использования самых разных веб-технологий, языков и платформ. Он одинаково хорошо разбирается с особенностями разработки как клиентских, так и серверных компонентов веб-приложений и всегда старается использовать самые производительные инструменты и приемы.

Павел является убежденным сторонником свободного, открытого программного обеспечения. Он с большим энтузиазмом участвует в работе над проектом AngularJS и ведет активную деятельность в сообществе пользователей AngularJS. Он также является одним из разработчиков Angular UI – комплекта компонентов, сопутствующего фреймворку AngularJS, где занимается разработкой директив Twitter Bootstrap для AngularJS.

В свободное от программирования время Павел занимается популяризацией AngularJS на различных конференциях и встречах.

**Питер Бэкон Дарвин** (Peter Bacon Darwin) занимается программированием уже больше двух десятилетий. Ему довелось работать с фреймворком .NET еще до того, как он был выпущен; Питер принимал участие в разработке IronRuby[1] и работал консультантом по информационным технологиям в Avanade и IMGROUP, пока не оставил их, чтобы присматривать за своими детьми и заниматься независимой разработкой.

Питер является заметной фигурой в сообществе AngularJS. Он недавно присоединился к команде разработки AngularJS в Google, как внештатный разработчик, и является одним из основателей проекта AngularUI. Он часто выступает с докладами о AngularJS на конференциях Devoxx UK и многочисленных встречах в Лондоне. Он так же ведет учебные курсы в AngularJS. Как консультант, он в первую очередь стремится помочь компаниям наиболее оптимально использовать фреймворк AngularJS.

1 Реализация языка Ruby для .NET. – *Прим. перев.*

# БЛАГОДАРНОСТИ


### *От Павла Козловского*

Даже не верится, что на протяжении последних месяцев, пока я работал над книгой, мне повезло сотрудничать с такими замечательными людьми. Эта книга едва ли появилась бы на свет без помощи и напряженного труда всех вас. Спасибо вам!

В первую очередь я хотел бы сказать «Спасибо!» всем участникам проекта AngularJS в компании Google. Вы – команда мечты, работающая над развитием удивительным фреймворка. Не останавливайтесь на достигнутом! Отдельную благодарность я хочу выразить Бреду Грину (Brad Green), Мишко Хеври (Miško Hevery), Игорю Минару (Igor Minar) и Войту Джину (Vojta Jína). Спасибо Бреду, что свел меня и Питера с издателем и побудил нас написать эту книгу. Спасибо Мишко за рецензирование нашей книги и за проявленное долготерпение, когда мы приставали с глупыми расспросами об AngularJS. Спасибо Игорю за неустанную поддержку и бесконечный поток советов, которые помогли сделать эту книгу лучше. Мы получили массу удовольствия от работы с вами!

Я хотел бы также выразить свою благодарность всему сообществу пользователей AngularJS, особенно тем, кто активно помогает в списках рассылки и на других форумах. Я не могу перечислить всех вас по именам, но ваши глубокомысленные вопросы служили для нас источником вдохновения при работе над этой книгой. Энергичное и доброжелательное сообщество, стоящее за фреймворком AngularJS, – это одна из причин, почему фреймворк получился таким удачным.

Спасибо всем сотрудникам издательства Packt Publishing: Роксане Хамбатта (Rukhsana Khambatta), Даяну Хаймсу (Dayan Hyames) и Аршаду Сопаривала (Arshad Sopariwala). Вы сделали весь процесс создания и публикации книги чрезвычайно легким и гладким. Спасибо вам!

Хочу также сказать спасибо моим коллегам из Amadeus, где я вплотную познакомился с разработкой клиентских компонентов веб-приложений. Прежде всего моим руководителям, Бертрану Ла-

порте (Bertrand Laporte) и Бруно Шабри (Bruno Chabrier). Спасибо Бертрану, что ввел меня в мир разработки клиентских компонентов и за поддержку в решении написать эту книгу. Спасибо Бруно, что позволил мне работать неполный рабочий день и сфокусироваться на этом проекте. Спасибо вам обоим за ваше великодушие. Спасибо Юлиану Дескоттсу (Julian Descottes) и Коринн Крич (Corinne Krych) за рецензирование первых рукописей книги и за весьма ценные отзывы.

Очень, очень большое спасибо Питеру, согласившемуся стать моим соавтором. Питер, я наслаждался каждой минутой работы с тобой! Я и мечтать не мог о лучшем соавторе.

Наконец, и что особенно важно для меня, я хочу поблагодарить мою невесту Аню. Без твоей поддержки и терпения я не смог бы даже приступить к работе над этой книгой.

### *От Питера Бэкона Дарвина*

Я хочу поблагодарить разработчиков из компании Google, давших нам AngularJS, и особенно тех, с кого все это началось: Мишко Хеври (Miško Hevery), Игоря Минара (Igor Minar), Бреда Грина (Brad Green) и Войта Джина (Vojta Jína). Они стали неиссякаемым источником энтузиазма. Спасибо моему соавтору, Павлу, ставшему основной движущей силой. Он придумал структуру этой книги и написал большую ее часть, а еще он – отличный парень, с которым было приятно работать. Спасибо удивительно активному сообществу, сплотившемуся вокруг AngularJS за столь короткое время, и особое спасибо участникам проекта AngularUI. Наконец, я должен сказать, что не смог бы закончить эту книгу без любви и поддержки моей супруги Келин (Kelyn) и моих детей, Лили (Lily) и Захария (Zachary).

# О РЕЦЕНЗЕНТАХ

**Стефан Биссон** (Stephane Bisson) – работал программистом в консалтинговой компании ThoughtWorks. В настоящее время живет в Торонто (Канада). Участвовал в разработке нескольких полнофункциональных веб-приложений для медицинских и финансовых учреждений, и предприятий обрабатывающей промышленности.

**Мишко Хеври** (Miško Hevery) – работал консультантом по внедрению методов гибкой разработки в Google, где отвечал за обучение сотрудников Google методам автоматизации тестирования. Его старания позволили Google чаще выпускать новые версии веб-приложений, с неизменно высоким качеством. Прежде ему приходилось работать в Adobe, Sun Microsystems, Intel и Xerox, где он стал экспертом в разработке веб-приложений с применением таких технологий, как Java, JavaScript, Flex и ActionScript. Мишко активно участвует в жизни сообщества open source и является автором нескольких проектов с открытым исходным кодом, наиболее примечательным из которых является AngularJS (http://angularjs.org).

**Ли Ховард** (Lee Howard) закончил Аппалачский государственный университет (США) по направлению информационных технологий и в настоящее время занимает должность ведущего программиста-аналитика в северо-западном отделении просвещения (Northwest Area Health Education Center) Баптистского медицинского центра Уэйк Форест (Wake Forest Baptist Health Medical Center) в городе Уинстон-Сейлем (Северная Каролина, США). Им было разработано множество различных веб-приложений, упрощающих создание, регистрацию и проведение очных и электронных образовательных курсов для Northwest AHEC. Им также было создано мобильное приложение CreditTrakr для устройств на базе iOS, позволяющее врачам и другим медицинским работникам следить за своим учебным расписанием с помощью мобильных устройств.

# ПРЕДИСЛОВИЕ

AngularJS – относительно новый фреймворк MVC на JavaScript, но уже нашедший практическое применение. В нем используются новейшие подходы к обработке шаблонов и применяется прием двунаправленного связывания данных, обеспечивающие широчайшие возможности и простоту использования фреймворка. Разработчик постоянно сообщают о существенном сокращении объема кода, необходимого для реализации приложений на основе AngularJS, в сравнении с другими подходами.

Фреймворк AngularJS – выдающийся образец инженерной мысли. С сильным упором на тестирование и высокое качество кода, он способствует использованию передового опыта, накопленного всей экосистемой JavaScript. Не удивительно, что многие разработчики, привлеченные высоким качеством кода и новизной технологий, образовали весьма активное и доброжелательное сообщество вокруг AngularJS, способствующее росту популярности фреймворка.

С ростом популярности AngularJS, все больше и больше разработчиков будут использовать его в сложных проектах. Но, как это часто бывает, вы вскоре столкнетесь с проблемами, которые не освещены в стандартной документации или в простых примерах, которые можно найти в Интернете. Как и многие другие технологии, фреймворк AngularJS имеет свой комплекс идиом, шаблонов и приемов, раскрытых сообществом на основе накопленного коллективного опыта.

И здесь вам на помощь придет эта книга – она постарается показать вам, как писать нетривиальные приложения на основе фреймворка AngularJS. В место описания особенностей работы фреймворка, эта книга концентрируется на описании особенностей использования AngularJS в сложных веб-приложениях. Здесь вы найдете ответы на многие вопросы, которые часто задаются в сообществе AngularJS.

Проще говоря, эта книга написана разработчиками приложений, для разработчиков приложений, на основе вопросов, часто задаваемых разработчиками приложений. Из этой книги вы узнаете:

- как создавать полноценные, надежные приложения с использованием имеющихся служб и директив AngularJS;
- как расширять AngularJS (создавать собственные директивы, службы, фильтры), если стандартных его возможностей оказывается недостаточно;
- как настраивать проекты приложений на основе AngularJS (организация кода, сборка, тестирование, профилирование).

# О чем рассказывается в этой книге

**Глава 1**, «Дзен Angular», служит введением в проект и фреймворк AngularJS. В этой главе вы познакомитесь с философией проекта, основными понятиями и базовыми составляющими.

**Глава 2**, «Сборка и тестирование», закладывает фундамент примера приложения, используемого в качестве иллюстрации на протяжении всей книги. Она знакомит с предметной областью и охватывает такие темы, как сборка и тестирование систем.

**Глава 3**, «Взаимодействие с серверными компонентами», рассказывает, как организовать получение данных со стороны сервера и как эффективно передавать их пользовательскому интерфейсу, действующему под управлением AngularJS. В этой главе детально будет рассматриваться доступный прикладной интерфейс API.

**Глава 4**, «Отображение и форматирование данных», предполагает, что отображаемые данные уже приняты от серверной части веб-приложения, и демонстрирует, как можно организовать отображение этих данных в пользовательском интерфейсе. Здесь обсуждается применение директив AngularJS для отображения пользовательского интерфейса и фильтров для форматирования данных.

**Глава 5**, «Создание сложных форм», иллюстрирует, как дать пользователям возможность манипулировать данными в формах и знакомит с различными типами полей ввода. Она охватывает различные способы ввода, поддерживаемые фреймворком AngularJS, и подробно рассказывает о приемах проверки данных в формах.

**Глава 6**, «Навигация», покажет, как организовать отдельные экраны в приложениях с поддержкой навигации. Глава начинается с описания роли адресов URL в одностраничных веб-приложениях и знакомит читателя с ключевыми службами AngularJS, обеспечивающими управление адресами URL и навигацией.

**Глава 7**, «Безопасность приложений», погрузит вас в детали обеспечения безопасности одностраничных веб-приложений, написанных с использованием AngularJS. В ней описываются понятия и приемы аутентификации и авторизации пользователей.

**Глава 8**, «Создание собственных директив», служит введением в разработку одних из самых интереснейших компонентов AngularJS: директив. Она познакомит читателя со структурой типовых директив и продемонстрирует приемы их тестирования.

**Глава 9**, «Создание сложных директив», основана на главе 8, «Создание собственных директив» и охватывает некоторые более сложные темы. Она наполнена практическими примерами действующих директив, ясно иллюстрирующими приемы их создания.

**Глава 10**, «Создание интернационализированных веб-приложений на основе AngularJS», описывает приемы интернационализации приложений на базе AngularJS. Охватывает темы, включающие приемы перевода шаблонов а также управления настройками, зависящими от национальных установок.

**Глава 11**, «Создание надежных веб-приложений на основе AngularJS» концентрируется на нефункциональных требованиях, предъявляемых к веб-приложениям. Она раскрывает закулисные механизмы AngularJS и знакомит читателя с характеристиками производительности. Хорошее знание внутреннего устройства AngularJS поможет вам избежать ловушек, связанных со снижением производительности приложений.

**Глава 12**, «Упаковка и развертывание веб-приложений на основе AngularJS» проведет вас через процедуру подготовки законченного веб-приложения к развертыванию. Она покажет, как оптимизировать загрузку приложений, особо остановившись на организации начальной страницы.

# Что потребуется при чтении этой книги

Для опробования любых примеров использования AngularJS, что приводятся в этой книге, вам потребуются только веб-браузер и текстовый редактор (или интегрированная среда разработки по вашему выбору). Но, чтобы получить максимум от этой книги, мы рекомендуем также установить платформу `node.js` (http://nodejs.org/) и ее диспетчер пакетов `npm` со следующими модулями:

- Grunt (http://gruntjs.com/)
- Karma runner (http://karma-runner.github.io)

Примеры кода, иллюстрирующие взаимодействие с серверными компонентами, используют облачную базу данных MongoDB (MongoLab), поэтому для опробования многих примеров необходимо так же иметь действующее подключение к Интернету.

# Кому адресована эта книга

Эта книга предназначена в основном для разработчиков, оценивших и решивших использовать в фреймворк AngularJS в своих проектах. У вас должно быть некоторое знакомство с AngularJS, хотя бы с простейшими примерами его использования. Мы полагаем, что вы обладаете знанием HTML, CSS и JavaScript.

# Соглашения

В этой книге используется несколько разных стилей оформления текста, с целью обеспечить визуальное отличие информации разных типов. Ниже приводится несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте оформляется, как показано в следующем предложении: «с помощью директивы `include` можно включать другие контексты».

Блоки программного кода оформляются так:

```
angular.module('filterCustomization', [])
  .config(function ($provide) {
varcustomFormats = {
      'fr-ca': {
        'fullDate': 'y'
      }
    };
```

Когда нам потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, мы будем выделять его жирным шрифтом :

```
<head>
<meta charset="utf-8">
<script src="/lib/angular/angular.js"></script>
<script src="/lib/angular/angular-locale_<%= locale %>.js"></script>
<base href="/<%= locale %>/">
```

**Новые термины** и **важные определения** будут выделяться в обычном тексте жирным. Текст, отображаемый на экране, например в меню или в диалогах, будет оформляться так: «щелкните на кнопке Next (Далее), чтобы перейти к следующему экрану».

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться советы и рекомендации.

# Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы читателей имеют для нас большое значение и помогают нам выпускать книги, действительно нужные вам.

Отправлять отзывы можно по адресу feedback@packtpub.com и не забудьте в теме письма указать название книги.

Если вы являетесь экспертом в какой-то области и у вас есть желание написать или представить уже готовую книгу, ознакомьтесь с руководством для авторов по адресу www.packtpub.com/authors.

# Поддержка клиентов

Теперь, когда вы приобрели книгу издательства Packt, мы можем предложить вам еще кое-что, что поможет вам извлечь максимум пользы из вашей покупки.

## *Загружаемые примеры программного кода*

Файлы с исходным кодом примеров для любой книги издательства Packt, приобретенной с использованием вашей учетной записи, можно на сайте http://www.packtpub.com. Если вы приобрели эту книгу каким-то иным способом, посетите страницу http://www.packtpub.com/support и зарегистрируйтесь, чтобы получить файлы непосредственно на электронную почту.

## Ошибки и опечатки

Мы тщательно проверяем содержимое наших книг, но от ошибок никто не застрахован. Если вы найдете ошибку в любой из наших книг – в тексте или в программном коде – мы будем весьма признательны, если вы сообщите нам о ней. Тем самым вы оградите других читателей от разочарований и поможете улучшить последующие версии этой книги. Чтобы сообщить об ошибке, посетите страницу http://www.packtpub.com/submit-errata, выберите нужную книгу, щелкните на ссылке **errata submission form** (форма отправки сообщения об ошибке) и заполните форму описанием обнаруженной ошибки. После проверки сообщения, оно будет принято и выгружено на наш веб-сайт, в раздел **Errata** (Ошибки и опечатки) для данной книги. Все обнаруженные ошибки можно увидеть на странице, по адресу: http://www.packtpub.com/support.

## Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательство Packt очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Ссылки на материалы, которые вам покажутся пиратскими, высылайте по адресу copyright@packtpub.com.

Мы высоко ценим любую помощь по защите наших авторов и помогающую нам предоставлять вам качественные материалы.

## Вопросы

Если у вас появились какие-либо вопросы, связанные с нашими книгами, присылайте их по адресу questions@packtpub.com, а мы приложим все усилия, чтобы ответить на них.

# ГЛАВА 1. Дзен Angular

Эта глава служит введением в AngularJS, фреймворк и проект, стоящий за ним. Сначала мы познакомимся с проектом: кто им управляет, где искать исходный код и документацию, как обращаться за помощью, и так далее.

Большая часть этой главы будет посвящена вводному знакомству с фреймворком AngularJS, его основными понятиями и шаблонами программирования. В ней охватывается значительный объем сведений, поэтому, чтобы ускорить процесс обучения и сделать его менее утомительным, здесь приводится довольно много примеров кода.

AngularJS – уникальный фреймворк, который, вне всяких сомнений, будет определять дальнейшие направления развития Веб в ближайшие годы. Именно поэтому в последней части главы рассказывается, что делает AngularJS таким особенным, проводится сравнение с другими существующими фреймворками и строятся прогнозы на будущее.

В этой главе рассматриваются следующие темы.

- Как написать простое приложение «Hello World» на основе AngularJS. В процессе этого вы узнаете, где взять исходный код фреймворка, где находится его документация и где искать сообщество.
- Основные строительные блоки, из которых создаются любые приложения на основе AngularJS: шаблоны с директивами, контексты и контроллеры.
- Начальные сведения о развитой системе внедрения зависимостей в AngularJS.
- Сравнение AngularJS с другими фреймворками и библиотеками (в частности с библиотекой jQuery) и описание характеристик, делающих его таким особенным.

# Знакомьтесь, AngularJS

AngularJS – это клиентский MVC-фреймворк, написанный на JavaScript. Он выполняется в веб-браузере и оказывает огромную помощь нам (разработчикам) в создании современных, одностраничных веб-приложений, использующих технологию AJAX. Это – многоцелевой фреймворк, но особенно ярко его особенности проявляются при реализации веб-приложений типа CRUD (Create Read Update Delete)[1].

## *Общие сведения о фреймворке*

AngularJS совсем недавно пополнил семейство клиентских MVC-фреймворков, тем не менее ему удалось привлечь к себе внимание, в основном благодаря своей инновационной системе шаблонов, простоте разработки с его использованием и применению надежных инженерных решений. Его система шаблонов действительно во многом уникальна:

- в качестве языка шаблонов в ней используется язык разметки HTML;
- она не требует явно обновлять дерево DOM, так как AngularJS способен следить за действиями пользователя, событиями браузера и изменениями в модели, и вовремя обнаруживать, когда и какой шаблон требуется обновить;
- имеет весьма интересную и расширяемую подсистему компонентов, и обладает возможностью обучать браузер распознаванию и правильной интерпретации новых тегов и атрибутов HTML.

Подсистема шаблонов является, пожалуй, самой заметной частью AngularJS, но было бы неправильным считать, что AngularJS – это обычный фреймворк, включающий в себя несколько утилит и служб, обычно необходимых для одностраничных веб-приложений.

AngularJS имеет в запасе несколько скрытых сокровищ, механизм **внедрения зависимостей** (Dependency Injection, DI) и сильный упор на тестируемость. Встроенная поддержка DI существенно упрощает сборку веб-приложений из небольших, надежно протестированных служб. Архитектура фреймворка и окружающих его инструментов способствует применению тестирования на всех этапах разработки.

1 Приложений, базирующихся на четырех основных операциях: «создание» (create), «чтение» (read), «изьменение» (update) и «удаление» (delete). – *Прим. перев.*

# Найдите свой путь в проект

AngularJS – относительно новый игрок на поле клиентских MVC-фреймворков; версия 1.0 была выпущена только в июне 2012. В действительности работа над этим фреймворком началась в 2009, как персональный проект Мишко Хеври (Miško Hevery), сотрудника Google. Однако идея фреймворка оказалась настолько хороша, что позднее проект был официально поддержан компанией Google Inc., и с того момента над фреймворком работает целая команда, а все работы оплачиваются компанией Google.

AngularJS – это проект с открытым исходным кодом, который можно найти на GitHub (https://github.com/angular/angular.js), и лицензируется компанией Google, Inc. на условиях лицензии MIT.

## Сообщество

По большому счету, ни один проект не выжил бы без участия людей, поддерживающих его. К счастью, вокруг AngularJS сплотилось большое сообщество. Ниже перечислены несколько каналов связи, с помощью которых можно принять участие в обсуждении проблем проекта и задать вопрос:

- список рассылки angular@googlegroups.com (в Google group);
- сообщество в Google+ (https://plus.google.com/u/0/communities/115368820700870330756);
- канал IRC #angularjs;
- тег [angularjs] на сайте http://stackoverflow.com.

Разработчики AngularJS находятся в постоянном контакте с сообществом, поддерживая блог (http://blog.angularjs.org/) и представительство в социальных сетях: в Google+ (+ AngularJS) и в Твиттере (`@angularjs`). Кроме того, сообществом достаточно часто организуются встречи по всему миру; если так случится, что одна из них будут проходить недалеко от вашего места жительства, обязательно посетите ее!

## Обучающие ресурсы в Интернете

Проект AngularJS имеет собственный веб-сайт (http://www.angularjs.org)[2], где можно найти все, что обычно ожидается от подобных проектов: концептуальный обзор, обучающие руководства, руководство разработчика, справочник API, и так далее. Исходный код всех выпу-

2 http://angular.ru/ – *Прим. перев.*

щенных версий AngularJS можно загрузить на странице http://code.angularjs.org.

Те, кто ищет примеры кода, не будут разочарованы, так как документация к фреймворку AngularJS включает массу фрагментов кода. Кроме того, можно также заглянуть в галерею приложений, созданных на основе AngularJS (http://builtwith.angularjs.org). На специализированном канале YouTube (http://www.youtube.com/user/angularjs) можно найти видеозаписи прошедших событий, а также множество весьма интересных обучающих видеороликов.

# Библиотеки и расширения

Хотя ядро AngularJS обладает всей необходимой функциональностью, наиболее активные члены сообщества практически ежедневно добавляют новые расширения. Многие из них перечислены на специализированном веб-сайте: http://ngmodules.org.

# Инструменты

Фреймворк AngularJS построен на основе HTML и JavaScript, двух технологий, давно используемых в веб-разработке. Благодаря этому мы можем продолжать использовать привычные для нас редакторы и интегрированные среды разработки (IDE), расширения для браузеров и другие инструменты. Кроме того, сообществом AngularJS было создано несколько интересных дополнений к существующему инструментальному набору HTML/JavaScript.

## *Batarang*

Batarang – дополнение к браузеру Chrome для разработчика, позволяющее исследовать веб-приложения на основе AngularJS. Дополнение Batarang очень удобно использовать для визуализации и исследования характеристик приложений AngularJS. Мы широко будем использовать его в этой книге, чтобы показать, что происходит внутри действующего приложения. Дополнение Batarang можно установить из Chrome Web Store (*AngularJS Batarang*), как и любое другое дополнение к Chrome.

### Plunker и jsFiddle

Инструменты Plunker (http://plnkr.co) и jsFiddle (http://jsfiddle.net) значительно упрощают совместное использование действующих фрагментов кода (JavaScript, CSS и HTML). Вообще эти инструменты создавались без привязки к фреймворку AngularJS, но быстро были взяты на вооружение сообществом AngularJS, чтобы обмениваться небольшими примерами кода, сценариями воспроизведения ошибок, и так далее. Инструмент Plunker заслуживает особого внимания, так как был написан в рамках проекта AngularJS и пользуется большой популярностью в сообществе.

### Расширения и дополнения для IDE

У каждого из нас имеется своя любимая среда разработки или редактор. И вас наверняка порадует факт существования расширений/дополнений для многих популярных IDE, таких как Sublime Text 2 (https://github.com/angular-ui/AngularJS-sublime-package), продуктов компании Jet Brains (http://plugins.jetbrains.com/plugin?pr=idea&pluginId=6971) и других.

## Ускоренное введение в AngularJS

Теперь, когда вы знаете, где искать исходные коды библиотек и сопутствующую документацию, можно перейти к программному коду и посмотреть на фреймворк AngularJS в действии. В этом разделе книги закладывается фундамент для всех последующих глав, и исследуются особенности поддержки в AngularJS шаблонов, механизма внедрения зависимостей и приемов модульного программирования. Все это – строительные блоки любого веб-приложения на основе AngularJS.

### Hello World – пример приложения на AngularJS

Рассмотрим пример типичного приложения «Hello, World!», реализованного на основе фреймворка AngularJS, чтобы получить первые впечатления о нем и поддерживаемом им синтаксисе.

```
<html>
<head>
    <script
       src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/
angular.js">
    </script>
</head>

<body ng-app ng-init="name = 'World'">
    <h1>Hello, {{name}}!</h1>
</body>
</html>
```

Прежде всего необходимо подключить библиотеку AngularJS, чтобы обеспечить корректную работу примера в браузере. Сделать это очень просто, так как ядро фреймворка AngularJS представляет собой единственный файл на JavaScript.

Библиотека AngularJS имеет относительно небольшой размер: минифицированная и сжатая версия имеет размер около 30 Кбайт. Минифицированная и не сжатая версия имеет размер около 80 Кбайт. К тому же, она не имеет зависимостей от сторонних библиотек.

В коротких примерах в этой книге мы будем использовать неминифицированную версию для разработчиков, доступную в сети доставки содержимого (Content Delivery Network, CDN) компании Google. Исходный код любой версии AngularJS можно загрузить на сайте http://code.angularjs.org.

Простого подключения библиотеки AngularJS недостаточно для запуска примера. Приложение необходимо еще активировать. Проще всего это сделать, добавив нестандартный HTML-атрибут `ng-app`.

Взглянув на тег `<body>` в примере, можно заметить в нем еще один нестандартный HTML-атрибут: `ng-init`. Атрибут `ng-init` используется для инициализации модели перед отображением шаблона. И последний необычный фрагмент в этом примере – выражение `{{name}}`, которое просто отображает модель.

Даже этот простой пример вскрывает несколько важнейших особенностей системы шаблонов AngularJS:

- для добавления динамического поведения в статический HTML-документ используются нестандартные теги и атрибуты HTML;
- для ограничения выражений, обеспечивающих вывод значений модели, используются двойные фигурные скобки (`{{выражение}}`).

Все специальные теги и атрибуты HTML, поддерживаемые фреймворком AngularJS, называются **директивами**.

### Двунаправленное связывание данных

Отображение шаблона с применением AngularJS реализуется очень просто; преимущества фреймворка проявляются особенно ярко, когда дело доходит до создания динамических веб-приложений. Чтобы оценить истинную мощь AngularJS, давайте дополним наш пример «Hello World» полем ввода, как показано в следующем фрагменте:

```
<body ng-app ng-init="name = 'World'">
    Say hello to: <input type="text" ng-model="name">
    <h1>Hello, {{name}}!</h1>
</body>
```

Как видите, в HTML-теге `<input>` нет ничего особенного, кроме дополнительного атрибута `ng-model`. Настоящее волшебство случается, как только выполняется попытка ввести текст в поле `<input>`. Страница на экране автоматически перерисовывается после каждого нажатия на клавиши, отражая введенное имя! Нам не нужно писать код, который перерисовывал бы шаблон, и от нас не требуется использовать какие-либо библиотечные функции, чтобы обновить модель. Фреймворк AngularJS автоматически определяет изменение модели и обновляет дерево DOM документа.

Большинство традиционных систем шаблонов используют линейную, одностороннюю процедуру отображения шаблонов: модель (переменные) и шаблон совмещаются некоторым способом, чтобы воспроизвести отображаемую разметку. Любое изменение в модели требует принудительного обновления шаблона. AngularJS действует иначе – любое изменение в представлении, произведенное пользователем, немедленно отражается на модели, а любое изменение в модели немедленно распространяется на шаблон.

## Шаблон MVC в AngularJS

Большинство существующих веб-приложений основано на хорошо известном шаблоне **Модель-Представление-Контроллер** (Model-View-Controller, MVC), реализованном в том или ином виде. Но проблема в том, что MVC – это не совсем шаблон, а скорее высокоуровневая архитектура. Кроме того, существует множество вариаций и производных оригинального шаблона (наиболее популярными

являются MVP и MVVM). Еще большую путаницу вносят различные фреймворки и разработчики, интерпретирующие упомянутые шаблоны по-разному. В результате одно и то же название MVC может использоваться для описания совершенно разных архитектур и подходов к реализации. Мартин Фаулер (Martin Fowler) так говорит об этом в своей замечательной статье «GUI Architectures» (http://martinfowler.com/eaaDev/uiArchs.html[3]):

> *Возьмем, к примеру, Model-View-Controller. Это решение часто воспринимается как шаблон, но я не вижу особой пользы воспринимать его как шаблон, хотя бы потому, что идеи, в него заложенные, различны по своей сути. Разные люди читают про MVC в различных источниках и воспринимают его идеи по-разному, но называют эти идеи одинаково — «MVC». Это приводит к замешательству и непониманию MVC, будто бы люди узнавали про него через «испорченный телефон».*

Разработчики AngularJS избрали самый прагматичный подход и объявили, что фреймворк основан на шаблоне MVW (Model-View-Whatever – Модель-Представление-Что_бы_то_ни_было). Однако, чтобы получить хоть какое-то представление об этом шаблоне, нужно видеть его в действии.

## С высоты птичьего полета

Все рассматривавшиеся выше примеры «Hello World» не дают представления об иерархии построения приложения: инициализация данных, логика и представление, все было свалено в кучу в одном файле. В настоящих приложениях, однако, необходимо больше внимания уделять разным уровням иерархии. К счастью, AngularJS реализует разнообразные архитектурные конструкции, позволяющие строить более сложные приложения.

Во всех последующих примерах в этой книге мы будем опускать код инициализации AngularJS (подключение библиотек, атрибут `ng-app` и другие элементы) для большей удобочитаемости.

Рассмотрим немного модифицированный пример «Hello World»:

```
<div ng-controller="HelloCtrl">
    Say hello to: <input type="text" ng-model="name"><br>
```

3 http://habrahabr.ru/post/50830/ – *Прим. перев.*

```
    <h1>Hello, {{name}}!</h1>
</div>
```

Мы удалили атрибут `ng-init` и добавили новую директиву `ng-controller` с соответствующей функцией на JavaScript. Функция `HelloCtrl` принимает таинственный аргумент `$scope`:

```
var HelloCtrl = function ($scope) {
    $scope.name = 'World';
}
```

### Контекст

Объект `$scope` в AngularJS служит для передачи предметной модели представлению (шаблону). Присваивая значения свойствам экземпляров контекста (scope), можно передавать новые значения для отображения в шаблоне.

Объекты контекста могут снабжаться не только данными, но и функциональными особенностями, характерными для данного представления. То есть, в экземпляре контекста можно передать шаблону некоторую логику для данного пользовательского интерфейса, определив в нем свою функцию.

Например, можно определить свою функцию получения значения переменной `name`, как показано ниже:

```
var HelloCtrl = function ($scope) {
    $scope.getName = function() {
        return $scope.name;
    };
}
```

А затем использовать ее в шаблоне:

```
<h1>Hello, {{getName()}}!</h1>
```

Объект `$scope` позволяет точно определить, какая часть предметной модели и какие операции будут доступны уровню представления. Концептуально объекты контекста в AngularJS близко напоминают ViewModel из шаблона MVVM.

### Контроллер

Основное предназначение контроллера – инициализация объектов контекста. На практике логика инициализации включает несколько этапов:

- инициализация значений в модели;
- расширение объекта `$scope` дополнительными функциональными особенностями (функциями).

Контроллеры – это обычные функции на JavaScript. Они не дополнены какими-то специфическими классами и не используют какие-либо функции фреймворка AngularJS для выполнения своей работы.

Имейте в виду, что устанавливая начальные значения модели, контроллер выполняет ту же работу, что и директива `ng-init`. Применение контроллеров позволяет выразить логику инициализации на языке JavaScript, не загромождая шаблоны HTML программным кодом.

### Модель

Модели в AngularJS – это самые обычные объекты JavaScript. От нас не требуется наследовать какие-либо классы из фреймворка или конструировать объекты моделей каким-то особым образом.

Мы можем взять любой имеющийся класс или объект на JavaScript и использовать его в качестве модели. При определении свойств моделей мы не ограничены значениями простых типов (можно использовать любой допустимый объект JavaScript или массив). Чтобы передать модель фреймворку AngularJS, достаточно просто присвоить ее переменной `$scope`.

AngularJS не навязывает никаких решений и не обязывает использовать в моделях какой бы то ни было код, так или иначе связанный с особенностями фреймворка.

## Подробнее о контекстах

Каждый объект `$scope` является экземпляром класса `Scope`. Класс `Scope` имеет методы управления жизненным циклом контекста, средства обработки событий и поддержку процедуры отображения шаблона.

### Иерархии контекстов

Рассмотрим иную реализацию простой функции `HelloCtrl`, с которой мы уже встречались выше:

```
var HelloCtrl = function ($scope) {
    $scope.name = 'World';
}
```

Эта реализация `HelloCtrl` очень напоминает обычную функцию-конструктор на JavaScript. В ней нет ничего особенного, кроме аргумента `$scope`. Но откуда берется этот аргумент?

Новый контекст был создан директивой `ng-controller`, вызовом метода `Scope.$new()`. Но постойте, похоже, что нам нужно иметь хотя бы один экземпляр контекста, чтобы создать новый контекст! Дело в том, что AngularJS имеет понятие корневого контекста `$rootScope` (родительского для всех остальных контекстов). Экземпляр `$rootScope` создается в момент инициализации нового приложения.

Директива `ng-controller` является примером директив, создающих контексты. AngularJS создает новые экземпляры класса `Scope` всякий раз, когда встречает в дереве DOM соответствующие директивы. Вновь созданный контекст ссылается на родительский контекст посредством свойства `$parent`. В дереве DOM может иметься множество директив, создающих контексты и как результат фреймворком может быть создано большое количество контекстов.

Родительские и дочерние контексты образуют древовидную иерархию с корнем в экземпляре `$rootScope`. Поскольку созданием контекстов управляет дерево DOM, не удивительно, что дерево контекстов имеет структуру, схожую с деревом DOM.

Теперь, когда стало известно, что некоторые директивы создают дочерние контексты, может возникнуть вопрос: к чему все эти сложности? Чтобы разобраться в этом, давайте рассмотрим пример, использующий директиву повторения `ng-repeat`.

Контроллер имеет следующий вид:

```
var WorldCtrl = function ($scope) {
$scope.population = 7000;
$scope.countries = [
    {name: 'France', population: 63.1},
    {name: 'United Kingdom', population: 61.8},
];
};
```

А для отображения информации используется следующая разметка:

```
<ul ng-controller="WorldCtrl">
    <li ng-repeat="country in countries">
        {{country.name}} has population of {{country.population}}
    </li>
```

```
    <hr>
    World's population: {{population}} millions
</ul>
```

Директива `ng-repeat` позволяет организовать итерации по коллекции стран и создать новые элементы DOM для каждого элемента коллекции. Она имеет простой для запоминания синтаксис; для каждого элемента коллекции создается новая переменная `country` и передается экземпляру `$scope` для отображения в представлении.

Но здесь есть одна проблема: новая переменная передается экземпляру `$scope` для каждой страны, и при этом мы не должны затереть прежде переданные значения. В AngularJS эта проблема решается созданием нового контекста для каждого элемента коллекции. Вновь созданные контексты образуют иерархию, совпадающую со структурой дерева DOM, в чем легко можно убедиться с помощью замечательного расширения Batarang для Chrome, как показано на рис. 1.1.

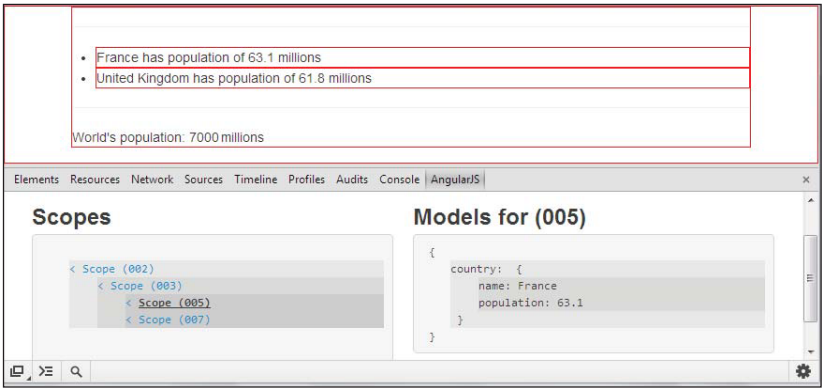


**Рис. 1.1.** Иерархия контекстов в точности соответствует иерархии элементов в дереве DOM

Как видно на этом скриншоте, каждый контекст (выделены прямоугольными границами) хранит собственный набор значений. В разных контекстах можно создать переменные с одинаковыми именами, не опасаясь конфликтов (разные элементы DOM просто будут ссылаться на разные контексты и использовать для отображения переменные из соответствующих контекстов). При такой организации каждый элемент коллекции образует собственное пространство имен. В предыдущем примере, каждый элемент `<li>` получает собственный контекст, где определяется переменная `country`.

### Иерархии областей контекстов и наследование

Свойства, определяемые в одном контексте, доступны всем дочерним контекстам, при условии, что дочерний контекст не определяет свойство с тем же именем! Это обстоятельство с успехом можно использовать на практике, так как нас ничто не обязывает снова и снова определять свойства, которые должны быть доступны во всей иерархии контекстов.

Опираясь на предыдущий пример, представим, что нам нужно выразить количество проживающих в каждой стране в процентах от общемирового народонаселения. Для этого можно определить функцию `worldsPercentage` в контексте, управляемом контроллером `WorldCtrl`, как показано ниже:

```
$scope.worldsPercentage = function (countryPopulation) {
    return (countryPopulation / $scope.population)*100;
}
```

И вызывать ее из каждого контекста, созданного директивой `ng-repeat`:

```
<li ng-repeat="country in countries">
    {{country.name}} has population of {{country.population}},
    {{worldsPercentage(country.population)}} % of the World's
population
</li>
```

Механизм наследования контекстов в AngularJS подчиняется тем же правилам, что и наследование прототипов в JavaScript (когда производится попытка прочитать значение свойства, выполняется обход дерева наследования снизу вверх, пока указанное свойство не будет найдено).

### Опасности наследования в иерархии контекстов

Механизм наследования в иерархии контекстов прост и понятен, когда речь идет о чтении свойств. Однако, когда дело доходит до операции записи, возникают определенные сложности.

Взгляните, что произойдет, если определить переменную в родительском контексте и опустить ее в дочернем. Вот код на JavaScript:

```
var HelloCtrl = function ($scope) {
};
```

А так выглядит разметка представления:

```
<body ng-app ng-init="name='World'">
<h1>Hello, {{name}}</h1>
```

```
<div ng-controller="HelloCtrl">
    Say hello to: <input type="text" ng-model="name">
    <h2>Hello, {{name}}!</h2>
</div>
</body>
```

Если попытаться выполнить этот код, вы обнаружите, что переменная `name` доступна во всем приложении; даже при том, что она определена только в контексте самого верхнего уровня! Этот пример показывает, что переменные наследуются вниз по иерархии контекстов. Иными словами, переменные, объявленные в родительском контексте, доступны потомкам.

А теперь посмотрим, что произойдет, если попытаться ввести текст в поле `<input>`, как показано на рис. 1.2.

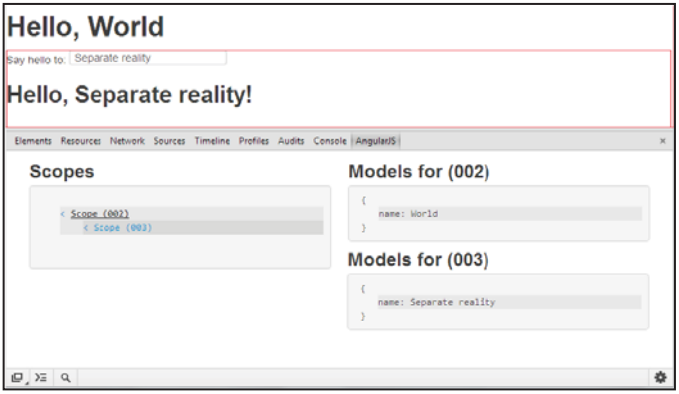


**Рис. 1.2.** Изменение переменной, отсутствующей в дочернем контексте, приводит к ее созданию

Возможно вы удивитесь, обнаружив, что в контексте, инициализированном контроллером `HelloCtrl`, появилась новая переменная, а не была изменена переменная в экземпляре `$rootScope`. Такое поведение будет менее удивительным, когда вы поймете, что контексты наследуют прототипы друг друга. Все правила, применяемые к наследованию прототипов объектов в JavaScript, так же применяются и к наследованию прототипов контекстов. В конце концов, контексты – это обычные объекты JavaScript.

Существует несколько способов изменить значение свойства в родительском контексте из дочернего. Во-первых, можно явно сослаться на родительский контекст, воспользовавшись свойством `$parent`. Для этого можно изменить шаблон, как показано ниже:

```
<input type="text" ng-model="$parent.name">
```

Хотя такой прием позволяет решить проблему в данном конкретном случае, следует понимать, что это весьма ненадежное решение. Проблема в том, что выражение, используемое в директиве `ng-model`, жестко опирается на предположения об общей структуре дерева DOM. Достаточно вставить еще одну директиву, создающую контекст, где-то выше тега `<input>`, и свойство `$parent` будет ссылаться на совершенно другой контекст.

Старайтесь не использовать свойство `$parent`, потому что оно тесно связывает выражения AngularJS со структурой DOM, создаваемой шаблонами. Приложение легко можно разрушить простым изменением структуры разметки HTML.

Другое решение заключается в использовании свойств объекта, а не свойства контекста, как показано ниже:

```
<body ng-app ng-init="thing = {name : 'World'}">
<h1>Hello, {{thing.name}}</h1>
<div ng-controller="HelloCtrl">
    Say hello to: <input type="text" ng-model="thing.name">
    <h2>Hello, {{thing.name}}!</h2>
</div>
</body>
```

Такое решение более надежно, так как не строит никаких предположений о структуре DOM.

Избегайте прямой привязки к свойствам контекста. Предпочтительнее использовать двунаправленное связывание со свойствами объектов (передаваемых в контекст). При таком подходе в выражении, передаваемом директиве `ng-model`, должен использоваться оператор точки (например, `ng-model="thing.name"`).

## Иерархия областей контекстов и система обработки событий

Контексты образуют иерархию, которую можно использовать как шину для передачи событий. Фреймворк AngularJS позволяет распространять именованные события с данными через иерархии контекстов. Событие может возбуждаться в любом контексте и передаваться либо снизу вверх (`$emit`), либо сверху вниз (`$broadcast`).

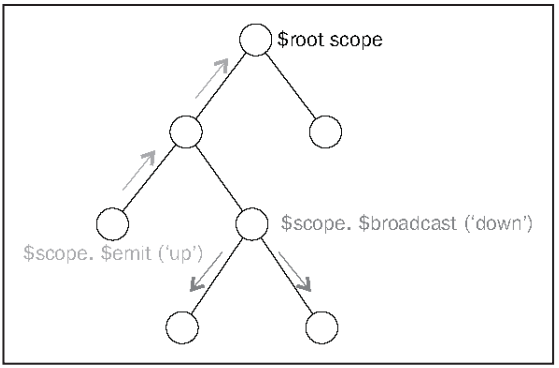


**Рис. 1.3.** Порядок распространения событий в иерархии контекстов

Базовые службы и директивы AngularJS используют такие шины для передачи сигналов о важных изменениях в состоянии приложения. Например, можно организовать прием события `$locationChangeSuccess` (распространяется вниз по иерархии из экземпляра `$rootScope`), чтобы обработать ситуацию изменения местоположения (URL в браузере), как показано в следующем фрагменте:

```
$scope.$on('$locationChangeSuccess', function(event, newUrl, oldUrl){
    // здесь реализуется реакция на изменение местоположения
    // например, обновляется отображение маршрута к странице
    // в соответствии с newUrl
});
```

Метод `$on` доступен во всех экземплярах контекстов и может применяться для регистрации обработчиков событий. Функция, действующая как обработчик, получит объект события в первом аргументе. Последующие аргументы могут использоваться для передачи дополнительных данных, зависящих от типа события.

Как и при обработке событий DOM, можно вызвать метод `preventDefault()` или `stopPropagation()` объекта события. Метод `stopPropagation()` прерывает дальнейшее всплытие события по дереву иерархии контекстов и доступен только для событий, распространяющихся снизу вверх (`$emit`).

Несмотря на близкое сходство механизмов распространения событий фреймворка AngularJS и DOM, оба они совершенно независимы друг от друга и не имеют точек пересечения.

Механизм распространения событий через иерархию контекстов позволяет получить весьма элегантное решение некоторых задач (особенно, когда требуется организовать передачу извещений в случае асинхронного изменения глобального состояния), однако его следует использовать очень экономно. Часто двустороннее связывание позволяет получить более понятное решение. Сам фреймворк AngularJS генерирует только три восходящих события (`$includeContentRequested`, `$includeContentLoaded`, `$viewContentLoaded`) и семь нисходящих (`$locationChangeStart`, `$locationChangeSuccess`, `$routeUpdate`, `$routeChangeStart`, `$routeChangeSuccess`, `$routeChangeError`, `$destroy`). Как видите, механизм распространения событий используется достаточно экономно внутри фреймворка и мы сами должны пытаться найти другие решения (обычно через двунаправленное связывание данных), прежде чем отправлять собственные события.

Не пытайтесь имитировать модель программирования DOM, управляемую событиями, в AngularJS. Часто можно найти более удачные способы структурировать приложение – механизм двунаправленного связывания данных обладает весьма широкими возможностями.

### Жизненный цикл контекстов

Контексты необходимы для организации изолированных пространств имен и исключения конфликтов имен переменных. Небольшие контексты, организованные в иерархии, помогают экономить память. Когда какой-либо контекст становится не нужен, его можно удалить. В результате модель и функциональность, связанные с этим контекстом, будут утилизированы сборщиком мусора.

Обычно контексты создаются и уничтожаются директивами, создающими контексты. Однако контексты можно также создавать и уничтожать вручную, вызывая методы `$new()` и `$destroy()`, соответственно (оба метода определены в типе `Scope`).

## Представление

Мы видели достаточно примеров шаблонов AngularJS, чтобы понять, что это не просто реализация еще одного языка шаблонов, а нечто совершенно иное. Мало того, что фреймворк поддерживает синтаксис шаблонов, основанный на языке разметки HTML, и позволяет рас-

ширять словарь HTML, он еще и обладает уникальной возможностью обновлять части экрана без постороннего вмешательства!

В действительности AngularJS имеет еще более тесную связь с HTML и DOM, потому что парсинг содержимого шаблона зависит от браузера (как и любого другого HTML-документа). Как только браузер закончит превращение текста разметки в дерево DOM, AngularJS активизируется и выполняет обход структуры DOM. Каждый раз, когда фреймворк встречает директиву, он выполняет свою логику, превращающую директиву в динамическую часть экрана.

Так как AngularJS зависит от того, как браузер выполнит парсинг шаблонов, необходимо гарантировать, что разметка является допустимым текстом на языке HTML. Особое внимание уделяйте корректному использованию закрывающих тегов HTML (в случае ошибки вы не увидите никаких предупреждающих сообщений, но представление будет отображаться неправильно). В своей работе AngularJS опирается на использование допустимого дерева DOM!

AngularJS дает возможность расширять словарь HTML (то есть, добавлять новые элементы и атрибуты HTML, и обучать браузер интерпретировать их). Это чем-то напоминает создание нового предметно-ориентированного языка (Domain-Specific Language, DSL) поверх HTML и обучение браузера обработке новых инструкций. Вы часто будете слышать, что AngularJS «обучает браузер новым трюкам».

### Декларативное представление шаблона – императивная логика контроллера

В составе AngularJS распространяется множество удобных директив, и мы расскажем о многих из них в последующих главах. Однако более важным является не синтаксис или функциональность отдельных директив, а философия AngularJS построения пользовательских интерфейсов.

AngularJS придерживается декларативного подхода к конструированию пользовательского интерфейса. С практической точки зрения это означает, что шаблоны описывают желаемый эффект, а не способ его достижения. Такое объяснение может показаться недостаточно ясным, поэтому сейчас в самый раз обратиться к примеру.

Представьте, что вам предложено создать форму, где пользователь сможет вводить короткие сообщения и отправлять их щелчком на кнопке. Существуют также некоторые дополнительные требования: длина сообщения не должна превышать 100 символов и кнопка **Send**

(Отправить) должна блокироваться, если сообщение содержит более 100 символов. Пользователь должен знать, сколько еще символов осталось у него в запасе. Если количество оставшихся символов меньше десяти, это число на экране должно отображаться с применением другого стиля, чтобы предупредить пользователя. Должна также поддерживаться возможность очистки поля ввода. В окончательном виде форма выглядит, как показано на рис. 1.4.


**Рис. 1.4.** Внешний вид формы

Предыдущие требования не особенно сложные и описывают самую, что ни на есть, стандартную текстовую форму. Однако в форме имеется несколько элементов пользовательского интерфейса, которые необходимо скоординировать между собой. Например, нужно гарантировать корректное управление блокировкой кнопки, точно отображать количество оставшихся символов, применяя разные стили для разных диапазонов значений, и так далее. Ниже показана самая первая реализация формы:

```
<div class="container" ng-controller="TextAreaWithLimitCtrl">
    <div class="row">
        <textarea ng-model="message">{{message}}</textarea>
    </div>
    <div class="row">
        <button ng-click="send()">Send</button>
        <button ng-click="clear()">Clear</button>
    </div>
</div>
```

Давайте возьмем этот код и построим свою форму на его основе. Сначала добавим отображение оставшихся символов, для чего достаточно добавить в шаблон следующую строку:

```
<span>Remaining: {{remaining()}}</span>
```

Функция `remaining()` определяется в контроллере `TextAreaWithLimitCtrl` контекста `$scope`:

```
$scope.remaining = function () {
    return MAX_LEN - $scope.message.length;
};
```

Теперь заблокируем кнопку **Send** (Отправить), если длина сообщения превышает установленный максимум. Этого легко добиться с помощью директивы `ng-disabled`:

```
<button ng-disabled="!hasValidLength()"...>Send</button>
```

Как видите, картина повторилась. Чтобы управлять пользовательским интерфейсом, достаточно затронуть небольшую часть шаблона и описать желаемый результат (отобразить количество оставшихся символов, заблокировать кнопку, и так далее) в терминах состояния модели (в данном случае – в терминах размера сообщения). Самое интересное, что здесь не требуется хранить какие-либо ссылки на элементы DOM в коде на JavaScript и не нужно явно манипулировать элементами DOM. Вместо этого мы просто сосредоточились на изменении модели и переложили всю основную работу на AngularJS. От нас потребовалось лишь дать несколько подсказок в форме директив.

Но вернемся к нашему примеру. Нам еще нужно обеспечить изменение стиля отображаемого числа, в зависимости от его величины. Это отличный повод увидеть еще один пример декларативного определения пользовательского интерфейса:

```
<span ng-class="{'text-warning' : shouldWarn()}">
    Remaining: {{remaining()}}
</span>
```

где метод `shouldWarn()` имеет следующую реализацию:

```
$scope.shouldWarn = function () {
    return $scope.remaining() < WARN_THRESHOLD;
};
```

Изменения в модели вызывают изменение класса CSS, но при этом нам не пришлось предусматривать операции с деревом DOM в коде на JavaScript! Перерисовывание пользовательского интерфейса реализовано простым декларативным выражением «желания». В данном случае с помощью директивы мы сказали: «CSS-класс `text-warning` следует добавлять в элемент `<span>` всегда, когда требуется предупредить о приближении к пределу количества символов». Это существенно отличается от алгоритма; «когда при вводе нового символа количество символов оказывается выше некоторого порога, следует найти элемент `<span>` и добавить в него CSS-класс `text-warning`».

Различия между обсуждаемыми здесь подходами могут показаться незначительными, но в действительности декларативный и импе-

ративный подходы являются полными противоположностями. При использовании императивного стиля разработчик сосредотачивается на описании отдельных этапов, ведущих к желаемому результату. При использовании декларативного подхода все внимание смещается ближе к желаемому результату, а о выполнении операций, необходимых для его достижения, позаботится фреймворк. Это можно сравнить с просьбой: «Уважаемый AngularJS, я хочу, чтобы пользовательский интерфейс выглядел вот так, когда модель достигнет определенного состояния. А теперь пойдите и решите, когда и как обновлять пользовательский интерфейс».

Декларативный стиль программирования обычно более выразительный, так как освобождает разработчика от необходимости давать подробные, низкоуровневые инструкции. Получающийся код часто получается очень кратким и легко читаемым. Но, чтобы иметь возможность использовать декларативный подход, необходим некоторый механизм, который сможет правильно интерпретировать высокоуровневые инструкции. Программы зависят от решений, реализованных в таком механизме, и мы теряем часть низкоуровневого контроля. При использовании императивного подхода мы имеем полный контроль над происходящим и можем тонко настраивать выполнение каждой отдельной операции. Но за полный контроль приходится платить необходимостью писать массу низкоуровневого, повторяющегося кода.

Читатели, знакомые с языком SQL, найдут вышесказанное знакомым (язык SQL – очень выразительный, декларативный язык для запроса данных). Мы можем просто описать желаемый результат (характеристики извлекаемых данных) и позволить базе данных (реляционной) самой решить, как извлекать указанные данные. В большинстве случаев эта процедура действует безупречно и мы быстро получаем желаемый результат. Однако иногда нам все еще приходится давать дополнительные подсказки (определять индексы, планировать запросы и так далее) или брать в свои руки управление процессом извлечения данных, чтобы добиться более высокой производительности.

Директивы в шаблонах AngularJS декларативно выражают желаемый результат, благодаря чему мы освобождаемся от необходимости давать пошаговые инструкции о том, как изменять отдельные свойства элементов DOM (как это часто случается в приложениях, основанных на библиотеке jQuery). Фреймворк AngularJS настойчиво подталкивает разработчика к использованию декларативного стиля

программирования при работе с шаблонами, и императивного – при работе с кодом на JavaScript (контроллеры и логика выполнения). При использовании AngularJS нам редко придется опускаться до низкоуровневых императивных инструкций, чтобы выполнить манипуляции с деревом DOM (единственное исключение – код, реализующий сами директивы).

Как правило, следует избегать операций с элементами DOM в контроллерах AngularJS. Получение ссылки на элемент DOM в контроллере и выполнение манипуляций со свойствами элементов – это императивный подход к управлению пользовательским интерфейсом, который противоречит философии построения пользовательских интерфейсов в AngularJS.

Возможность декларативного описания шаблонов пользовательского интерфейса с помощью директив AngularJS позволяет быстро создавать сложные и интерактивные интерфейсы. А принятие решений о том, когда и как изменять элементы DOM берет на себя AngularJS. В подавляющем большинстве случаев AngularJS принимает «правильные решения» и обновляет интерфейс должным образом (и своевременно). Однако для нас важно разобраться с тем, как действуют внутренние механизмы AngularJS, чтобы при необходимости мы могли дать дополнительные подсказки. Если вернуться к аналогии с языком SQL, в большинстве случаев нам не приходится задумываться о том, как действует планировщик запросов. Но когда мы сталкиваемся с проблемами производительности, знакомство с особенностями работы планировщика поможет нам дать ему дополнительные подсказки. То же правило применимо и к пользовательским интерфейсам, управляемым фреймворком AngularJS: для эффективного использования директив и шаблонов важно иметь полное представление о механизмах, на которых они базируются.

## Модули и внедрение зависимостей

Наиболее внимательные читатели наверняка заметили, что во всех примерах, приводившихся до сих пор, для определения контроллеров использовались глобальные функции-конструкторы. Но использование глобального состояния – это пагубная практика, она мешает созданию гибкой структуры приложения, усложняет сопровождение кода, его тестирование и чтение. В AngularJS отсутствуют компоненты, требующие использования именно глобального состо-

яния. Напротив, фреймворк предоставляет прикладной интерфейс, который облегчает создание модулей и регистрацию объектов в этих модулях.

## Модули в AngularJS

Давайте посмотрим, как превратить уродливый глобальный контроллер в его модульный эквивалент. Прежде контроллер был объявлен так:

```
var HelloCtrl = function ($scope) {
    $scope.name = 'World';
}
```

А с использованием модулей его объявление изменилось, как показано ниже:

```
angular.module('hello', [])
    .controller('HelloCtrl', function($scope){
        $scope.name = 'World';
    });
```

Сам фреймворк AngularJS определяет глобальное пространство имен `angular`. В нем находятся различные вспомогательные функции и утилиты, в число которых входит также функция `module`. Функция `module` действует как контейнер для других объектов (контроллеров, служб и так далее), управляемых фреймворком AngularJS. Как будет показано ниже, модули в AngularJS – это не только пространства имен, используемые для организации кода.

Чтобы определить новый модуль необходимо передать функции `module` его имя в первом аргументе. С помощью второго аргумента можно определить зависимости от других модулей (в примере выше определяемый модуль не имеет зависимостей от других модулей).

В ответ функция `angular.module` возвращает экземпляр вновь созданного модуля. Получив доступ к этому экземпляру, мы можем приступать к определению новых контроллеров, для чего достаточно вызвать функцию `controller` со следующими аргументами:

- имя контроллера (строка);
- функция-конструктор контроллера.

Глобальные функции-конструкторы контроллеров хороши только когда требуется быстро набросать и опробовать прототип будущего контроллера. Никогда не используйте глобальные функции контроллеров в больших, настоящих приложениях.

Итак, модуль объявлен, теперь нужно сообщить фреймворку AngularJS о его существовании. Для этого достаточно указать имя модуля в атрибуте `ng-app`, как показано ниже:

```
<body ng-app="hello">
```

Разработчики часто забывают указать имя модуля в атрибуте `ng-app`, что приводит к путанице и появлению ошибок, указывающих, что контроллер не определен.

## Взаимодействие объектов

Как было показано выше, AngularJS обеспечивает возможность организации объектов в модули. Но модули можно использовать не только для регистрации объектов, которые вызываются фреймворком непосредственно (контроллеров, фильтров и так далее), но и любых других объектов, определяемых разработчиками.

Модули чрезвычайно удобны для организации кода, но AngularJS делает еще шаг вперед. Помимо регистрации объектов в пространствах имен он позволяет декларативно описывать зависимости между объектами.

### Внедрение зависимостей

Мы уже видели магическое внедрение объекта `$scope` в экземпляры контроллеров. Фреймворк AngularJS каким-то способом обнаруживает, что контроллеру необходим новый экземпляр контекста, создает его и внедряет. Единственное, что требуется от контроллера – выразить свою зависимость от экземпляра `$scope` (при этом не требуется указывать, как должен быть создан новый объект `$scope`, и вообще должен ли быть создан новый экземпляр или может повторно использоваться экземпляр, созданный предыдущим вызовом). Управление зависимостями целиком сводится к объявлению своих пожеланий, которое на простом языке можно выразить так: «Для корректной работы функции необходим вспомогательный объект: я не знаю, где взять или как создать его, я знаю только, что он мне нужен, поэтому дайте мне его, пожалуйста!».

AngularJS имеет встроенный механизм внедрения зависимостей (Dependency Injection, DI). Он может выполнять следующие действия:

- распознавать потребности во вспомогательных объектах, выражаемых декларативным способом;

- находить необходимые вспомогательные объекты;
- связывать объекты между собой.

Возможность декларативно выражать зависимости предоставляет широчайшие возможности; она освобождает от необходимости беспокоиться о жизненном цикле вспомогательных объектов. Что еще лучше, взаимозаменяемость вспомогательных объектов позволяет создавать различные приложения простой заменой одних служб другими. Это также ключевой аспект эффективности модульного тестирования.

### Преимущества приема внедрения зависимостей

Чтобы увидеть весь потенциал применения приема внедрения зависимостей, рассмотрим пример создания службы извещений, которой можно передавать сообщения и извлекать их позднее. Чтобы несколько усложнить сценарий, допустим, что нам требуется организовать также службу архивирования. Она должна взаимодействовать со службой извещений следующим образом: как только количество извещений превысит некоторое пороговое значение, самые старые извещения должны быть сохранены в архив. Дополнительная сложность заключается в том, что должна иметься возможность использовать разные службы архивирования в разных приложениях. Иногда достаточно будет сбросить старые сообщения в консоль браузера; иногда необходимо будет передать их на сервер с помощью вызовов XHR.

Ниже показана реализация службы извещений:

```
var NotificationsService = function () {
    this.MAX_LEN = 10;
    this.notificationsArchive = new NotificationsArchive();
    this.notifications = [];
};

NotificationsService.prototype.push = function (notification) {

    var newLen, notificationToArchive;

    newLen = this.notifications.unshift(notification);
    if (newLen > this.MAX_LEN) {
        notificationToArchive = this.notifications.pop();
        this.notificationsArchive.archive(notificationToArchive);
    }
};

NotificationsService.prototype.getCurrent = function () {
    return this.notifications;
};
```

Предыдущий код тесно связан с единственной реализацией архивирования (`NotificationsArchive`), так как создает экземпляр именно этой реализации с помощью ключевого слова `new`. Это не самое удачное решение, начиная от единственно возможного контракта, которого вынуждены придерживаться оба класса, до метода `archive` (принимающего извещения, подлежащие сохранению в архиве).

Возможность замены вспомогательных объектов чрезвычайно важна для тестирования. Сложно представить, как можно всесторонне протестировать некоторый объект, не имея возможности подменять фактические вспомогательные объекты их фиктивными реализациями. Далее в этой главе мы покажем, как превратить тесно связанный конгломерат объектов в гибкий набор взаимодействующих служб, пригодных для всестороннего тестирования. Для этой цели мы будем использовать преимущества подсистемы внедрения зависимостей в AngularJS.

## Регистрация служб

AngularJS способен связывать только объекты, знакомые ему. По этой причине, самый первый шаг, который следует выполнить перед включением механизма DI, – регистрация объектов в модуле AngularJS. Нам не требуется регистрировать экземпляры объектов непосредственно, скорее мы должны передать механизму внедрения зависимостей рецепты по их созданию. После этого фреймворк AngularJS будет создавать объекты по указанным рецептам и связывать их между собой. В результате получится множество взаимосвязанных объектов, образующих действующее приложение.

В AngularJS существует выделенная служба `$provide`, позволяющая регистрировать различные рецепты создания объектов. Зарегистрированные рецепты затем используются службой `$injector` для создания готовых к применению экземпляров объектов (со всеми внедренными зависимостями).

Объекты, созданные службой `$injector`, так же называют службами. Каждый рецепт интерпретируется фреймворком AngularJS только один раз в течение жизни приложения и, как результат, создает только один экземпляр объекта.

Службы, созданные с помощью `$injector`, являются объектами-одиночками (singletons). В приложении может существовать только один экземпляр каждой такой службы.

В конечном итоге, модуль AngularJS просто хранит множество экземпляров объектов, но мы можем управлять порядком их создания.

### Значения

Самый простой способ передать объект под управление фреймворку AngularJS – зарегистрировать его экземпляр, как показано ниже:

```
var myMod = angular.module('myMod', []);
myMod.value('notificationsArchive', new NotificationsArchive());
```

Любая служба, управляемая механизмом AngularJS DI, должна иметь уникальное имя (например, `notificationsArchive` в предыдущем примере). А все остальное является рецептом создания новых экземпляров.

Объекты-значения не представляют большого интереса, так как объекты, регистрируемые этим методом, не могут зависеть от других объектов. Впрочем, для экземпляра `NotificationArchive` это не является проблемой, так как он не имеет никаких зависимостей. На практике этот метод используется только для регистрации самых простых объектов (обычно экземпляров встроенных объектов или объектов-литералов).

### Службы

Службу `NotificationsService` нельзя зарегистрировать как объект-значение, потому что нам требуется выразить его зависимость от службы архивирования. Самый простой способ зарегистрировать рецепт создания объектов, зависящих от других объектов – указать функцию-конструктор. Сделать это можно с помощью метода `service`:

```
myMod.service('notificationsService', NotificationsService);
```

где конструктор `NotificationsService` может быть теперь реализован, как показано ниже:

```
var NotificationsService = function (notificationsArchive) {

    this.notificationsArchive = notificationsArchive;

};
```

Воспользовавшись механизмом внедрения зависимостей в AngularJS, мы смогли избавиться от ключевого слова `new` в конструкторе `NotificationsService`. Теперь эта служба не занимается созданием экземпляра своей зависимости и может принять любую службу

архивирования. Теперь наше простенькое приложение стало намного гибче!

Термин «служба» слишком перегружен и может использоваться для обозначения самых разных понятий. В AngularJS под термином «служба» (service) подразумевается либо метод (`service`) регистрации конструкторов (как показано в примере выше), либо произвольный объект-одиночка (singleton), созданный и управляемый подсистемой AngularJS DI, независимо от имени метода, применявшегося для его регистрации (именно этот смысл вкладывают в термин «служба» большинство разработчиков в контексте модулей AngularJS).

На практике метод `service` применяется нечасто, но его удобно использовать для регистрации уже имеющихся функций-конструкторов и тем самым заставлять AngularJS управлять объектами, созданными с помощью этих конструкторов.

### Фабрики

Метод `factory` – еще один способ регистрации рецептов создания объектов. Он обеспечивает большую гибкость, чем метод `service`, так как способен регистрировать любые функции, создающие и возвращающие объекты. Взгляните на следующий фрагмент:

```
myMod.factory('notificationsService', function(notificationsArchive){

    var MAX_LEN = 10;
    var notifications = [];

    return {
        push:function (notification) {
            var notificationToArchive;
            var newLen = notifications.unshift(notification);

            // метод push может использовать поддержку замыканий!
            if (newLen > MAX_LEN) {
                notificationToArchive = this.notifications.pop();
                notificationsArchive.archive(notificationToArchive);
            }
        },
        // другие методы NotificationsService
    };
```

AngularJS будет использовать указанную фабричную функцию для регистрации возвращаемого объекта. Им может быть любой допустимый объект JavaScript, даже функции!

Метод `factory` – наиболее часто используемый прием регистрации объектов в подсистеме внедрения зависимостей AngularJS. Он обладает большой гибкостью и может принимать достаточно сложную логику создания. Так как фабрики фактически являются обычными функциями, мы можем использовать новую лексическую область видимости для имитации «приватных» (private) переменных. Это очень удобно, когда желательно скрыть тонкости реализации службы. В действительности, в предыдущем примере `notificationToArchive` все конфигурационные параметры (`MAX_LEN`) и внутреннее состояние (`notifications`) являются «приватными» переменными.

### Константы

Наша служба `NotificationsService` становится лучше и лучше: мы разорвали тесную связь между взаимодействующими объектами и скрыли ее приватное состояние. К сожалению, здесь все еще присутствует жестко «зашитая» в код константа `MAX_LEN`. В AngularJS имеется решение и этой проблемы – фреймворк позволяет определять константы на уровне модуля и внедрять их в объекты.

В идеале хотелось бы, чтобы служба `NotificationsService` получала параметры настройки следующим способом:

```
myMod.factory('notificationsService',

function (notificationsArchive, MAX_LEN) {
    ...
    // логика создания не изменилась
});
```

А затем определять параметры настройки за пределами `NotificationsService`, на уровне модуля:

```
myMod.constant('MAX_LEN', 10);
```

Константы очень удобны для создания служб, которые могут использоваться различными приложениями (так как клиенты определять свои настройки для служб). Единственный недостаток констант – если служба объявляет зависимость от константы, клиент обязан определить значение этой константы. Иногда было бы неплохо иметь параметры настройки со значениями по умолчанию и позволять клиентам изменять только те из них, которые действительно требуют изменения.

### Провайдеры

Все методы регистрации, описанные к настоящему моменту, являются всего лишь специальными случаями более универсального метода `provider`. Ниже представлен пример регистрации службы `notificationsService` как провайдера (provider):

```
myMod.provider('notificationsService', function () {

    var config = {
        maxLen : 10
    };
    var notifications = [];

    return {
        setMaxLen : function(maxLen) {
            config.maxLen = maxLen || config.maxLen;
        },

        $get : function(notificationsArchive) {
            return {
                push:function (notification) {
                    ...
                    if (newLen > config.maxLen) {
                        ...
                    }
                },
                // другие методы
            };
        }
    };
});
```

В первую очередь провайдер – это функция, которая должна вернуть объект, обладающий свойством `$get`. Свойство `$get` – это фабричная функция, возвращающая экземпляр службы. Провайдеры можно считать объектами, встраивающими фабричные функции в свое свойство `$get`.

Объект, возвращаемый функцией `provider`, может иметь дополнительные методы и свойства. Эти методы и свойства можно использовать для определения параметров настройки перед вызовом фабричного метода `$get`. Фактически, мы все еще можем установить свойство `maxLen`, но уже не обязаны делать это. Кроме того, есть возможность определить более сложную логику настройки, реализовав в службе методы вместо простых конфигурационных значений.

## Жизненный цикл модулей

В предыдущих абзацах было показано, что AngularJS поддерживает различные рецепты создания объектов. Провайдер – это рецепт особого рода, позволяющий выполнить дополнительные настройки перед созданием экземпляров объектов. Для эффективной поддержки провайдеров, AngularJS разбивает жизненный цикл модуля на две фазы:

- **фаза настройки**: в течение этой фазы производится сбор и настройка всех рецептов;
- **фаза выполнения**: в течение этой фазы выполняется любая логика после создания экземпляров.

### Фаза настройки

Провайдеры могут настраиваться только в ходе первой фазы – фазы настройки. Это вполне объяснимо: какой смысл менять рецепт уже после создания объектов? Настройка провайдеров выполняется, как показано в следующем фрагменте:

```
myMod.config(function(notificationsServiceProvider){
    notificationsServiceProvider.setMaxLen(5);
});
```

Обратите внимание на зависимость от объекта `notificationsServiceProvider` (с окончанием `Provider` в имени), представляющего рецепт, готовый к использованию. Наличие фазы настройки дает нам возможность внести заключительные изменения в формулу создания объектов.

### Фаза выполнения

Наличие фазы выполнения позволяет регистрировать любые задания, которые должны быть выполнены в ходе инициализации приложения. Фазу выполнения можно сравнить с методом `main` в других языках программирования. Основное отличие состоит в том, что модули в AngularJS могут иметь несколько блоков настройки и выполнения. То есть, здесь нет единственной точки входа (выполняющееся приложение действительно является коллекцией взаимодействующих объектов).

Чтобы показать практическую ценность фазы выполнения, давайте представим, что нам нужно показать время запуска приложения (или продолжительность его работы). С этой целью можно было бы определить время запуска приложения, как свойство экземпляра `$rootScope`:

```
angular.module('upTimeApp', []).run(function($rootScope) {
    $rootScope.appStarted = new Date();
});
```

И затем извлекать его значение в том или ином шаблоне:

```
Application started at: {{appStarted}}
```

В примере выше, демонстрирующем блок `run` в действии, выполняется непосредственная инициализация свойств экземпляра `$rootScope`. Важно помнить, что `$rootScope` – это глобальная переменная и она подвержена всем проблемам, которыми страдает глобальное состояние. Экземпляр `$rootScope` следует использовать очень экономно для определения новых свойств и только если эти свойства должны быть доступны в нескольких шаблонах.

### Различия между фазами и различия между методами регистрации

В табл. 1.1 перечислены разные методы создания объектов, и как они соотносятся с разными фазами жизненного цикла модулей.

**Таблица 1.1.** Методы создания объектов и фазы жизненного цикла модулей

| Метод | Что регистрирует | Внедряется на этапе настройки | Внедряется на этапе выполнения |
|---|---|---|---|
| `constant` | Постоянное значение | Да | Да |
| `value` | Значение переменной | – | Да |
| `service` | Новый объект, создаваемый функцией-конструктором | – | Да |
| `factory` | Новый объект, создаваемый фабричной функцией | – | Да |
| `provider` | Новый объект, создаваемый фабричной функцией `$get` | Да | – |

### Зависимости между модулями

Фреймворк AngularJS не только прекрасно справляется с зависимостями между объектами, но и способен обрабатывать зависимости между модулями. Мы легко можем сгруппировать взаимосвязанные службы в один модуль и тем самым создать библиотеку служб (пригодную для многократного использования).

Например, мы можем поместить службы извещений и архивирования в собственные модули (с именами `notifications` и `archive`, соответственно) и затем объединить их, как показано ниже:

```
angular.module('application', ['notifications', 'archive'])
```

Таким способом можно каждую службу (или группу взаимосвязанных служб) встроить в модуль, пригодный для повторного использования. В конечном итоге модуль самого верхнего уровня (уровня приложения) может объявить зависимости от всех модулей, необходимых данному приложению для решения стоящих перед ним задач.

Возможность объявлять зависимости от других модулей не является исключительной привилегией модулей верхнего уровня. Любой модуль может объявлять зависимости от дочерних модулей. Таким способом можно конструировать иерархии модулей. То есть, при работе с модулями в AngularJS следует иметь в виду, что они могут представлять две разные, но взаимосвязанные иерархии: иерархию модулей и иерархию служб (потому что службы так же могут иметь зависимости от других служб, значений и констант).

Модули в AngularJS могут зависеть друг от друга, и каждый модуль может содержать несколько служб. Но отдельные службы тоже могут зависеть от других служб. В связи с этим возникает несколько интересных вопросов.

- Может ли служба, объявленная в одном модуле, зависеть от служб в другом модуле?
- Может ли служба, объявленная в дочернем модуле, зависеть от службы в родительском модуле, или она может зависеть только от служб в дочерних модулях?
- Можно ли определять приватные службы, недоступные за пределами модуля?
- Можно ли в разных модулях определить службы с одинаковыми именами?

## Службы и их доступность в других модулях

Как вы уже наверняка догадались, службы, объявленные в дочерних модулях, доступны для внедрения в службы, объявленные в родительских модулях, как показано в следующем фрагменте:

```
angular.module('app', ['engines'])

    .factory('car', function ($log, dieselEngine) {
        return {
            start: function() {
```

```
                $log.info('Starting ' + dieselEngine.type);
            };
        }
    });

angular.module('engines', [])
    .factory('dieselEngine', function () {
        return {
            type: 'diesel'
        };
    });
```

Здесь служба `car` определена в модуле `app`. Модуль `app` объявляет зависимость от модуля `engines`, где определена служба `dieselEngine`. Едва ли кого-то удивит тот факт, что в `car` (автомобиль) может быть внедрен экземпляр службы `dieselEngine` (дизельный двигатель).

Гораздо удивительнее, что службы, объявленные в модулях одного уровня, также доступны друг другу. Мы можем поместить службу `car` в отдельный модуль и затем изменить зависимости модулей так, что приложение будет зависеть сразу от двух модулей, `cars` и `engines`:

```
angular.module('app', ['engines', 'cars'])

angular.module('cars', [])
    .factory('car', function ($log, dieselEngine) {
        return {
            start: function() {
                $log.info('Starting ' + dieselEngine.type);
            }
        };
    });

angular.module('engines', [])
    .factory('dieselEngine', function () {
        return {
            type: 'diesel'
        };
    });
```

В этом примере экземпляр `engine` так же может быть внедрен в экземпляр `car`.

Служба, объявленная в одном из модулей приложения, будет доступна во всех других модулях. Иными словами, положение модулей в иерархии не влияет на доступность его служб в других модулях. Когда AngularJS инициализирует приложение, он объединяет все службы, объявленные во всех модулях в единое приложение, то есть, включает их в глобальное пространство имен.

Так как фреймворк AngularJS объединяет все службы из всех модулей в одно большое множество служб, не может существовать двух или более служб с одинаковыми именами. Это обстоятельство можно использовать в своих интересах, например, когда необходимо объявить зависимость от некоторого модуля и переопределить некоторые из его служб. Чтобы продемонстрировать это, давайте переопределим службу `dieselEngine` непосредственно в модуле `cars`:

```
angular.module('app', ['engines', 'cars'])
    .controller('AppCtrl', function ($scope, car) {
        car.start();
    });

angular.module('cars', [])
    .factory('car', function ($log, dieselEngine) {
        return {
            start: function() {
                $log.info('Starting ' + dieselEngine.type);
            };
        }
    })

    .factory('dieselEngine', function () {
        return {
            type: 'custom diesel'
            };
        });
```

В данном случае в службу `car` будет внедрена служба `dieselEngine`, объявленная в том же модуле, что и служба `car`. Служба `dieselEngine` из модуля `car` переопределит (скроет) службу `dieselEngine` из модуля `engines`.

В приложении на основе AngularJS может существовать только одна служба с заданным именем. Службы в модулях, находящихся ближе к корню иерархии, будут переопределять (скрывать) службы в дочерних модулях.

В текущей версии AngularJS все службы, объявленные в одном модуле, будут доступны во всех остальных модулях. Нет никакой возможности ограничить область видимости службы некоторым подмножеством модулей.

На момент написания этих строк в фреймворке отсутствовала поддержка приватных служб.

### Зачем нужны модули в AngularJS

Тот факт, что AngularJS объединяет все службы из всех модулей в одно большое пространство имен, может показаться удивительным, и у кого-то даже возникнет вопрос: зачем вообще использовать модули, если в конце концов все службы окажутся в одной большой котомке? Какой смысл затрачивать лишние усилия, чтобы разносить их по отдельным модулям?

Поддержка модулей в AngularJS может помочь разбить приложение на несколько файлов JavaScript. Существует множество стратегий деления приложения на модули, и мы посвятим большую часть главы 2, «Сборка и тестирование», обсуждению различных подходов, их «за» и «против». Кроме того, разбиение на маленькие модули облегчает их тестирование, так как позволяет загружать для тестирования четко определенное множество служб. Напомню еще раз, что дополнительные подробности по этому вопросу вы найдете в главе 2.

## AngularJS и остальной мир

Выбор лучшего фреймворка в качестве основы следующего проекта – достаточно сложная задача для любого разработчика. Некоторые фреймворки могут лучше соответствовать приложениям определенного типа, практическому опыту команды или личным предпочтениям. На окончательный выбор может влиять целое множество факторов.

Фреймворк AngularJS неизбежно будет сравниваться с другими популярными MV*-фреймворками на JavaScript. Разные сравнения наверняка приведут к разным результатам, а разные точки зрения будут питать жаркие дискуссии. Вместо того чтобы предложить точные критерии сравнения, мы хотели бы рассказать об отличиях AngularJS от других фреймворков.

Желающие увидеть, чем отличается код, использующий AngularJS, от кода, использующего другие фреймворки, могут посетить сайт TodoMVC (http://addyosmani.github.com/todomvc). Это – проект, где можно увидеть реализацию одного и того же приложения (список дел) с использованием разных MV*-фреймворков на JavaScript. Он дает уникальную возможность сравнить архитектурные подходы и синтаксис, объем кода и его читаемость.

Фреймворк AngularJS обладает множеством особенностей, выделяющих его из общего ряда. Мы уже видели, насколько новаторским

выглядит его подход к обработке шаблонов пользовательского интерфейса, особенности которого перечислены ниже.

- Автоматическое обновление и двунаправленное связывание данных освобождают разработчиков от утомительной работы по организации явных вызовов функций перерисовывания пользовательского интерфейса.
- Из разметки HTML, используемой как язык шаблонов, генерируется «живое» дерево DOM. Но, что особенно важно, фреймворк AngularJS позволяет расширять словарь языка HTML (за счет создания новых директив), и затем конструировать пользовательские интерфейсы с применением нового предметного языка (DSL), основанного на HTML.
- Декларативный подход к определению пользовательских интерфейсов позволяет получить весьма краткий и выразительный код.
- Превосходный механизм шаблонов не накладывает никаких ограничений на код JavaScript (например, модели и контроллеры могут создаваться вообще без использования AngularJS API).

AngularJS закладывает новый прочный фундамент, привнося устоявшиеся приемы тестирования в мир JavaScript. Сам фреймворк тщательно протестирован (практика, которую многие проповедуют!), но история с тестированием на этом не заканчивается, не только фреймворк, но и вся экосистема вокруг него была построена так, что подразумевает обязательное тестирование. Это выражается в следующем.

- Механизм внедрения зависимостей поддерживает тестирование, позволяя составлять приложения из маленьких служб, легко поддающихся тестированию.
- Большинство примеров кода в документации к фреймворку AngularJS сопровождается тестами, являющимися лучшим доказательством, что код, написанный для AngularJS, действительно легко поддается тестированию!
- Коллективом разработчиков AngularJS был создан на JavaScript замечательный инструмент запуска тестов, с названием Testacular. Этот инструмент превращает процесс тестирования в увлекательный эксперимент. Тестирование иногда может оказаться сложной процедурой, поэтому так важно иметь инструменты, помогающие нам, а не стоящие у нас на пути.

Самое главное, AngularJS снова сделал разработку веб-приложений увлекательным занятием! Он берет на себя решение такого количества рутинных задач, что код приложения получается чрезвычайно кратким. Часто можно услышать, что переход на AngularJS уменьшает объем кода приложения в пять раз, и даже больше. Разумеется, все зависит от конкретного приложения и команды, и все таки AngularJS позволяет двигаться к намеченной цели быстрее и выдавать результаты в мгновение ока!

## jQuery и AngularJS

AngularJS и jQuery находятся в весьма интересных взаимоотношениях, требующих особого упоминания. Для начала отмечу, что фреймворк AngularJS включает в себя упрощенную версию библиотеки jQuery – jqLite. В действительности эта версия реализует ограниченное подмножество функциональных возможностей полной библиотеки jQuery, касающихся только манипуляций с деревом DOM.

Благодаря встраиванию библиотеки jqLite, фреймворк AngularJS может работать без привлечения каких-либо внешних библиотек.

Но фреймворк AngularJS является добропорядочным членом сообщества JavaScript и способен рука об руку работать совместно с jQuery. Обнаружив присутствие jQuery в приложении, AngularJS будет использовать механизмы управления деревом DOM из этой библиотеки, вместо предлагаемых минимальной версией jqLite.

Если планируется использовать в приложении библиотеку jQuery совместно с AngularJS, ее необходимо подключать первой, до подключения сценария AngularJS.

Однако, если попытаться использовать какие-либо компоненты пользовательского интерфейса из библиотеки jQuery UI, ситуация становится намного сложнее. Некоторые из них будут продолжать работать, не вызывая никаких проблем, но чаще вам придется преодолевать различные трудности. Эти две библиотеки реализуют настолько разные философии, что не стоит ожидать от них бесшовной интеграции. В главе 8, «Создание собственных директив», мы погрузимся в вопросы интеграции и создания виджетов пользовательского интерфейса, которые корректно работают в приложениях на основеAngularJS.

### Яблоки и апельсины

jQuery and AngularJS поддерживают возможность сотрудничества, но их нельзя сравнивать непосредственно. Прежде всего, jQuery зарождалась как библиотека инструментов, упрощающих управление деревом DOM, и потому основными ее функциями являются обход узлов документа, обработка событий, воспроизведение анимационных эффектов и выполнение взаимодействий с применением технологии Ajax.

AngularJS, напротив, представляет собой полноценный фреймворк, пытающийся управлять всеми аспектами разработки современных приложений Web 2.0.

Самое важное, о чем следует помнить, – AngularJS реализует совершенно иной подход к созданию пользовательских интерфейсов, в соответствии с которым декларативно объявленное представление управляется изменениями в модели. При использовании библиотеки jQuery слишком часто приходится писать код, управляющий деревом DOM, который по мере роста проекта легко может выйти из-под контроля (и в смысле объема, и в смысле функционирования).

Парадигмы, реализованные в AngularJS и jQuery, кардинально отличаются друг от друга. Разработчики, обладающие большим опытом использования jQuery, приступая к использованию AngularJS, легко попадают в ловушку привычки мыслить парадигмой jQuery. В результате они начинают «бороться с AngularJS», вместо того, чтобы использовать его богатый потенциал. Именно поэтому мы рекомендуем не использовать jQuery в процессе изучения AngularJS (чтобы не испытывать соблазн вернуться к старым привычкам и научиться решать проблемы способом, характерным для AngularJS).

Фреймворк AngularJS проповедует целостный подход к разработке современных веб-приложений и пытается сделать браузер лучшей платформой для разработки.

## Взгляд в будущее

Фреймворк AngularJS несет по-настоящему новаторский взгляд на многие аспекты веб-разработки и формирует новые подходы к созданию кода для будущих браузеров. На момент написания этих строк в разработке находились две интересные спецификации, основанные на идеях, похожих на те, что продвигает AngularJS.

Спецификация `Object.observe` (http://wiki.ecmascript.org/doku.php?id=harmony:observe) ставит целью внедрение в браузеры меха-

низма слежения за изменениями в объектах JavaScript. При возбуждении событий обновления пользовательского интерфейса AngularJS опирается на простое сравнение состояний объектов (dirty checking). Если в браузерах появится встроенный механизм определения изменений в модели, это поможет значительно увеличить производительность многих MVC-фреймворков на JavaScript, включая и AngularJS. В действительности разработчики AngularJS провели некоторые эксперименты, опираясь на спецификацию `Object.observe`, и пришли к выводу, что в случае ее реализации прирост производительности будет составлять от 20 до 30 процентов.

Спецификация веб-компонентов (http://dvcs.w3.org/hg/webcomponents/raw-file/tip/explainer/index.html)[4] определяет виджеты с богатыми визуальными возможностями (недоступными при использовании только CSS), легкие в разработке и пригодные для повторного использования (что невозможно с применением современных библиотек на JavaScript).

Цель достаточно сложная в достижении, но опыт применения директив AngularJS показывает, что вполне возможно реализовать независимые виджеты, пригодные для многократного использования.

AngularJS – не только инновационный фреймворк по современным меркам, но и способный влиять на индустрию веб-разработки завтрашнего дня. Команда проекта AngularJS тесно сотрудничает с авторами упомянутых спецификаций, поэтому есть шанс, что многие идеи, продвигаемые проектом AngularJS, в конечном итоге превратятся во внутренние механизмы браузеров! Мы смело можем рассчитывать, что время, потраченное на изучение AngularJS и эксперименты с ним, с лихвой окупится в будущем.

# В заключение

Мы немало нового узнали в этой главе. Сначала мы познакомились с проектом AngularJS и людьми, стоящими за ним. Мы узнали, где найти библиотеку и документацию к ней, и написали наше первое приложение «Hello World». Было очень приятно узнать, что AngularJS прост в изучении и использовании.

Однако большая часть этой главы была посвящена заложению основ для оставшейся части книги. В ней было показано, как работать с контроллерами, контекстами и представлениями, и как все эти элементы взаимодействуют между собой. Немалая часть этой главы

4 http://habrahabr.ru/post/152001/ – *Прим. перев.*

была посвящена приемам создания служб в модулях и связыванию их с применением механизма внедрения зависимостей.

В заключение мы получили возможность сравнить AngularJS с другими фреймворками JavaScript и узнали, что делает его особенным. Надеемся, теперь вы уверены, что время на изучение AngularJS не будет потрачено впустую.

Представления, контроллеры и службы – это основные компоненты любого приложения на основе AngularJS, поэтому совершенно необходимо было поближе познакомиться с ними. Теперь мы знаем, как создавать службы и представления и готовы заняться настоящими проектами. В следующей главе мы заложим каркас нетривиального приложения, начав с организации кода, и затем рассмотрим такие темы, как сборка и тестирование.

# ГЛАВА 2.
# Сборка и тестирование

Предыдущая глава служила введением в AngularJS и в ней мы познакомились с самим фреймворком, проектом, людьми, участвующими в этом проекте, и основными примерами использования. Теперь мы готовы перейти к созданию законченного, более сложного веб-приложения. В оставшейся части этой книги мы будем рассматривать пример проекта, демонстрирующий процесс конструирования приложений на основе AngularJS.

В следующих главах мы создадим упрощенную версию инструмента управления проектами с поддержкой методологии SCRUM гибкой разработки приложений. Этот пример приложения поможет продемонстрировать прикладной интерфейс фреймворка AngularJS и его идиомы, а также охватить типичные ситуации, такие как взаимодействие с серверными компонентами приложений, организация навигации, обеспечение безопасности, интернационализация и так далее. В этой главе мы познакомимся с самим приложением в общем, областью его применения и используемым стеком технологий.

Любой проект начинается с принятия решений, касающихся стратегии организации файлов, выбора системы сборки и организации рабочего процесса. Наш пример приложения не является исключением, и в этой главе мы обсудим темы, связанные с выбором системы сборки и принципов организации структуры проекта.

Автоматическое тестирование – давно устоявшаяся практика, продвигаемая проектом AngularJS и всей экосистемой, окружающей его. Мы твердо уверены, что автоматическое тестирование является обязательной составляющей любого нетривиального проекта. Именно по этой причине последняя часть данной главы будет полностью посвящена тестированию: различным типам тестирования, его организации, а также применяемым приемам и инструментам.

В этой главе мы познакомимся:

- с примером приложения, которое будет использоваться на протяжении всей книги, его областью применения и используемым стеком технологий;
- с рекомендованной системой сборки веб-приложений на основе AngularJS, а также с сопутствующими ей инструментами и рекомендациями по организации рабочего процесса;
- с предложениями по организации файлов, папок и модулей;
- с приемами автоматического тестирования, разными типами тестирования и их местом в проекте; вашему вниманию будут представлены библиотеки и инструменты тестирования, обычно используемые в проектах веб-приложений на основе AngularJS.

# Введение в пример приложения

В этом разделе обсуждаются некоторые детали примера приложения, которое будет служить демонстрационным целям на протяжении всей книги.

Исходный код примера свободно доступен в git-репозитории на сайте GitHub, по адресу: https://github.com/angular-app/angular-app. Этот репозиторий содержит полный исходный код, подробные инструкции по установке и описание хронологии разработки проекта.

## *Область применения*

Чтобы представить AngularJS в наиболее выгодном свете, мы создадим инструмент управления проектами для поддержки коллективов разработчиков, использующих методологию гибкой разработки SCRUM.

Преимущества фреймворка AngularJS проявляются особенно ярко, когда он используется для создания CRUD-подобных приложений – приложений, состоящих из множества экранов, с динамическими формами, списками и таблицами.

SCRUM – популярная методология гибкой разработки программного обеспечения и мы надеемся, что многие читатели уже знакомы с ней. Те же, кто слышит о ней впервые, не должны беспокоиться, по-

тому что основные положения этой методологии просты для понимания. Существует множество замечательных книг и статей, подробно описывающих методологию SCRUM, но, чтобы получить базовые сведения о ней, достаточно простого знакомства со статьей в Википедии: http://ru.wikipedia.org/wiki/Scrum.

Цель нашего примера – помочь коллективам разработчиков в управлении артефактами SCRUM: проектами и их резервами, спринтами и их резервами, задачами, диаграммами и так далее. В состав приложения также входит полнофункциональный административный модуль для управления пользователями и проектами, над которыми они работают.

Наш пример приложения не претендует на полноту и точное соблюдение всех принципов методологии SCRUM. При необходимости мы будем сокращать реализацию некоторых особенностей, чтобы нагляднее показать особенности использования AngularJS, пусть и в ущерб функциональности инструмента управления проектами.

По окончании интерфейс приложения будет выглядеть как показано на рис. 2.1.

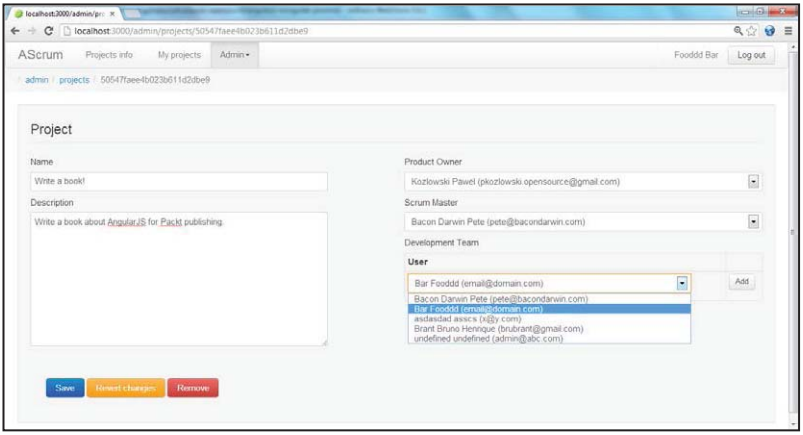


**Рис. 2.1.** Интерфейс инструмента управления проектами

Взглянув на рис. 2.1. можно сразу заметить, что нам предстоит создать самое типичное CRUD-подобное веб-приложение, которое:

- извлекает, отображает и позволяет редактировать данные, хранящиеся в некотором хранилище;
- поддерживает аутентификацию и авторизацию;

- имеет достаточно сложную схему навигации со всеми характерными элементами пользовательского интерфейса, такими как меню, навигационные цепочки в виде последовательности ссылок и так далее.

## Стек технологий

Основной целью этой книги является представление возможностей фреймворка AngularJS, но для создания нетривиального веб-приложения требуется нечто большее, чем простая библиотека JavaScript, выполняющаяся в браузере. Как минимум необходимо еще некоторое хранилище для данных и серверный программный компонент. Эта книга не ограничивает выбор того или иного стека, и мы понимаем, что наши читатели будут использовать AngularJS в сочетании с самыми разными серверными технологиями – фреймворками и хранилищами данных. Однако, чтобы двигаться дальше, нам придется остановить свой выбор на чем-то конкретном.

Прежде всего нам хотелось бы выбрать технологии, максимально дружественные по отношению к JavaScript. Существует множество замечательных серверных технологий и хранилищ данных, но мы хотели бы сделать наши примеры наиболее простыми для разработчиков, использующих JavaScript. Кроме того, мы хотели использовать технологии, считающиеся основными в экосистеме JavaScript.

Все элементы стека и инструменты, описываемые в этой книге, основаны на использовании языка JavaScript и являются свободно распространяемыми проектами с открытым исходным кодом.

## Хранилище данных

В настоящее время существует множество вариантов организации хранения данных, а последние разработки баз данных типа NoSQL еще больше увеличили количество альтернатив. Для наших целей мы будем использовать документо-ориентированную базу данных MongoDB, как более полно соответствующую окружению JavaScript:

- документы хранятся в формате, подобном JSON (Binary JSON – BSON);
- запросы и операции с данными могут быть реализованы на языке JavaScript и с применением синтаксиса JSON;

- есть возможность организовать экспортирование и импортирование данных в формате JSON через конечные точки REST.

Чтобы следовать за примерами в книге от вас не требуется предварительного знакомства с MongoDB, а при необходимости мы будем пояснять некоторые тонкости во время обсуждения фрагментов кода. Мы ни в коем случае не утверждаем, что MongoDB или документоориентированные базы данных типа NoSQL лучше всего подходят для приложений на основе AngularJS. Фреймворк AngularJS никак не зависит от выбора серверной технологии или типа хранилища данных.

### MongoLab

MongoDB – относительно простая база данных и большинство разработчиков на JavaScript будут чувствовать себя как дома при работе с ней. Чтобы упростить наш пример еще больше, мы будем пользоваться аутсорсинговой версией MongoDB. Это избавит нас от необходимости устанавливать какое-либо программное обеспечение для опробования примеров из этой книги.

Существует множество облачных служб, предоставляющих услуги доступа к базе данных MongoDB, но мы выбрали MongoLab (https://mongolab.com), простую и надежную службу, позволяющую бесплатно размещать базы данных объемом до 0,5 Гбайт. Объема хранилища, бесплатно предлагаемого компанией MongoLab, более чем достаточно для нужд примеров в этой книге.

Кроме того, компания MongoLab имеет одно важное преимущество; она предоставляет доступ к базам данных через хорошо продуманный интерфейс REST. На основе этого интерфейса мы покажем, как фреймворк AngularJS может взаимодействовать с конечными точками REST.

Для работы с MongoLab требуется обязательная регистрация, но это достаточно простой процесс, который сводится к заполнению короткой электронной формы. Пройдя его вы получите доступ к базе данных MongoDB, интерфейсу REST, поддерживающему обмен данными в формате JSON, и консоли администратора.

### Серверное окружение

Базы данных, расположенные на серверах компании MongoLab, доступны приложениям на основе AngularJS непосредственно через интерфейс REST. Прямое взаимодействие с базой данных из браузера может быть и неплохой выбор для очень простых проектов; но такой

подход совершенно не годится для настоящих приложений, находящихся в общественном доступе. Предложение MongoLab просто не обладает достаточной безопасностью.

На практике большинство пользовательских интерфейсов, реализованных с применением AngularJS, взаимодействуют с некоторым серверным компонентом, извлекающим данные из хранилища. Этот промежуточный компонент обычно реализует службы обеспечения безопасности (аутентификацию) и осуществляет проверку прав доступа (авторизацию). Наш пример приложения также нуждается в таком серверном компоненте. И снова мы собираемся отдать предпочтение решению на основе JavaScript – `node.js`.

Скомпилированные версии `node.js` доступны для всех популярных операционных систем, и могут быть получены по адресу: http://nodejs.org/. Вам необходимо загрузить и установить среду выполнения `node.js`, чтобы получить возможность опробовать примеры из этой книги на своем локальном компьютере.

Знакомство с node.js далеко выходит за рамки этой книги. К счастью, для опробования примеров из этой книги достаточно лишь общего знакомства с node.js и его диспетчером пакетов (node.js package manager – npm). Разработчики, знакомые с node.js, легко разберутся с тем, что происходит в серверной части нашего примера приложения, но для изучения особенностей фреймворка AngularJS совсем необязательно знать и понимать, как работает node.js.

Кроме самой среды выполнения node.js мы будем использовать следующие библиотеки для сборки серверных компонентов примера приложения:

- Express (http://expressjs.com/) – фреймворк серверных веб-приложений, обеспечивающий маршрутизацию, обслуживание данных и статических ресурсов;
- Passport (http://passportjs.org/) – промежуточный компонент поддержки безопасности для `node.js`;
- Restler (https://github.com/danwrong/restler) – клиентская библиотека поддержки протокола HTTP для `node.js`.

Хотя знакомство с `node.js` и упомянутыми библиотеками может быть полезным, тем не менее, оно не требуется для эффективного изучения AngularJS. Однако, если вы захотите вникнуть в суть примеров серверных компонентов приложения, подготовленных нами для этой книги, вам может потребоваться поближе познакомиться с `node.js` и с библиотеками, перечисленными выше. Если же вы в своей практике

используете другие серверные технологии, вы можете просто пропустить описание подробностей, касающихся использования `node.js`.

### Сторонние библиотеки JavaScript

AngularJS можно использовать как единственную библиотеку, даже для создания очень сложных приложений. В то же время разработчики не стремились повторно воплощать то, что уже реализовано в других популярных библиотеках.

В примере приложения мы постарались свести к минимуму зависимости от сторонних библиотек, чтобы сосредоточиться на возможностях, предлагаемых фреймворком AngularJS. Тем не менее, многие проекты используют широко известные библиотеки, такие как jQuery, underscore.js и другие.

Чтобы доказать, что AngularJS может сосуществовать с другими библиотеками, в проект была включена последняя совместимая версия jQuery.

### Bootstrap CSS

AngularJS не предъявляет каких-либо специфических требований к использованию таблиц стилей CSS, поэтому каждый может определять свои стили для своих приложений. Чтобы придать нашему приложению SCRUM более привлекательный вид, мы будем использовать популярную библиотеку стилей Twitter Bootstrap CSS (http://twitter.github.com/bootstrap/).

Пример приложения будет подключать шаблоны LESS из библиотеки Bootstrap, чтобы показать, как включить комплект LESS в цепочку сборки.

# Система сборки

Времена, когда JavaScript был игрушечным языком, давно прошли. В последние годы JavaScript превратился в один из основных языков программирования. Каждый день пишутся и развертываются новые, большие и сложные приложения на этом языке. Возрастающая сложность приложений практически всегда означает увеличение объема кода. Приложения, состоящие из десятков тысяч строк кода на JavaScript, в наши дни стали обычным делом.

Уже давно стало неудобно хранить весь код на JavaScript в одном файле, который просто подключается к HTML-документу, поэтому нам нужна система сборки. Наши файлы с кодом на JavaScript и CSS

подвергаются большому количеству проверок и преобразований, прежде чем они будут развернуты на действующих серверах. Примерами таких проверок и преобразований могут служить.

- Проверка исходного кода на JavaScript на соответствие стандартам оформления, с использованием таких инструментов, как `jslint` (http://www.jslint.com/), `jshint` (http://www.jshint.com/) и им подобных.
- Тестирование с применением наборов тестов, которое желательно проводить как можно чаще – хотя бы в процессе выполнения каждой сборки. В силу этого инструменты и процедуры тестирования должны быть интегрированы в систему сборки.
- Создание дополнительных файлов (например, файлов CSS на основе шаблонов, таких как LESS).
- Объединение и минификация файлов для оптимизации производительности браузеров.

Помимо этапов, перечисленных выше, обычно приходится выполнять и другие операции с файлами, прежде чем приложение сможет быть развернуто на серверах. Например, скопировать файлы в каталог назначения, обновить документацию и так далее. В сложных проектах всегда приходится выполнять массу рутинных операций, которые желательно автоматизировать.

## Принципы построения систем сборки

Разработчики обожают писать код, но в действительности они лишь часть своего времени проводят в текстовом редакторе. Помимо приятного времяпрепровождения (проектирования, общения с коллегами, создания кода и исправления ошибок) они вынуждены также заниматься выполнением бесчисленного множества рутинных операций, иногда часто, иногда очень часто. Одной из таких операций является сборка системы, которую требуется выполнять снова и снова. Поэтому она должна производиться максимально быстро и с как можно меньшими трудозатратами. Далее мы обсудим некоторые принципы управления сборкой системы.

### Автоматизируйте все, что только возможно

Авторы этой книги плохо справляются с выполнением операций вручную, требующих многократного повторения. Мы слишком медлительны, часто допускаем ошибки и, честно признаться, быстро утомляемся при выполнении рутинных задач. Если складом характе-

ра вы похожи на нас, то без труда поймете, почему мы так стремимся автоматизировать каждый шаг процесса сборки.

Компьютеры отлично справляются с такими рутинными операциями; они не допускают ошибок, им не нужно прерываться, чтобы выпить чашечку кофе, и они не отвлекаются от работы. Наши компьютеры не жалуются, когда мы нагружаем их работой! Время, потраченное на автоматизацию всего, что только возможно, быстро окупается, так давайте экономить свое время!

### Ошибки должны обнаруживаться максимально быстро

Типичная процедура сборки состоит из нескольких отдельных шагов. Некоторые из них в большинстве случаев выполняются без каких-либо проблем, другие время от времени терпят неудачу. Операции, которые наиболее вероятно закончатся неудачей, должны выполняться как можно раньше и немедленно прерывать процедуру сборки при обнаружении ошибки. Это гарантирует, что другие продолжительные операции не будут запущены, пока не будут решены основные проблемы.

Нам пришлось преодолеть длинный и трудный путь, прежде чем мы пришли к этому важному выводу. Первоначально система сборки примера приложения, демонстрируемого в этой книге, автоматически выполняла тестирование перед проверкой исходного кода с помощью `jshint`. Мы часто с расстройством обнаруживали, что нам приходится ждать завершения автоматического тестирования, чтобы в итоге обнаружить, что сборка невозможна из-за наличия синтаксической ошибки. В результате мы переместили проверку с помощью `jshint` в самое начало процедуры сборки, чтобы она прерывалась как можно раньше.

Важно также включать ясные и понятные сообщения об ошибках, прерывающих процедуру сборки. Сообщения должны позволять понять причину ошибки буквально за секунды. Нет ничего хуже, чем гадать о причинах, уставившись в туманное сообщение, особенно когда поджимает время.

### Разные операции, разные команды

Как разработчикам нам приходится заниматься решением разных задач. Один день мы можем добавлять новый код (с тестами!), другой– заниматься отладкой новых особенностей. Система сборки должна соответствовать этим реалиям, предоставляя разные команды

для выполнения разных процедур. По нашему опыту система сборки должна поддерживать три отдельные процедуры сборки:

1. Команда быстрой проверки корректности кода: в проектах на JavaScript под этим может подразумеваться выполнение проверок с помощью `jslint/jshint` и модульных тестов. Эта команда должна действовать быстро, чтобы ее можно было запускать достаточно часто. Такая команда очень пригодится при разработке кода с использованием методики **разработки через тестирование** (Test Driven Development, TDD).
2. Команда развертывания готового приложения для последующего тестирования: после выполнения этой команды должна появляться возможность запустить приложение в браузере. Эта процедура требует больше времени, так как в процессе ее выполнения необходимо сгенерировать файлы CSS и выполнить еще целый ряд дополнительных операций. Эта команда предназначена для нужд тестирования пользовательского интерфейса.
3. Команда подготовки готового приложения к развертыванию в эксплуатационном окружении должна производить все проверки, перечисленные выше, а также подготавливать приложение к окончательному развертыванию, выполняя слияние и минификацию файлов, производя интеграционное тестирование, и так далее.

### Сценарии сборки тоже программы

Сценарии сборки являются составной частью артефактов проекта, и мы должны писать их с такой же заботой, как и любой другой продукт. Сценарии сборки будут читаться и сопровождаться, подобно любому другому исходному коду. Плохо написанный код сценариев сборки может существенно замедлить работу и повлечь необходимость потратить дополнительное время на его отладку.

## *Инструменты*

Принципы построения систем сборки, обозначенные в предыдущем разделе, являются общими для любых проектов и инструментов. Мы остановили свой выбор на платформонезависимых инструментах, чтобы вы могли опробовать примеры в любой популярной операционной системе.

Для примера приложения мы выбрали лучшие, по нашему мнению, инструменты сборки. Но мы понимаем, что в различных проектах будут использоваться разные инструменты, поэтому в этой и в следующих главах мы будем давать рекомендации, которые легко можно будет распространить на другие системы сборки.

### Grunt.js

Система сборки примера приложения SCRUM основана на инструменте Grunt (http://gruntjs.com/), который на сайте проекта позиционируется как:

> «*Инструмент командной строки для сборки проектов на JavaScript с применением заданий.*»

Мы считаем важным то обстоятельство, что сценарии сборки для `grunt.js` пишутся на JavaScript и выполняются под управлением платформы `node.js`. Для нас это означает, что мы можем использовать единую платформу и язык программирования для запуска приложения и его сборки.

Grunt.js принадлежит к той же категории инструментов сборки, что и Gradle (для сборки приложений на языке Groovy) или Rake (для сборки приложений на языке Ruby), поэтому те из вас, кто знаком с этими инструментами, будут чувствовать себя вполне уверенно.

### Библиотеки и инструменты тестирования

Фреймворк AngularJS способствует использованию приемов автоматического тестирования. Разработчики AngularJS очень серьезно относятся к тестированию и гарантируют простоту тестирования кода, использующего AngularJS. Но этим возможности тестирования не исчерпываются, так как команда проекта AngularJS постоянно пишет новые или расширяет существующие инструменты с целью упростить практику тестирования.

### Jasmine

Тесты для фреймворка AngularJS были написаны с использованием **Jasmine** (http://pivotal.github.com/jasmine/). Jasmine – это фреймворк для тестирования кода на JavaScript и своими корнями уходит в методику **разработки через реализацию поведения** (Behavior Driven Development, BDD), оказавшей существенное влияние на его синтаксис.

Все примеры в оригинальной документации к AngularJS написаны с использованием синтаксиса Jasmine, поэтому выбор данного инструмента для тестирования нашего примера приложения выглядит вполне естественным. Кроме того, разработчиками AngularJS было написано множество фиктивных (mock) объектов и расширений для Jasmine, чтобы обеспечить бесшовную интеграцию с ним и упростить его использование для решения повседневных задач.

### Karma runner

**Karma runner** (http://karma-runner.github.io) – это инструмент, упрощающий выполнение тестов для кода на JavaScript. Утилита karma runner появилась как замена другому популярному инструменту запуска тестов – JS TestDriver.

Эта утилита способна передавать исходный код вместе с кодом тестов запущенному экземпляру браузера (или запустить новый, если потребуется!), выполнить тесты, собрать результаты тестирования и составить отчет на их основе. Для тестирования она использует настоящие браузеры. Это огромное достижение в мире JavaScript, благодаря которому мы можем выполнить тестирование кода сразу в нескольких браузерах и убедиться, что он действует правильно в дикой природе.

Утилита Karma runner – обладает удивительными характеристиками, в смысле стабильности и производительности. Она используется в проекте AngularJS для выполнения тестов в рамках непрерывной интеграционной сборки. В каждой операции сборки Karma runner выполняет порядка 2000 модульных тестов во множестве браузеров. Всего приблизительно 14,000 тестов выполняются в течение примерно 20 секунд. Эти цифры должны упрочить вашу уверенность в Karma runner, как инструменте, и AngularJS, как фреймворке.

Тестирование занимает настолько важное положение в разработке AngularJS, что мы более подробно рассмотрим вопросы создания тестов на основе Jasmine и использования утилиты Karma runner.

## Организация файлов и каталогов

К настоящему моменту мы приняли ряд важных решений, касающихся используемого стека технологий и инструментов. Теперь нам нужно решить еще один немаловажный вопрос; как организовать файлы и папки проекта.

Существует множество способов организации файлов в проектах. Иногда выбор делается без нашего участия, так как некоторые инструменты и фреймворки используют предопределенные структуры каталогов. Но ни `grunt.js`, ни AngularJS не накладывают никаких ограничений на организацию каталогов, что дает нам свободу выбора. В следующих подразделах мы рассмотрим один из вариантов организации файлов проекта и его логическое обоснование. Вы можете взять эту структуру на вооружение для использования в своих будущих проектах или внести свои коррективы, в соответствии со своими потребностями.

## Каталоги верхнего уровня

Проектируя структуру каталогов проекта, следует помнить о простоте навигации по исходному коду, и в то же время стараться удержать ее сложность в разумных пределах.

Ниже перечислено несколько основных правил, используемых при проектировании высокоуровневой структуры каталогов.

- Исходный код приложения и сопроводительных тестов должен храниться в разных каталогах. Это упростит сопровождение системы сборки, потому что тестирование и сборка исходного кода обычно выполняется различными комплексами задач.
- Код любых сторонних библиотек должен храниться отдельно от внутреннего кода проекта. Сторонние библиотеки будут развиваться в ином темпе, нежели наш продукт. Поэтому мы должны предусмотреть возможность обновления внешних зависимостей в любой момент. Смешивание внутреннего исходного кода с внешними библиотеками может сделать такое обновление сложным мероприятием, отнимающим много времени.
- Результаты выполнения сценариев сборки должны сохраняться в специально выделенный каталог, а не в каталог с исходным кодом. Состав и содержимое файлов, получающихся в результате сборки должны в точности соответствовать требованиям среды развертывания приложения. Результаты сборки должны сохраняться так, чтобы их легко можно было извлечь и развернуть в конечном местоположении.

Применив правила, перечисленные выше, мы получили следующую структуру каталогов проекта:

- `src`: хранит исходный код приложения;
- `test`: хранит код автоматически выполняемых тестов;
- `vendor`: хранит сторонние зависимости;
- `build`: хранит сценарии сборки;
- `dist`: хранит результаты сборки, готовые к развертыванию в целевой среде.

На рис. 2.2 показано, как выглядит та же структура каталогов в окне диспетчера файлов.

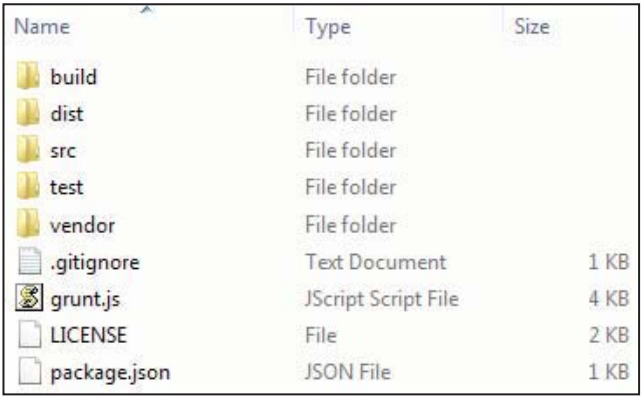


**Рис. 2.2.** Структура каталогов проекта в окне диспетчера файлов

В дополнение к описанным каталогам на рис. 2.2 можно заметить несколько файлов в корневом каталоге проекта, краткое описание которых приводится ниже:

- `.gitignore`: включает правила, определяющие, какие файлы не должны сохраняться в репозитории git;
- `LICENSE`: содержит положения лицензии MIT;
- `Gruntfile.js`: точка входа в систему сборки `grunt.js`;
- `package.json`: содержит описание пакета для приложений `node.js`.

## Каталог с исходным кодом

Теперь мы готовы углубиться в структуру каталога `src`. Но для начала взгляните на рис. 2.3, где изображено содержимое этого каталога.

Файл `index.html` – это точка входа в приложение. Здесь также присутствует четыре каталога, в двух из которых хранится код, использующий фреймворк AngularJS (`app` и `common`), и в двух – код артефактов, независимых от AngularJS (`assets` и `less`).

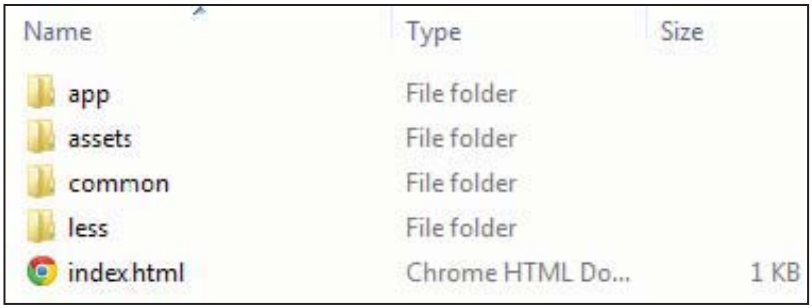


**Рис. 2.3.** Содержимое каталога `src`

Назначение каталогов `assets` и `less` очевидно – первый служит для хранения изображений и ярлыков, а второй – различных переменных шаблона LESS. Обратите внимание, что шаблоны LESS для Twitter Bootstrap CSS хранятся в каталоге `vendor`, а в этом каталоге хранятся только значения для переменных.

### Файлы, использующие фреймворк AngularJS

Приложения на основе AngularJS состоят из файлов двух разных типов, а именно, сценариев и шаблонов в разметкой HTML. Любой нетривиальный проект будет содержать множество файлов обоих типов и нам следует найти наиболее оптимальный способ организации этой массы файлов. В идеале было бы желательно хранить взаимосвязанные файлы вместе, а независимые – отдельно. Проблема в том, что файлы могут быть связаны друг с другом самыми разными отношениями, а у нас имеется только одно дерево каталогов, чтобы выразить эти отношения.

Типичные стратегии решения этой проблемы заключаются в группировке файлов по их назначению, по архитектуре или по типу. Мы хотели бы предложить следующее гибридное решение.

- Большинство файлов приложения будет группироваться по их назначению. Сценарии и другие файлы с родственной функциональностью должны храниться вместе. Такая организация упрощает работу с вертикальными срезами приложения, так как все файлы, которые должны изменяться одновременно, хранятся вместе.
- Файлы, содержащие реализации сквозных задач (доступ к хранилищу данных, локализация, общие директивы и так далее) должны храниться вместе. Такая организация объясняется тем,

что подобные инфраструктурные сценарии изменяются не так часто, как код, реализующий функциональность приложения. В типичных приложениях техническая инфраструктура создается на самых ранних этапах разработки и только потом фокус смещается в сторону функциональности приложения. Файлы в каталоге `common`, образующие инфраструктурный слой, лучше организовать по архитектурному признаку.

Рекомендации, перечисленные выше, можно транслировать непосредственно в структуру каталогов, изображенную на рис. 2.4.

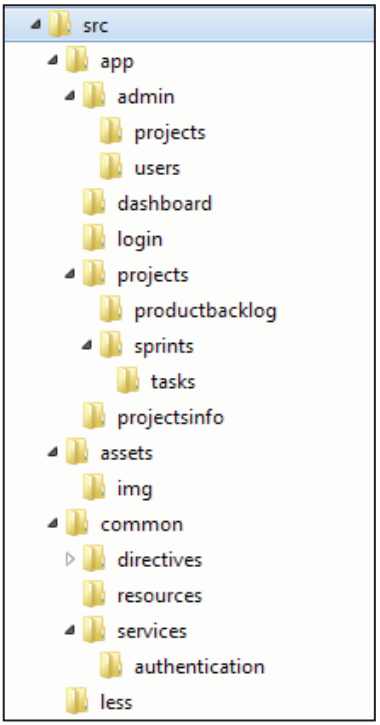


**Рис. 2.4.** Структура дерева каталогов с исходным кодом

Структура каталогов, описанная здесь, отличается от рекомендованной проектом AngularJS seed (https://github.com/angular/angular-seed). Структура каталогов, предлагаемая проектом angular-seed, отлично подходит для простых проектов, но по общему мнению членов сообщества AngularJS, структуру каталогов в больших проектах лучше организовать по назначению файлов.

### Начните с самого простого

Взглянув на структуру каталогов, можно заметить, что некоторые каталоги, хранящие функциональный код, имеют глубокую иерархию. Эти каталоги довольно точно отражают иерархию навигации в самом приложении, что вполне оправданно, так как взглянув на пользовательский интерфейс, мы легко сможем определить, где хранится соответствующий код.

Всегда начинайте проект с создания простой структуры каталогов, а затем мелкими шагами двигайтесь к окончательной организации. Так каталог `admin`, в примере приложения, первоначально не содержал подкаталогов, и вся функциональность управления проектами и пользователями хранилась в одном общем каталоге. В процессе разработки, по мере увеличения размеров файлов (и их количества), были добавлены новые подкаталоги. Структура каталогов может изменяться и расширяться, в точности как исходный код.

### Храните контроллеры и шаблоны вместе

Очень часто организация файлов в проектах выполняется по их типам. В контексте AngularJS это означает, что сценарии на JavaScript и шаблоны часто помещаются в разные каталоги. На первый взгляд такое разделение выглядит оправданным, но на практике шаблоны и соответствующие им контроллеры разрабатываются одновременно. Именно поэтому в примере приложения SCRUM шаблоны и контроллеры хранятся вместе. Для каждой функциональной области создан свой каталог, где вместе хранятся шаблоны и контроллеры.

Например, на рис. 2.5 показано содержимое каталога, где хранится все, что имеет отношение к функции администрирования пользователей.

| Name | Date modified | Type | Size |
|---|---|---|---|
| admin-users.js | 2012-11-20 17:59 | JScript Script File | 1 KB |
| admin-users-edit.js | 2012-11-27 21:25 | JScript Script File | 3 KB |
| users-edit.tpl.html | 2012-11-26 19:29 | Chrome HTML Document | 2 KB |
| users-list.tpl.html | 2012-11-20 17:59 | Chrome HTML Document | 1 KB |

**Рис. 2.5.** Файлы, связанные с реализацией функции администрирования пользователей

### Каталог с тестами

Автоматизированные тесты создаются с целью убедиться, что приложение функционирует правильно, а код тестов тесно связан с

прикладным кодом. Поэтому вас не должно удивлять наше решение создать в каталоге `test` структуру, повторяющую структуру каталога `src/app`, как показано на рис. 2.6.

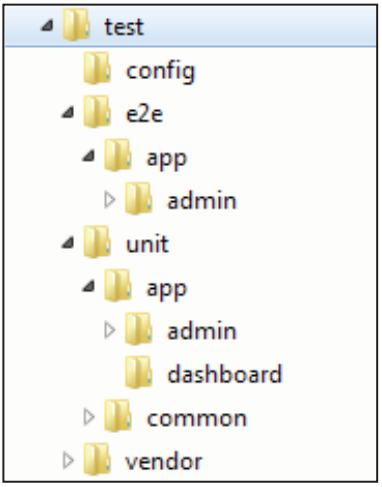


**Рис. 2.6.** Структура каталога `test`

Нетрудно заметить, что содержимое каталога `test` практически в точности отражает корневой каталог с исходным кодом. Все тестовые библиотеки помещаются в выделенный для них каталог `vendor`, а настройки утилиты Karma runner – в собственный отдельный каталог (`config`).

## Соглашения по именованию файлов

Очень важно установить некоторые соглашения по именованию файлов, чтобы упростить их поиск в каталоге с исходным кодом. Ниже приводится несколько соглашений, которым часто следуют в сообществе AngularJS, и которых будем придерживаться в этой книге:

- все файлы с кодом на JavaScript должны иметь стандартное расширение `.js`;
- шаблоны должны иметь расширение `.tpl.html`, чтобы их проще было отличать от других HTML-файлов;
- файлы с тестами должны иметь имена, совпадающие с именами тестируемых ими файлов, а расширения – в зависимости от типа теста, например, модульные тесты должны иметь расширение `.spec.js`.

# Модули и файлы AngularJS

Теперь, когда организация приложения по файлам и каталогам закончена, можно приступать к знакомству с содержимым отдельных файлов. Далее основное внимание будет уделяться файлам с кодом на JavaScript, их содержимому и отношениям с модулями AngularJS.

## *Один файл, один модуль*

В главе 1, «Дзен Angular», мы узнали, что модули AngularJS могут зависеть друг от друга. Это означает, что в приложениях на основе AngularJS нам придется иметь дело не только с иерархиями каталогов, но и с иерархиями модулей. Давайте рассмотрим этот вопрос поглубже, чтобы выработать практические рекомендации по организации модулей.

Существует три основных подхода к организации взаимосвязи между файлами и модулями:

- один файл JavaScript может содержать несколько модулей;
- один модуль может быть разбит на несколько файлов JavaScript;
- в каждом файле JavaScript определяется точно один модуль;

Нет ничего плохого, чтобы поместить несколько модулей в один файл, однако в результате могут получаться файлы с сотнями строк кода. При такой организации сложнее будет отыскать нужный модуль, так как потребуется найти не только файл с модулем, но и сам модуль внутри файла. Такой прием размещения нескольких модулей в одном файле может быть и неплохо подходит для простых проектов, но он никуда не годится для больших объемов кода.

Ситуации, когда один модуль разбит на несколько файлов, вообще лучше избегать, так как в этом случае нам необходимо будет помнить об определенном порядке загрузки этих файлов: объявление модуля должно загружаться перед функциями регистрации. Кроме того, такие модули сложнее в сопровождении. Большие модули особенно нежелательны с точки зрения модульного тестирования, когда требуется загружать и тестировать код как можно меньшими фрагментами.

Их этих трех вариантов наиболее разумным выглядит только вариант, когда один файл содержит только один модуль.

Старайтесь придерживаться принципа «один файл, один модуль». Это даст вам возможность организовать код в небольшие файлы и модули с узкоспециализированной функциональностью, избавит от необходимости запоминать порядок загрузки файлов и позволит загружать лишь отдельные модули на этапе модульного тестирования.

## Внутри модуля

Как только модуль будет объявлен, его можно сразу же использовать для регистрации рецептов создания объектов. Напомним, что в терминологии AngularJS рецепты называют провайдерами. Каждый провайдер создает точно один экземпляр службы. Хотя рецепты создания служб можно выразить множеством способов (фабрики, службы, провайдеры и переменные), конечный результат всегда один и тот же – настроенный экземпляр службы.

### Разные способы регистрации провайдеров

Чтобы зарегистрировать нового провайдера, необходимо сначала получить экземпляр модуля. Это несложно, так как экземпляр модуля всегда можно получить вызовом метода `angular.module`.

Ссылку на экземпляр модуля, возвращаемую методом, можно сохранить в переменной и повторно использовать для регистрации нескольких провайдеров. Например, регистрацию двух контроллеров в модуле, ответственном за управление проектами, можно реализовать так:

```
var adminProjects = angular.module('admin-projects', []);

adminProjects.controller('ProjectsListCtrl', function($scope) {
    // здесь находится реализация контроллера
});

adminProjects.controller('ProjectsEditCtrl', function($scope) {
    // здесь находится реализация контроллера
});
```

Этот способ действует, как ожидается, однако он имеет один недостаток; мы вынуждены объявлять временную переменную (в данном случае `adminProjects`), только чтобы получить возможность зарегистрировать несколько провайдеров в одном модуле. Самое неприятное, что такая временная переменная может оказаться в глобальном пространстве имен, если не предпринять дополнительных мер предос-

торожности (например, завернуть объявление модуля в замыкание, создать пространство имен и так далее). В идеале было бы неплохо иметь возможность каким-то образом извлекать уже объявленный модуль. Как оказывается, это вполне возможно:

```
angular.module('admin-projects', []);

angular.module('admin-projects').controller('ProjectsListCtrl',
function($scope) {
    // здесь находится реализация контроллера
});

angular.module('admin-projects').controller('ProjectsEditCtrl',
function($scope) {
    // здесь находится реализация контроллера
});
```

Мы сумели избавиться от лишней переменной, но сам код от этого ничего не выиграл. Возможно вы заметили что вызов `angular.module('admin-projects')` повторяется несколько раз. Любое дублирование кода неприемлемо, и мы будем наказаны за это, если позднее решим переименовать модуль. Кроме того, сходство синтаксиса объявления нового модуля и получения ссылки на существующий легко может привести к путанице. Просто сравните `angular.module('myModule')` и `angular.module('myModule', [])`. Различие между ними легко не заметить, разве не так?

Старайтесь избегать извлечения модулей с использованием конструкции `angular.module('myModule')`. Такой подход приводит к появлению повторений в коде. Хуже того, объявления модулей легко спутать с обращениями к существующим модулям.

К счастью существует еще один способ, решающий проблемы, упомянутые выше. Взгляните сначала на следующий фрагмент:

```
angular.module('admin-projects', [])
    .controller('ProjectsListCtrl', function($scope) {
        // здесь находится реализация контроллера
    })

    .controller('ProjectsEditCtrl', function($scope) {
        // здесь находится реализация контроллера
    });
```

Как показывает этот пример, существует возможность объединять в цепочки вызовы методов регистрации контроллеров. Каждый вы-

зов метода `controller` возвращает экземпляр модуля, относительно которого был вызван метод. Другие методы регистрации провайдеров (`factory`, `service`, `alue` и так далее) тоже возвращают экземпляр модуля, поэтому мы можем регистрировать провайдеров разных типов, используя один и тот же прием.

В примере приложения SCRUM мы будем использовать только что описанный синтаксис цепочек вызовов для регистрации всех провайдеров. Этот прием избавляет от необходимости создавать лишние переменные (возможно глобальные!) и писать повторяющийся код. Кроме того, он улучшает читаемость кода, если конечно добавить некоторое форматирование.

### Синтаксис объявления блоков настройки и выполнения

Как рассказывалось в главе 1, процесс инициализации приложения на основе AngularJS делится на две фазы: фазу настройки и фазу выполнения. Каждый модуль может иметь несколько блоков настройки и выполнения. То есть мы не ограничены единственным блоком.

Как оказывается, AngularJS поддерживает два разных способа регистрации функций, которые должны вызываться в фазе настройки. Мы уже знаем, что функцию настройки можно передать методу `angular.module` в третьем аргументе:

```
angular.module('admin-projects', [], function() {
    // здесь находится логика настройки
});
```

Предыдущий способ позволяет зарегистрировать один, и только один блок настройки. К тому же такое определение модуля слишком многословно, особенно если одновременно определяются зависимости от других модулей. Альтернативное решение заключается в использовании метода `angular.config`:

```
angular.module('admin-projects', [])

    .config(function() {
        // блок настройки 1
    })

    .config(function() {
        // блок настройки 2
    });
```

Как видите, альтернативный способ позволяет зарегистрировать несколько блоков настройки. Это дает возможность разбить длинные

функции настройки (особенно с множественными зависимостями) на более короткие, специализированные функции. Короткие, специализированные функции проще читаются, сопровождаются и тестируются.

В примере приложения SCRUM мы будем использовать второй способ регистрации функций настройки и выполнения, так как считаем его более удобным.

**Загрузка исходных кодов примера.** Файлы с исходным кодом примеров для любой книги издательства Packt, приобретенной с использованием вашей учетной записи, можно загрузить на сайте http://www.packtpub.com. Если вы приобрели эту книгу каким-то иным способом, посетите страницу http://www.packtpub.com/support и зарегистрируйтесь, чтобы получить файлы непосредственно на электронную почту.

# Автоматическое тестирование

Разработка программного обеспечения – сложное дело. Как разработчики мы должны стараться воплотить требования клиентов, уложиться в установленные сроки, эффективно сотрудничать с другими членами команды и продолжать совершенствовать свои навыки владения различными инструментами и языками программирования. С ростом сложности программ растет и количество ошибок. Все мы – люди, и все мы допускаем ошибки. Программные ошибки – это лишь одна из проблем, с которыми приходится постоянно бороться. Другой разновидностью проблем является настройка и интеграция.

Так как мы легко допускаем ошибки, нам необходимы эффективные инструменты и приемы борьбы с ними. В индустрии разработки программного обеспечения уже давно пришли к выводу, что только строгое применение автоматического тестирования может гарантировать высокое качество и своевременное завершение работ.

Приемы автоматического тестирования первоначально были разработаны в рамках методологий гибкой разработки, таких как экстремальное программирование (eXtreme Programming, XP). Но то, что еще несколько лет назад считалось революционно новым (и до некоторой степени экстравагантным) теперь считается общепринятой стандартной практикой. В настоящее время никакими причинами нельзя оправдать отсутствие автоматического тестирования в любом проекте.

Первую линию обороны против программных ошибок обычно образуют модульные тесты. Эти тесты, как можно предположить из названия, предназначены для тестирования небольших блоков кода (модулей), обычно отдельных классов или групп объектов, тесно связанных между собой. Модульные тесты – это инструмент для разработчика. Они позволяют проверить правильную работу низкоуровневых конструкций и убедиться, что наши функции возвращают ожидаемые результаты. Но модульные тесты несут гораздо больше преимуществ, чем однократная проверка кода:

- они помогают обнаруживать проблемы на ранних этапах разработки, когда стоимость отладки и исправления ошибок относительно невысока;
- удачно составленный набор тестов может выполняться в процессе каждой сборки, что придает спокойствия и позволяет с уверенностью вносить изменения в программный код;
- едва ли существует более легковесная среда разработки, чем простой редактор и инструмент запуска тестов, и благодаря модульным тестам мы можем быстро проверить наши изменения, минуя этапы сборки, развертывания и опробования получившегося приложения;
- следование принципу «сначала тест», пропагандируемому методикой разработки через тестирование (Test Driven Development, TDD), помогает проектировать классы и их интерфейсы и использовать тесты как одну из разновидностей клиентов, вызывающих наш код;
- наконец тесты могут служить документацией – мы можем открыть тест и посмотреть, как следует вызывать тот или иной метод, какие значения передаются в аргументах, но самое важное, что эта документация может быть выполнена и всегда ответствует самым последним изменениям.

Модульные тесты оказывают существенную помощь, но они не способны выявить все возможные проблемы. В частности, проблемы настройки и интеграции не выявляются модульными тестами. Чтобы полностью охватить приложение автоматизированными тестами, необходимо также проводить высокоуровневое, интеграционное тестирование. Цель интеграционных тестов – запустить собранное приложение и убедиться, что все его компоненты взаимодействуют друг с другом как хорошо отлаженный механизм.

Создание и сопровождение автоматизированных тестов – это особый навык, который как и любой другой требуется получать и отта-

чивать. Сначала у вас наверняка будет складываться впечатление, что внедрение практики тестирования замедляет работу над проектом и не стоит потраченных усилий. Но по мере накопления опыта вы начнете замечать, как много времени в действительности экономит строгое следование практике тестирования кода.

Раньше существовало изречение, что создание кода без применения системы контроля версий (Version Control System, VCS) напоминает свободное падение без парашюта. Сегодня просто невозможно представить разработку без использования VCS. То же относится и к автоматическому тестированию, и мы так же можем сказать: «Создание программного обеспечения без набора автоматизированных тестов подобно восхождению без страховки или свободному падению без парашюта». Конечно, вы можете попробовать пойти наперекор, но результат наверняка будет печальным.

Разработчики AngularJS понимают важность автоматического тестирования и неуклонно следуют практике тестирования в процессе работы над фреймворком. Но, что самое замечательное – AngularJS распространяется вместе с комплектом инструментов, библиотек и рекомендаций, упрощающих тестирование наших приложений.

## Модульные тесты

Роль платформы тестирования в AngularJS играет фреймворк Jasmine: модульные тесты для самого фреймворка AngularJS также написаны с применением Jasmine, и во всех примерах в документации используется синтаксис Jasmine. Более того, AngularJS расширяет оригинальную библиотеку Jasmine дополнительными инструментами, еще больше упрощающими тестирование.

### Структура теста Jasmine

Сам фреймворк Jasmine позиционируется, как «фреймворк для тестирования кода на JavaScript, разработанный с применением методики разработки через реализацию поведения». Наследственность методики разработки через реализацию поведения проявляется в синтаксисе инструкций, которые читаются как предложения на обычном английском языке. Давайте рассмотрим пример простого теста, проверяющего стандартный класс на JavaScript, и посмотрим, как он действует:

```
describe('hello World test', function () {

    var greeter;
    beforeEach(function () {
        greeter = new Greeter();
    });

    it('should say hello to the World', function () {
        expect(greeter.say('World')).toEqual('Hello, World!');
    });
});
```

Здесь присутствует несколько конструкций, о которых следует рассказать подробнее.

- Функция `describe` описывает некоторую особенность приложения. Она действует как контейнер, куда складываются отдельные тесты. Если вы знакомы с другими фреймворками тестирования, то считайте блоки `describe` наборами тестов. Блоки `describe` могут вкладываться друг в друга.
- Реализация самого теста находится внутри функции `it`. Идея состоит в том, чтобы в одном тесте проверять только какой-то один аспект особенности. Тест имеет имя и тело. Обычно в первом разделе в теле теста вызываются методы тестируемого объекта, а в последнем производится сравнение полученных результатов с ожидаемыми.
- Код в блоке `beforeEach` будет выполняться перед каждым отдельным тестом. Это отличное место для размещения логики инициализации, которая должна выполняться перед каждым тестом.
- И, наконец, функции `expect` и `toEqual`. С помощью этих двух конструкций можно сравнить полученные результаты с ожидаемыми. Фреймворк Jasmine, как и многие популярные фреймворки тестирования, включает богатый набор инструментов сопоставления, такие как `toBeTruthy`, `toBeDefined`, `toContain` и многие другие.

### Тестирование объектов AngularJS

Тестирование объектов AngularJS не сильно отличается от тестирования любых других классов JavaScript. Особенность фреймворка AngularJS заключается в его механизме внедрения зависимостей и в способе, каким этот механизм может использоваться в модульных тестах. Чтобы узнать, как писать тесты, использующие механизм DI, мы разберем на примере тестирования служб и контроллеров.

Все расширения, имеющие отношение к тестированию, и фиктивные объекты распространяются в составе AngularJS в отдельном файле `angular-mocks.js`. Не забудьте подключить его в своем сценарии, выполняющем тестирование. Не забудьте при этом отключить данный файл при подготовке версии приложения, готовой к развертыванию.

## Тестирование служб

Тестировать объекты, регистрируемые в модулях AngularJS, совсем несложно, но требует выполнения некоторых подготовительных операций. Точнее, необходимо обеспечить инициализацию соответствующего модуля и ввод в действие всего механизма DI. К счастью в AngularJS имеется набор методов, позволяющих совместить тесты Jasmine с механизмом внедрения зависимостей.

Давайте разберем простой тест для службы `notificationsArchive`, представленной в главе 1, и посмотрим, как выполняется тестирование служб AngularJS. Чтобы напомнить, о чем идет речь, ниже повторно приводится код реализации службы:

```
angular.module('archive', [])
    .factory('notificationsArchive', function () {

    var archivedNotifications = [];
    return {
        archive:function (notification) {
            archivedNotifications.push(notification);
        },
        getArchived:function () {
            return archivedNotifications;
        }};
    });
```

А вот и соответствующий тест:

```
describe('notifications archive tests', function () {

    var notificationsArchive;
    beforeEach(module('archive'));

    beforeEach(inject(function (_notificationsArchive_) {
        notificationsArchive = _notificationsArchive_;
    }));

    it('should give access to the archived items', function () {
        var notification = {msg: 'Old message.'};
```

```
        notificationsArchive.archive(notification);

        expect(notificationsArchive.getArchived())
            .toContain(notification);
    });
});
```

Теперь вы наверняка должны опознать уже знакомую вам структуру теста Jasmine и пару новых функций: `module` и `inject`.

Функция `module` используется в тестах Jasmine, чтобы указать на необходимость подготовки к тестированию служб из данного модуля. Своей ролью этот метод напоминает директиву `ng-app`. Он сообщает, что для данного модуля (и всех его зависимостей) должен быть создан объект `$injector`.

Не путайте функцию `module` в тестах с методом `angular.module`. Несмотря на сходство имен, их роли в корне отличаются. Метод `angular.module` используется для объявления новых модулей, а функция `module` указывает модули, используемые в тестах.

В действительности в одном тесте можно много раз вызывать функцию `module`. В этом случае через экземпляр `$injector` будут доступны все службы, значения и константы из указанных модулей.

Функция `inject` выполняет одну простую операцию – она внедряет службы в тесты.

Последнее, что может смутить вас – это странные символы подчеркивания в вызове функции `inject`:

```
var notificationsArchive;
beforeEach(inject(function (_notificationsArchive_) {
    notificationsArchive = _notificationsArchive_;
}));
```

Дело в том, что объект `$injector` автоматически выбрасывает парные символы подчеркивания в начале и в конце, когда анализирует аргументы функции, чтобы извлечь зависимости. Эта особенность может пригодиться, например, чтобы сохранить имена переменных без подчеркиваний для самого теста.

### Тестирование контроллеров

Тест для контроллера имеет аналогичную структуру, что и тест для службы. Рассмотрим фрагмент контроллера `ProjectsEditCtrl` из примера приложения. Этот контроллер отвечает за редактирование информации о проекте в административном разделе приложения.

Здесь мы протестируем методы контроллера, добавляющие и удаляющие членов команды проекта:

```
angular.module('admin-projects', [])
    .controller('ProjectsEditCtrl', function($scope, project) {

        $scope.project = project;

        $scope.removeTeamMember = function(teamMember) {
            var idx = $scope.project.teamMembers.indexOf(teamMember);
            if(idx >= 0) {
                $scope.project.teamMembers.splice(idx, 1);
            }
        };

        // другие методы контроллера
    });
```

Представленный контроллер имеет очень простую логику, что позволит нам полностью сосредоточиться на самом тесте:

```
describe('ProjectsEditCtrl tests', function () {

    var $scope;
    beforeEach(module('admin-projects'));
    beforeEach(inject(function ($rootScope) {
        $scope = $rootScope.$new();
    }));

    it('should remove an existing team member', inject(function
($controller) {

        var teamMember = {};
        $controller('ProjectsEditCtrl', {
            $scope: $scope,
            project: {
                teamMembers: [teamMember]
            }
        });

        // проверить начальные настройки
        expect($scope.project.teamMembers).toEqual([teamMember]);

        // выполнить и проверить результат
        $scope.removeTeamMember(teamMember);
        expect($scope.project.teamMembers).toEqual([]);
    }));
});
```

Тестируемый метод `removeTeamMember` будет определен в объекте `$scope`, который является экземпляром контролле-

ра `ProjectsEditCtrl`. Чтобы эффективно проверить метод `removeTeamMember`, необходимо создать новый контекст, новый экземпляр контроллера `ProjectsEditCtrl` и связать их вместе. По сути нам нужно вручную проделать то, что делает директива `ng-controller`.

Теперь на минутку обратим наше внимание на раздел `beforeEach`, так как здесь происходит кое-что интересное. Прежде всего здесь выполняется обращение к службе `$rootScope` и создается новый экземпляр `$scope` (вызовом `$rootScope.$new()`). Тем самым мы имитируем ситуацию, когда директива `ng-controller` в действующем приложении создает новый контекст.

Чтобы создать экземпляр контроллера, можно воспользоваться службой `$controller` (обратите внимание, что вызов функции `inject` допускается вставлять в раздел `beforeEach`, как и в раздел `it`).

Посмотрите, насколько просто можно передать аргументы конструктору контроллера, – достаточно лишь вызвать службу `$controller`. Это – внедрение зависимостей во всей своей красе; мы можем передать и фиктивный контекст, и тестовые данные для проверки реализации контроллера в полной изоляции.

## Фиктивные объекты и тестирование асинхронного кода

Как видите, AngularJS обеспечивает все необходимое для интеграции своего механизма внедрения зависимостей с фреймворком тестирования Jasmine. Но и это еще не все! AngularJS предоставляет также множество замечательных фиктивных объектов.

Асинхронное программирование – обычное дело в JavaScript. К сожалению, асинхронный код очень сложно тестировать. Асинхронные события непредсказуемы, они могут возникать в любой последовательности и в любые моменты времени. Однако все не так плохо, потому что командой проекта AngularJS было создано множество отличных фиктивных объектов. Они существенно упрощают тестирование асинхронного кода и, что самое важное, они предсказуемы. Разве такое возможно? Давайте рассмотрим пример теста, проверяющего код с помощью службы `$timeout`. Сначала сам код:

```
angular.module('async', [])
    .factory('asyncGreeter', function ($timeout, $log) {
        return {
            say:function (name, timeout) {
```

```
                $timeout(function(){
                    $log.info("Hello, " + name + "!");
                })
            }
        };
    });
```

Служба `$timeout` служит заменой функции `setTimeout`. Для выполнения отложенных действий предпочтительнее использовать службу `$timeout`, потому что она тесно интегрируется с компилятором AngularJS и вызывает обновление дерева DOM спустя указанное время. Получившийся в результате код легко поддается тестированию.

А теперь сам тест:

```
describe('Async Greeter test', function () {

    var asyncGreeter, $timeout, $log;
    beforeEach(module('async'));
    beforeEach(inject(function (_asyncGreeter_, _$timeout_,
_$log_) {
        asyncGreeter = _asyncGreeter_;
        $timeout = _$timeout_;
        $log = _$log_;
    }));

    it('should greet the async World', function () {
        asyncGreeter.say('World', 9999999999999999999);
        //
        $timeout.flush();
        expect($log.info.logs).toContain(['Hello, World!']);
    });
});
```

Большая часть этого кода не требует пояснений, но здесь есть два интересных момента, на которых мы остановимся подробнее. Первый – вызов метода `$timeout.flush()`. Этот незатейливый вызов метода фиктивного объекта `$timeout` имитирует возбуждение асинхронного события. Самое замечательное, что мы получаем полный контроль над тем, когда это событие произойдет. Нам не нужно ждать, пока истечет установленный таймаут или возникнет внешнее событие. Обратите внимание, что мы определили очень большой таймаут, но, несмотря на это тест будет выполнен немедленно, так как он не зависит от функции `setTimeout` – вместо нее используется фиктивный объект `$timeout`, имитирующий поведение асинхронной среды.

Замечательные, предсказуемые фиктивные объекты асинхронных служб – это одна из причин, почему тесты в AngularJS выполняются так быстро.

На многих платформах часто имеются глобальные службы, сложно поддающиеся тестированию. Примерами таких служб, вызывающих головную боль при тестировании, являются журналирование и обработка исключений. К счастью в AngularJS имеется решение и этой проблемы; фреймворк предоставляет собственные службы, решающие эти инфраструктурные задачи, вместе с сопутствующими фиктивными объектами. Возможно вы заметили, что в тесте выше используется еще один фиктивный объект – `$log`. Фиктивный объект, имитирующий службу `$log`, накапливает выводимые сообщения и сохраняет их для последующей проверки. Использование фиктивного объекта гарантирует, что тестируемый код не будет обращаться к фактической службе; вызов которой может оказаться весьма дорогостоящим, в терминах производительности, и производить побочные эффекты (например, можно предположить, что фактическая служба `$log` посылает сообщения на удаленный сервер и было бы крайне нежелательно генерировать сетевой трафик в процессе выполнения тестов).

## Интеграционные тесты

В AngularJS имеется собственное решение для организации интеграционного тестирования, которое называется **Scenario Runner**. Scenario Runner может выполнять интеграционное тестирование, управляя действиями в настоящем браузере. Этот инструмент автоматически выполняет требуемые операции (заполняет формы, имитирует щелчки мышью на кнопках и ссылках, и так далее) и проверяет отклик пользовательского интерфейса (изменение страницы, корректность отображаемой информации, и так далее), существенно снижая необходимость ручного тестирования, традиционно выполняемого специальными группами контроля качества.

Решение AngularJS обладает функциональностью, похожей на другие инструменты (например, очень популярный Selenium), но тесно интегрированное с фреймворком. Если говорить точнее, Scenario Runner.

- Поддерживает асинхронные события (наиболее важными из которых являются события XHR). Он способен приостанавливать выполнение теста до завершения обработки асинхрон-

ного события. На практике это означает отсутствие необходимости явно использовать в тестах какие-либо инструкции ожидания. Любой, кто хотя бы однажды пытался протестировать веб-приложение с поддержкой Ajax, используя традиционные инструменты, высоко оценят эту особенность.

- Позволяет использовать информацию связывания, уже имеющуюся в шаблонах AngularJS, для выбора элементов DOM с целью выполнения дальнейших операций с ними и проверки. В частности есть возможность находить элементы DOM и поля ввода форм, опираясь на модель, выполняющей связывание этих элементов. Это означает отсутствие необходимости встраивать избыточные атрибуты HTML (`ID` или классы CSS) или использовать ненадежные выражения XPath для поиска элементов DOM в тестах.
- Предоставляет функции, по-настоящему упрощающие поиск элементов DOM, взаимодействие с ними и исследование их свойств. Фактически в этом инструменте реализован полноценный **предметно-ориентированный язык** (Domain-Specific Language, DSL) для поиска и повторного сопоставления, ввода данных и так далее.

К сожалению, с течением времени проявлялось все больше и больше ограничений Scenario Runner. Из-за этого, к моменту написания этих строк, активная поддержка Scenario Runner практически прекратилась. Существуют планы его замены другим решением, Protractor, основанным на интеграции с инструментом Selenium. Следить за ходом работ можно в репозитории GitHub: https://github.com/angular/protractor.

Мы рекомендуем воздержаться от использования существующего решения Scenario Runner. Этот инструмент больше не поддерживается и не развивается. Обратите лучше внимание на Protractor.

### Повседневный рабочий процесс

Для достижения максимальной эффективности, автоматическое тестирование должно выполняться неукоснительно. Тесты должны выполняться максимально часто, а обнаруженные ими ошибки следует исправлять как можно быстрее. Ошибку в тесте следует интерпретировать как ошибку сборки, а исправление ошибок сборки должно быть самой приоритетной задачей в команде.

В течение дня мы постоянно переключаемся между кодом на JavaScript и шаблонами пользовательского интерфейса. Сосредоточившись исключительно на программном коде и на других артефактах AngularJS (таких как фильтры или директивы), важно запускать модульное тестирование как можно чаще. Фактически, благодаря высокой скорости выполнения тестов в AngularJS, тесты можно выполнять практически при каждом сохранении любого файла!

В процессе разработки примера приложения SCRUM, авторы книги использовали инструмент Karma runner, настроенный так, чтобы следить за изменениями во всех файлах (с кодом приложения и тестов). Каждый раз, когда сохранялся какой-нибудь файл, автоматически выполнялся весь комплект тестов, что позволяло мгновенно получать информацию о состоянии проекта. При таких настройках разработчик получает быструю обратную связь и всегда знает – действует ли код проекта, как ожидается, или что-то было нарушено несколько секунд тому назад. Если все в порядке, можно продолжать быстро двигаться вперед, но если тесты начинают терпеть неудачу, можно быть уверенными, что проблемы вызваны последними изменениями. Запуская тесты достаточно часто, можно навсегда распрощаться с утомительными сеансами отладки.

Выполнение тестов не должно требовать приложения значительных усилий. Очень важно иметь удобную среду тестирования, где комплект автоматизированных тестов может выполняться как можно чаще. Если для тестирования будет необходимо выполнить несколько утомительных операций вручную, мы подсознательно будем стремиться уклониться от тестирования.

Современные инструменты позволяют настраивать очень эффективные окружения тестирования. Например, на рис. 2.7 представлен скриншот среды разработки, использовавшейся для создания примеров к этой книге. Как видно на рис. 2.7, исходный код и код теста открыты одновременно и между ними легко переключаться. Инструмент Karma runner осуществляет мониторинг файловой системы и запускает тестирование всякий раз, когда обнаруживает операцию сохранения файла, и немедленно выводит отчет (в нижней панели).

Среда модульного тестирования, описываемая здесь, фактически является самой легковесной средой разработки. Мы можем вообще не покидать среду разработки и все внимание уделять созданию кода. Нам не нужно переключаться между окнами, чтобы выполнить сборку, развертывание и проверку правильности работы приложения в браузере.

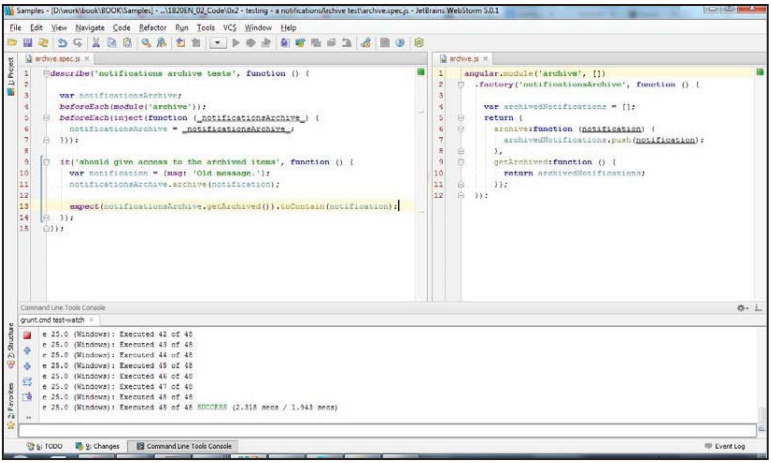

**Рис. 2.7.** Среда разработки с открытыми исходными файлами и панелью с результатами тестирования

## Советы и рекомендации по использованию Karma runner

Применение методики разработки через тестирование (Test Driven Development, TDD) значительно уменьшает количество длинных сеансов отладки. В этом случае мы вносим изменения в тесты небольшими порциями и тут же выполняем тестирование. Если тест терпит неудачу, нам не нужно идти далеко, так как ошибка находится рядом– всего в нескольких нажатиях клавиш. Но, какие бы усилия мы ни прикладывали, иногда все же бывает очень трудно понять, что происходит, не прибегая к помощи отладчика. В таких случаях следует тут же изолировать тест, вызвавший ошибку, и сосредоточиться именно на нем.

### Выполнение подмножества тестов

Версия Jasmine, распространяемая вместе с Karma runner, имеет очень полезные расширения для быстрой изоляции тестов:

- добавление приставки `x` к тесту или комплекту тестов (`xit`, `xdescribe`) предотвратит запуск этого теста/комплекта;
- добавление приставки `d` к комплекту тестов (`ddescribe`) предотвратит запуск всех комплектов, кроме данного;
- добавление приставки `i` к тесту (`iit`) предотвратит запуск всех тестов, кроме данного.

Эти короткие приставки очень удобны в применении и могут использоваться, как показано ниже:

```
describe('tips & tricks', function () {

    xdescribe('none of the tests here will execute', function () {

        it('won't execute - spec level', function () {
        });

        xit('won't execute - test level', function () {
        });
    });

    describe('suite with one test selected', function () {

        iit('will execute only this test', function () {
        });
        it('will be executed only after removing iit', function () {
        });
    });
});
```

### Отладка

Сужение области поиска причин ошибки до одного теста – это половина успеха. Но нам все еще нужно понять, что же собственно произошло, а для этого часто приходится начать сеанс отладки. Но как отлаживать тесты, запускаемые с помощью Karma runner? Как оказывается, в этом нет ничего сложного. Достаточно просто добавить инструкцию `debugger` в тест или в прикладной код.

Используя инструкцию `debugger`, не забудьте включить инструменты разработчика в своем браузере, иначе браузер не будет останавливаться, и вы не сможете отладить тесты.

Иногда бывает проще вывести значения некоторых переменных в консоль. В арсенале фиктивных объектов AngularJS имеется очень удобный метод как раз для этой цели: `angular.mock.dump(object)`. В действительности этот метод экспортируется также в глобальный контекст (в контекст окна), поэтому вызов можно записать проще: `dump(object)`.

# В заключение

В этой главе мы подготовились разработке настоящих приложений на основе AngularJS. Сначала мы познакомились с некоторыми осо-

бенностями обсуждаемого примера приложения; его назначением и стеком технологий. В оставшейся части книги мы увидим множество примеров, инспирированных приложением для управления проектами с применением методики гибкой разработки SCRUM. Для разработки этого приложения мы будем использовать открытые программные продукты на JavaScript (или, по крайней мере, дружественные для JavaScript): аутсорсинговую версию MongoDB для хранения данных, `node.js` – в качестве серверного решения, и, конечно же AngularJS – как фреймворк для клиентской части.

Мы потратили значительный объем времени на обсуждение проблем, связанных со сборкой, и ее практическую реализацию. Для создания сценариев сборки примера приложения мы выбрали инструмент `grunt.js`, реализованный на JavaScript (и действующий под управлением `node.js`). В процессе знакомства с системой сборки мы пришли к необходимости изучения вопросов организации файлов и каталогов проекта. В результате мы определили структуру каталогов, хорошо согласующуюся с системой модулей в AngularJS. После рассмотрения различных способов объявления модулей и провайдеров, мы выбрали синтаксические конструкции, устраняющие беспорядок, исключающие загрязнение глобального пространства имен и удерживающие сложность системы сборки на приемлемом уровне.

Преимущества автоматического тестирования хорошо известны в наши дни. Нам очень повезло, что в AngularJS тестированию кода, написанного с использованием фреймворка, уделяется особое внимание. Мы узнали, что AngularJS содержит полноценный комплект инструментов тестирования (Karma runner), хорошо интегрируется с существующими библиотеками тестирования (Jasmine) и предусматривает в своем механизме внедрения зависимостей возможность тестирования объектов по отдельности. Мы увидели, как можно тестировать артефакты AngularJS, поэтому тестирование служб и контроллеров не должно вызывать у вас затруднений. После знакомства со всеми решениями мы увидели, как легко объединить их все вместе и получить легковесную среду разработки и тестирования.

К настоящему моменту мы познакомились с основами AngularJS и узнали, как структурировать нетривиальные приложения. Теперь мы можем использовать эти основы и добавить к ним действующие прикладные элементы. Так как любому CRUD-приложению приходится работать с данными, извлекаемыми из хранилища, следующая глава будет посвящена приемам взаимодействий с различными серверными компонентами, поддерживаемым фреймворком AngularJS.

# ГЛАВА 3. Взаимодействие с сервером

Редко можно встретить веб-приложение, которому не требуется взаимодействовать с хранилищем, чтобы получить данные для обработки. Это особенно верно для CRUD-подобных приложений, в которых правка данных является основной операцией.

Фреймворк AngularJS обеспечивает поддержку взаимодействий с различными серверными компонентами с применением запросов типа **XMLHttpRequest (XHR)** и **JSONP**. В нем имеется универсальная служба `$http` выполняющая запросы XHR и JSONP, а также специализированная служба `$resource`, обеспечивающая взаимодействия с конечными точками RESTful.

В этой главе мы исследуем различные функции и способы взаимодействий с разными серверными компонентами. В частности мы узнаем, как:

- выполнять запросы XHR с помощью службы `$http` и тестировать код, пользующийся этой службой;
- эффективно работать с асинхронными запросами, используя методы службы `$q` в AngularJS;
- взаимодействовать с конечными точками RESTful, используя специализированную фабрику `$resource`;
- создавать собственные `$resource`-подобные инструменты для взаимодействия с нестандартными серверными компонентами.

## Выполнение запросов XHR и JSONP с помощью $http

Служба `$http` является фундаментальной, универсальной службой для выполнения запросов XHR и JSONP. Это хорошо продуманная и удобная служба, в чем вы вскоре убедитесь. Но, прежде чем пог-

ружаться в обсуждение особенностей $http, нам необходимо создать модель данных для нашего примера приложения SCRUM, чтобы наполнить последующие примеры смыслом.

## Модель данных и адреса URL в MongoLab

В приложении SCRUM используется достаточно простая модель данных, изображенная на рис. 3.1.


**Рис. 3.1.** Модель данных в приложении SCRUM

Существует пять разных коллекций MongoDB, в которых хранятся сведения о пользователях, проектах и артефактах проектов. Все данные доступны через интерфейс RESTful веб-службы MongoLab. Обращения к интерфейсу REST выполняются с помощью специальных адресов URL, сформированных по следующему шаблону:

```
https://api.mongolab.com/api/1/databases/[DB name]/
collections/
[collection name]/[item id]?apiKey=[secret key]
```

Все обращения к REST API баз данных, размещенных на серверах MongoLab, должны включать параметр запроса с именем apiKey. В параметре apiKey должен передаваться уникальный идентификатор учетной записи, необходимый для авторизации операций с MongoLab REST API. Полное описание REST API, экспортируемого MongoLab, можно найти по адресу: https://support.mongolab.com/entries/20433053-rest-api-for-mongodb.

## Краткий обзор $http

Служба $http существенно упрощает выполнение запросов XHR и JSONP. Давайте рассмотрим пример извлечения данных в формате JSON с помощью GET-запроса:

```
var futureResponse = $http.get('data.json');
futureResponse.success(function (data, status, headers, config) {
    $scope.data = data;
});
```

```
futureResponse.error(function (data, status, headers, config) {
    throw new Error('Something went wrong...');
});
```

В этом примере можно сразу заметить специализированный метод для выполнения запросов XHR GET. Существуют эквивалентные методы и для запросов других типов:

- GET: `$http.get(url, config);`
- POST: `$http.post(url, data, config);`
- PUT: `$http.put(url, data, config);`
- DELETE: `$http.delete(url, config);`
- HEAD: `$http.head`.

Существует так же возможность выполнять запросы JSONP с помощью метода `$http.jsonp(url, config)`.

Параметры, принимаемые методами службы `$http`, несколько отличаются, в зависимости от метода HTTP-запроса. Для запросов, которые несут данные в своем теле (POST и PUT), сигнатура метода включает следующие параметры:

- `url`: адрес URL, куда направляется запрос;
- `data`: данные для передачи в теле запроса;
- `config`: объект JavaScript, содержащий дополнительные параметры настройки, оказывающие влияние на отправку запроса и получение ответа.

В запросах остальных типов (GET, DELETE, HEAD, JSONP) не требуется передавать данные в теле запроса, поэтому соответствующие методы принимают всего два параметра: `url` и `config`.

Объект, возвращаемый методами `$http`, позволяет зарегистрировать функции обратного вызова `success` и `error`.

### Пример объекта с настройками

Объект с настройками может хранить различные параметры, управляющие запросом, ответом и передаваемыми данными. Объект с параметрами может обладать следующими свойствами (кроме прочих):

- `method`: используемый HTTP-метод запроса;
- `url`: адрес URL, куда направляется запрос;
- `params`: параметры для добавления в строку запроса URL;
- `headers`: дополнительные заголовки для добавления в запрос;
- `timeout`: таймаут (в миллисекундах) ожидания ответа на запрос XHR;

- `cache`: управляет кешированием запроса XHR GET;
- `transformRequest`, `transformResponse`: функции преобразования для предварительной и заключительной обработки данных, которыми обмениваются клиент и сервер.

Возможно вас удивит присутствие параметров `method` и `url`, поскольку они уже заключены в сигнатуру методов службы `$http`. Как оказывается, сама служба `$http` – это функция, которая может вызываться обобщенным способом:

```
$http(configObject);
```

Эта обобщенная форма может пригодиться для выполнения запросов, для которых в AngularJS отсутствует «специализированный» метод, (например, PATCH или OPTIONS). Вообще говоря, специализированные методы позволяют писать более краткий и выразительный код и мы рекомендуем всегда использовать их, если это возможно.

### Преобразование данных запроса

Методы `$http.post` и `$http.put` принимают в аргументе `data` любой объект JavaScript (или строку). Если передается объект JavaScript, он автоматически будет преобразован в строку в формате JSON.

По умолчанию механизм преобразования объектов в формат JSON игнорирует все свойства, имена которых начинаются со знака доллара (`$`). Вообще говоря, свойства с именами, начинающимися со знака доллара, интерпретируются в AngularJS как приватные. Это может стать источником проблем при работе с некоторыми серверными технологиями, которые принимают свойства с такими именами (например, MongoDB). Обходное решение заключается в том, чтобы выполнять преобразование данных вручную (с помощью метода `JSON.stringify`, например).

Чтобы увидеть, как действует механизм преобразования, можно попробовать выполнить POST-запрос с целью создать нового пользователя в MongoLab:

```
var userToAdd = {
    name:'AngularJS Superhero',
    email:'superhero@angularjs.org'
};

$http.post('https://api.mongolab.com/api/1/databases/ascrum/
collections/users',
    userToAdd, {
        params:{
```

```
            apiKey:'4fb51e55e4b02e56a67b0b66'
        }
    });
```

Этот пример также иллюстрирует, как можно добавлять параметры в URL HTTP-запроса (в данном случае: `apiKey`).

### Обработка HTTP-ответов

Запрос может увенчаться успехом или потерпеть неудачу. В AngularJS имеются два метода для регистрации функций обратного вызова, чтобы обработать эти две ситуации: успешное выполнение и неудачу. Оба метода принимают функцию обратного вызова, которая будет вызвана со следующими параметрами:

- `data`: фактические данные в ответе;
- `status`: HTTP-код состояния ответа;
- `headers`: функция, дающая доступ к HTTP-заголовкам ответа;
- `config`: объект с настройками, применявшимися при отправке запроса.

Функция `success` будет вызвана фреймворком AngularJS для любых ответов с HTTP-кодом состояния в диапазоне от 200 до 299. Для обработки ответов с другими кодами состояния будет вызвана функция `error`. Ответы с информацией о перенаправлении (с кодами состояния 3xx) автоматически будут обрабатываться самим браузером.

Обе функции, `success` и `error`, являются необязательными. Если при выполнении запроса не была зарегистрирована ни одна функция обратного вызова, ответ просто будет проигнорирован.

### Преобразование данных ответа

Как и в случае с преобразованием данных запроса, служба `$http` попытается преобразовать строку в формате JSON, содержащуюся в ответе, в объект JavaScript. Преобразование будет выполнено перед вызовом функции `success` или `error`. Поведение по умолчанию механизма преобразования поддается настройке.

В текущей версии AngularJS, служба `$http` будет пытаться выполнить преобразование в объект JavaScript любой строки в ответе, которая выглядит, как строка в формате JSON (то есть, начинается с открывающей скобки `{` или `[` и заканчивается закрывающей скобкой `]` или `}`).

## Ограничения политики общего происхождения

Веб-браузеры проводят принудительную **политику общего происхождения** (Same-Origin Security Policy). Согласно этой политике, XHR-взаимодействия могут выполняться только с ресурсами, имеющими тот же источник (идентификация источника выполняется по комбинации протокола, имени сервера и номера порта) и запрещены с «посторонними» ресурсами.

Как веб-программистам нам постоянно придется балансировать между обеспечением безопасности и функциональными требованиями, чтобы обеспечить сбор данных из нескольких источников. В действительности достаточно часто возникает необходимость получить данные из стороннего источника и отобразить их. К сожалению, очень непросто организовать отправку запросов XHR серверам, находящимся за пределами исходного домена, не прибегая к разного рода хитростям.

В настоящее время существует множество приемов доступа к данным на внешних серверах: **JSON с дополнением** (JSON with Padding, JSONP) и **ресурсы, разделяемые разными источниками** (Cross-Origin Resource Sharing, CORS), являются, пожалуй, самыми популярными на сегодняшний день. В этом разделе мы покажем, как AngularJS помогает применять эти приемы на практике.

### Преодоление ограничений политики общего происхождения с помощью JSONP

Прием JSONP – это один из трюков, помогающих преодолевать ограничения политики общего происхождения и получать данные из сторонних источников. Он опирается на тот факт, что браузеры свободно могут загружать сценарии на JavaScript со сторонних серверов с помощью тега `<script>`.

При использовании приема JSONP запросы XHR не производятся, а генерируется тег `<script>`, атрибут `source` которого ссылается на внешний ресурс. Как только сгенерированный тег `<script>` окажется в дереве DOM, браузер отправит запрос серверу. Сервер дополнит ответ вызовом функции (откуда и взялось слово «с дополнением» (padding) в названии приема JSONP) в нашем приложении.

Давайте рассмотрим пример запроса JSONP и ответ на него, чтобы понять, как все это действует на практике. Сначала выполним запрос JSONP:

```
$http
    .jsonp('http://angularjs.org/greet.php?callback=JSON_CALLBACK', {
        params:{
            name:'World'
        }
    }).success(function (data) {
        $scope.greeting = data;
    });
```

Метод `$http.jsonp` динамически создаст новый элемент `<script>` в дереве DOM:

```
<script type="text/javascript"
  src="http://angularjs.org/greet.php?callback=angular.
callbacks._k&name=World">
</script>
```

Как только это тег будет включен в дерево DOM, браузер отправит запрос по адресу, указанному в атрибуте `src`. Обратно будет получен ответ, содержащей примерно такое тело:

```
angular.callbacks._k (
    {"name":"World","salutation":"Hello","greeting":"Hello World!"}
);
```

Ответ JSONP выглядит как вызов обычной функции JavaScript и фактически таковым и является. AngularJS автоматически сгенерирует функцию `angular.callbacks._k`. Когда эта функция будет вызвана, она произведет вызов функции `success`. Адрес URL, что передается методу `$http.jsonp`, должен содержать параметр запроса `JSON_CALLBACK`, а AngularJS превратит эту строку в имя динамической функции.

Имена функций обратного вызова, генерируемые фреймворком AngularJS при использовании приема JSONP, всегда имеют форму `angular.callbacks._[variable]`. Поэтому убедитесь, что серверная часть приложения способна принимать имена сфункций с точками.

## Ограничения JSONP

Прием JSONP является неплохим обходным решением для преодоления ограничений политики общего происхождения, но он имеет несколько ограничений. Прежде всего, с его помощью можно выполнять только запросы HTTP GET. Обработка ошибок также является слабым местом этого приема, так как браузеры не возвращают HTTP-код

ответа из тега <script>. На практике это означает, что очень сложно будет определить HTTP-код ошибки и вызвать функцию error.

Кроме того, применение приема JSONP снижает безопасность веб-приложения. Помимо нападений типа XSS, самая большая проблема заключается в том, что в ответ на JSONP-запрос сервер может сгенерировать произвольный код на JavaScript. Этот запрос будет загружен браузером и выполнен в контексте сеанса пользователя. То есть, злоумышленник получает возможность выполнить любой злонамеренный код, вызывающий различные повреждения, от простого приведения страницы в неработоспособное состояние, до кражи конфиденциальных данных. Учитывая это, следует проявлять большую осторожность в выборе служб, позволяющих получать данные с использованием приема JSONP, и пользоваться только доверенными серверами.

### Преодоление ограничений политики общего происхождения с помощью CORS

Прием разделения ресурсов разными источниками (Cross-Origin Resource Sharing, CORS) разработан консорциумом W3C и преследует ту же цель, что и JSONP, но стандартным, надежным и безопасным способом. Прием CORS основан на применении объекта XMLHttpRequest для выполнения междоменных AJAX-запросов четко оговоренным и контролируемым способом.

Идея этого приема состоит в том, что браузер и сторонний сервер должны договариваться между собой (отправкой соответствующих заголовков в запросах и ответах). Отсюда следует, что сторонний сервер должен быть настроен соответствующим образом. Браузеры должны иметь возможность посылать запросы с соответствующими заголовками и интерпретировать ответы сервера, чтобы обеспечить успешное выполнение междоменных запросов.

Сторонний сервер должен быть настроен соответственно, чтобы иметь возможность участвовать в диалоге CORS. Если вам потребуется настроить свои серверы на прием запросов HTTP CORS, обращайтесь за дополнительной информацией по адресу: http://www.html5rocks.com/en/tutorials/cors/. А мы сосредоточимся далее исключительно на роли браузера в этом взаимодействии.

Запросы CORS грубо делятся на две категории – «простые» и «сложные». Простыми считаются запросы GET, POST и HEAD (но только с определенным подмножеством заголовков). Попытка вы-

полнить другой HTTP-запрос или использовать заголовки, не входящие в перечень допустимых, вынудит браузер отправить «сложный» запрос CORS.

Большинство современных браузеров поддерживают запросы CORS, что называется «из коробки». Версии 8 и 9 браузера Internet Explorer поддерживают запросы CORS только с помощью нестандартного объекта `XDomainRequest`. Из-за ограничений, характерных для объекта `XDomainRequest`, AngularJS не поддерживает его. Таким образом, запросы CORS не поддерживаются службой `$http` в версиях IE 8 и 9.

При выполнении сложного запроса браузер вынужден послать предварительный (**предполетный – preflight**) запрос OPTION, получить утвердительный ответ от сервера и только потом выполнить основной запрос. Эта особенность браузеров часто вызывает недопонимание, так как при анализе HTTP-трафика обнаруживаются мистические запросы OPTIONS. Мы можем увидеть эти запросы, пытаясь вызвать MongoLab REST API непосредственно из браузера. Например, рассмотрим HTTP-запрос на удаление пользователя:

```
$http.delete(
    'https://api.mongolab.com/api/1/databases/ascrum/
collections/users/' +
    userId,
    {
        params:{
            apiKey:'4fb51e55e4b02e56a67b0b66'
        }
    }
);
```

В результате выполнения этого фрагмента можно увидеть, что по указанному адресу было отправлено два запроса (OPTIONS и DELETE), как показано на рис. 3.2.

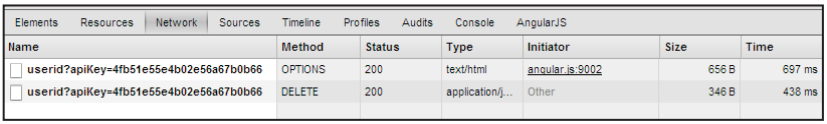


**Рис. 3.2.** По указанному адресу отправлено два запроса (OPTIONS и DELETE)

Ответ от сервера MongoLab включает заголовки, подтверждающие допустимость выполнения запроса DELETE, как показано на рис. 3.3.

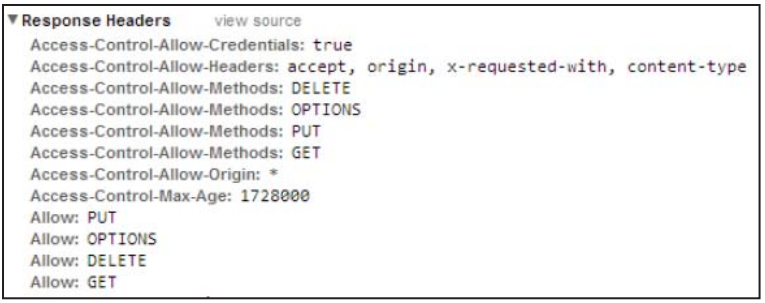


**Рис. 3.3.** Ответ сервера подтверждает допустимость запроса DELETE

Серверы MongoLab отлично настроены на отправку соответствующих заголовков в ответ на запросы CORS. Если ваш сервер не будет настроен на обработку запросов OPTIONS должным образом, запросы клиентов будут терпеть неудачу.

Не удивляйтесь, увидев запросы OPTIONS при анализе HTTP-трафика – это всего лишь способ установить связь при выполнении запросов CORS. Ошибки при выполнении запросов OPTIONS чаще всего свидетельствуют о том, что сервер не настроен должным образом.

### Проксирование запросов на стороне сервера

Прием JSONP – не самый идеальный способ выполнения междоменных запросов. Прием CORS несколько улучшает ситуацию, но он требует дополнительных настроек на стороне сервера и использования браузера, поддерживающего стандарты.

Если по каким-то причинам вы не можете использовать прием CORS или JSONP, остается еще один способ преодоления ограничений, накладываемых политикой общего происхождения. Суть его заключается в организации на локальном сервере проксирования запросов к сторонним серверам. Применяя соответствующие настройки на локальном сервере, можно организовать передачу междоменных запросов через наш сервер, то есть браузер будет взаимодействовать только с нашим сервером. Этот прием может использоваться с любыми браузерами и не требует выполнять «предполетные» запросы OPTIONS. Кроме того, он не влечет за собой рисков, связанных с безопасностью. Недостаток такого подхода заключается в выполнении дополнительных настроек на сервере.

Пример приложения SCRUM, описываемого в данной книге, опирается на работу с сервером `node.js`, настроенным на проксирование запросов к MongLab REST API.

# Promise API и служба $q

Программисты на JavaScript приучены работать с асинхронной моделью программирования. И браузер, и среда выполнения `node.js` наполнены асинхронными событиями, такими как ответы XHR, события DOM, ввод/вывод и таймауты, которые могут возникнуть в любой момент и в случайном порядке. Но, даже имея навыки работы в асинхронной среде выполнения, асинхронное программирование может иногда вызывать затруднения, особенно когда возникает необходимость синхронизировать асинхронные события.

В синхронном мире вызовы функций, выстроенные в цепочку (когда каждая следующая функция вызывается с результатом вызова предыдущей), и обработка исключений (с помощью конструкции `try/catch`) выполняются последовательно. В асинхронном мире мы не можем просто взять и составить цепочку из вызовов функций и вынуждены опираться на функции обратного вызова. Функции обратного вызова отлично справляются со своими обязанностями, когда требуется обработать единственное асинхронное событие, но когда возникает потребность скоординировать несколько асинхронных событий, сложности начинают нарастать как снежный ком. Обработка исключений в таких ситуация становится особенно сложным предприятием.

Чтобы упростить асинхронное программирование, совсем недавно в некоторых популярных библиотеках JavaScript была реализована поддержка Promise API. Концепция, лежащая в основе Promise API, не нова и была предложена еще в конце 1970 годов, но лишь совсем недавно она нашла свое место в программировании на JavaScript.

Основная идея Promise API состоит в том, чтобы привнести в асинхронный мир простоту составления цепочек вызовов функций и обработки ошибок, которая имеется в мире синхронного программирования.

В состав AngularJS входит служба `$q` – легковесная реализация Promise API. Многие службы в AngularJS (наиболее заметными из

которых являются `$http`, `$timeout` и другие) прочно опираются на использование интерфейса в стиле Promise API, поэтому для эффективного их использования нам совершенно необходимо познакомиться со службой `$q`.

Толчком к реализации службы `$q` стало появление библиотеки Q Promise API, разработанной Крисом Ковалем (Kris Kowal): https://github.com/kriskowal/q. Ознакомьтесь с этой библиотекой, чтобы получить более полное представление об основной концепции и возможность сравнить легковесную реализацию в AngularJS с библиотекой, реализующей полноценный Promise API.

## Получение отложенных результатов с помощью службы $q

Знакомиться с относительно небогатым прикладным интерфейсом службы `$q` мы будем на примерах, взятых из нашей практики, наглядно демонстрирующих, как использовать Promise API для обработки не только вызовов XHR, но и любых асинхронных событий.

### Основы службы $q

Представьте, что мы хотим заказать по телефону пиццу с доставкой на дом. В результате этого звонка либо пицца будет доставлена на дом, либо по телефону нам сообщат, что это невозможно. Чтобы заказать пиццу, достаточно одного короткого телефонного звонка, для выполнения заказа (доставки пиццы) потребуется некоторое время, при этом событие доставки можно расценивать как асинхронное.

Чтобы получить представление о Promise API, давайте смоделируем заказ пиццы и успешную ее доставку с помощью службы `$q`. Прежде всего определим реакцию человека – довольную, в ответ на получение вожделенной пиццы, и недовольную, в ответ на отказ:

```
var Person = function (name, $log) {

    this.eat = function (food) {
        $log.info(name + " is eating delicious " + food);
    };

    this.beHungry = function (reason) {
        $log.warn(name + " is hungry because: " + reason);
    }
};
```

Конструктор `Person`, представленный выше, можно использовать для создания объекта с методами `eat` и `beHungry`. Эти методы мы будем использовать в качестве функций обратного вызова `success` и `error`, соответственно.

Теперь смоделируем заказ пиццы и благополучное его выполнение с помощью теста Jasmine:

```
it('should illustrate basic usage of $q', function () {

    var pizzaOrderFulfillment = $q.defer();
    var pizzaDelivered = pizzaOrderFulfillment.promise;

    pizzaDelivered.then(pawel.eat, pawel.beHungry);

    pizzaOrderFulfillment.resolve('Margherita');
    $rootScope.$digest();

    expect($log.info.logs).toContain(['Pawel is eating delicious
Margherita']);
});
```

Модульный тест начинается с вызова метода `$q.defer()`, возвращающего **объект отложенного задания** (deferred). Концептуально он представляет задание, которое будет выполнено (или не выполнено) когда-то в будущем. Этот объект играет двоякую роль:

- хранит объект отложенного результата (свойство `promise`), служащий контейнером для будущего результата задания;
- предоставляет методы, вызывающие успешное (`resolve`) или неудачное (`reject`) завершение задания.

При использовании Promise API всегда действуют два игрока: один контролирует ход выполнения задания (может вызывать методы объекта отложенного задания), и другой, реагирующий на тот или иной результат выполнения задания.

Объект отложенного задания представляет задание, которое может завершиться успехом или неудачей в будущем. Объект отложенного результата служит контейнером для фактического результата, который будет получен в будущем в случае успешного выполнения задания.

Объект, контролирующий выполнение задания (в нашем случае это была бы пиццерия), предоставляет доступ к объекту с результатом (`pizzaOrderFulfillment.promise`) другому объекту, заинтересованному в получении результата. В нашем примере заинтересованным

объектом является объект `Pawel` и он может выразить свою заинтересованность регистрацией функций обратного вызова в объекте отложенного задания. Регистрация выполняется с помощью метода `then(successCallBack, errorCallBack)`. Этот метод принимает функцию, которая должна быть вызвана в случае успеха (в данном случае метод `eat`) или неудачи (в данном случае метод `beHungry`) выполнения задания в будущем. Функция обработки ошибки является необязательной и может быть опущена. Если эта функция не была зарегистрирована, и попытка выполнить задание потерпела неудачу, ошибка просто будет проигнорирована.

Чтобы сообщить об успешном завершении задания, следует вызвать его метод `resolve`. Аргумент, переданный методу `resolve`, автоматически будет передан функции обработки успешного выполнения задания. После этого отложенное задание считается выполненным, а объект отложенного результата – готовым к использованию. Аналогично, вызов метода `reject` сообщает о неудачной попытке выполнить задание и вызывает функцию обработки ошибки.

В примере реализации теста выше присутствует таинственный вызов `$rootScope.$digest()`. В AngularJS событие успешного (или неудачного) завершения задания, распространяется как часть цикла `$digest`. Подробнее о внутренних механизмах фреймворка AngularJS и цикле `$digest` будет рассказываться в главе 11.

## Объекты с результатами – обычные объекты JavaScript

На первый взгляд может показаться, что применение Promise API только увеличивает сложность кода. Но не будем торопиться с выводами, а рассмотрим дополнительные примеры, которые помогут оценить настоящую мощь этого интерфейса. Прежде всего обратите внимание, что объекты отложенных результатов, – это самые обычные объекты JavaScript. Их можно передавать функциям в виде аргументов и возвращать из функций. Это позволяет реализовать асинхронные операции в виде служб. Например, представьте себе упрощенную службу, моделирующую работу ресторана:

```
var Restaurant = function ($q, $rootScope) {

    var currentOrder;

    this.takeOrder = function (orderedItems) {
```

```
            currentOrder = {
                deferred:$q.defer(),
                items:orderedItems
            };
            return currentOrder.deferred.promise;
        };

        this.deliverOrder = function() {
            currentOrder.deferred.resolve(currentOrder.items);
            $rootScope.$digest();
        };

        this.problemWithOrder = function(reason) {
            currentOrder.deferred.reject(reason);
            $rootScope.$digest();
        };
    };
```

Эта служба инкапсулирует асинхронные задания и возвращает только объекты отложенного результата из метода `takeOrder`. Возвращаемый объект может использоваться клиентом для доступа к будущему результату и получения извещения, когда этот результат будет доступен.

Чтобы показать, как действует созданная нами модель, напишем код, который будет моделировать неудачу и вызывать функцию обработки ошибки:

```
it('should illustrate promise rejection', function () {

    pizzaPit = new Restaurant($q, $rootScope);
    var pizzaDelivered = pizzaPit.takeOrder('Capricciosa');

    pizzaDelivered.then(pawel.eat, pawel.beHungry);

    pizzaPit.problemWithOrder(
        'no Capricciosa, only Margherita left');
    expect($log.warn.logs).toContain(
        ['Pawel is hungry because: no Capricciosa, only
Margherita left']);
});
```

## Объединение функций обратного вызова

Для одного объекта отложенного результата можно зарегистрировать сразу несколько функций обратного вызова. Чтобы понять, где это может пригодиться, давайте представим, что оба автора этой книги вместе заказали пиццу и оба заинтересованы в выполнении заказа:

```
it('should allow callbacks aggregation', function () {

    var pizzaDelivered = pizzaPit.takeOrder('Margherita');
    pizzaDelivered.then(pawel.eat, pawel.beHungry);
    pizzaDelivered.then(pete.eat, pete.beHungry);

    pizzaPit.deliverOrder();
    expect($log.info.logs)
        .toContain(['Pawel is eating delicious Margherita']);
    expect($log.info.logs)
        .toContain(['Peter is eating delicious Margherita']);
});
```

В этом примере мы зарегистрировали несколько функций обратного вызова для обработки ситуации успешного выполнения задания и все они будут вызваны при получении результата. Аналогично можно зарегистрировать несколько функций обработки ошибки.

### Регистрация функций обратного вызова и жизненный цикл отложенных заданий

Как только отложенное задание завершится успехом или неудачей, его состояние уже не изменится. Отложенное задание имеет только один шанс получить результат. Иными словами невозможно:

- заменить отрицательный результат положительным;
- заменить один положительный результат другим положительным результатом;
- заменить положительный результат отрицательным;
- заменить один отрицательный результат другим отрицательным результатом.

Эти правила вполне очевидны. Например, было бы бессмысленно, если бы у пиццерии имелась возможность перезвонить нам и сообщить, что заказ не может быть доставлен, уже после того как пицца была доставлена (и может быть даже съедена!).

Любые функции обратного вызова, зарегистрированные после завершения отложенного задания, будут немедленно вызваны с уже имеющимся результатом.

### Объединение асинхронных операций в цепочки

Возможность объединения функций обратного вызова может найти определенное применение в практике, однако истинная мощь

Promise API заключена в возможности имитировать синхронные вызовы функций в асинхронном мире.

Продолжая пример с пиццей, давайте представим, что на этот раз мы приглашены на пиццу к нашим друзьям. Хозяева закажут пиццу, и когда она будет доставлена, ее нарежут и подадут на стол. Здесь наблюдается целая цепочка асинхронных событий: сначала пицца должна быть доставлена, и только потом нарезана и подана на стол. Здесь присутствуют два задания, которые должны быть выполнены перед тем, как мы получим возможность насладиться пиццей: пиццерия должна доставить пиццу, а затем хозяева должны нарезать и подать ее. Рассмотрим код, моделирующий эту ситуацию:

```
it('should illustrate successful promise chaining', function () {

    var slice = function(pizza) {
        return "sliced "+pizza;
    };

    pizzaPit.takeOrder('Margherita').then(slice).then(pawel.eat);

    pizzaPit.deliverOrder();

    expect($log.info.logs)
        .toContain(['Pawel is eating delicious sliced
Margherita']);});
```

В этом примере можно видеть цепочку заданий (вызовов метода `then`). Эта конструкция близко напоминает следующий синхронный код:

```
pawel.eat(slice(pizzaPit));
```

Объединение заданий в цепочку возможно, только благодаря тому, что метод `then` возвращает новый отложенный результат, фактическим результатом которого будет значение, возвращаемое функцией обратного вызова.

Еще более привлекательно выглядит простота обработки ошибок. Рассмотрим пример распространения ошибочной ситуации до человека, получившего отказ:

```
it('should illustrate promise rejection in chain', function ()
{

    pizzaPit
```

```
        .takeOrder('Capricciosa').then(slice).then(pawel.eat,
pawel.beHungry);

    pizzaPit
        .problemWithOrder('no Capricciosa, only Margherita left');
    expect($log.warn.logs).toContain(
        ['Pawel is hungry because: no Capricciosa, only
Margherita left']);
});
```

Здесь отказ пиццерии принять заказ доходит до человека, заинтересованного в конечном результате. Именно так работает механизм обработки исключений в синхронном мире: исключение будет всплывать вверх по стеку вызовов до ближайшего блока `catch`.

Функции обратного вызова обработки ошибок в Promise API действуют подобно блоку `catch`, и как при использовании стандартных блоков `catch` у нас есть несколько возможностей обработки исключений. Мы можем:

- восстановить нормальную работу (вернуть значение из блока `catch`);
- позволить исключению продолжить всплытие вверх по стеку (повторно возбудив его).

С помощью Promise API легко можно имитировать восстановление нормальной работы в блоке `catch`. Например, предположим, что хозяева приложили дополнительные усилия и заказали другую пиццу, когда выяснилось, что желаемая пицца отсутствует в продаже:

```
it('should illustrate recovery from promise rejection',
function () {

    var retry = function(reason) {
        return pizzaPit.takeOrder('Margherita').then(slice);
    };

    pizzaPit.takeOrder('Capricciosa')
        .then(slice, retry)
        .then(pawel.eat, pawel.beHungry);

    pizzaPit.problemWithOrder('no Capricciosa, only Margherita
left');
    pizzaPit.deliverOrder();

    expect($log.info.logs)
        .toContain(['Pawel is eating delicious sliced
Margherita']);
});
```

Из функции обратного вызова обработки ошибок можно вернуть новый отложенный результат. В этом случае возвращаемый отложенный результат станет частью цепочки отложенных результатов, и клиент даже не заметит, что произошла какая-то ошибка. Это очень мощный прием, который может применяться в любых сценариях, где требуется организовать повторение попыток. Мы будем использовать этот подход в главе 7 для реализации системы безопасности в приложении.

Другой случай, который мы должны рассмотреть, – повторное возбуждение исключения, что может потребоваться, если восстановление нормальной работы окажется невозможным. В таких ситуациях единственный выбор – сгенерировать другую ошибку, для чего служба `$q` предоставляет специальный метод (`$q.reject`):

```
it('should illustrate explicit rejection in chain', function () {

    var explain = function(reason) {
        return $q.reject('ordered pizza not available');
    };

    pizzaPit.takeOrder('Capricciosa')
        .then(slice, explain)
        .then(pawel.eat, pawel.beHungry);

    pizzaPit
        .problemWithOrder('no Capricciosa, only Margherita left');

    expect($log.warn.logs)
        .toContain(['Pawel is hungry because: ordered pizza not
available']);
});
```

Метод `$q.reject` является асинхронным эквивалентом инструкции возбуждения исключения. Он возвращает новый отложенный результат, который получает отрицательное значение, указанное в аргументе.

### Дополнительные сведения о службе $q

Служба `$q` обладает двумя дополнительными методами: `$q.all` и `$q.when`.

### Объединение отложенных результатов

Метод `$q.all` дает возможность запустить несколько асинхронных заданий и организовать получение извещения, только когда все они будут выполнены. Фактически, он объединяет отложенные результаты нескольких асинхронных операций и возвращает единственный,

объединенный отложенный результат, который может действовать как точка сопряжения.

Для иллюстрации ценности метода `$q.all`, рассмотрим пример заказа нескольких блюд из разных ресторанов. Мы могли бы пожелать дождаться, пока все заказы будут доставлены, и только потом подавать блюда на стол:

```
it('should illustrate promise aggregation', function () {

    var ordersDelivered = $q.all([
        pizzaPit.takeOrder('Pepperoni'),
        saladBar.takeOrder('Fresh salad')
    ]);

    ordersDelivered.then(pawel.eat);

    pizzaPit.deliverOrder();
    saladBar.deliverOrder();
    expect($log.info.logs)
        .toContain(['Pawel is eating delicious Pepperoni,Fresh
salad']);
});
```

Метод `$q.all` принимает массив отложенных результатов и возвращает объединенный отложенный результат, который получит фактическое значение, только когда будут получены фактические значения всех отдельных результатов. Но, если хотя бы одна из асинхронных операций потерпит неудачу, объединенный результат так же будет отрицательным:

```
it('should illustrate promise aggregation when one of the
promises fail',
    function () {

    var ordersDelivered = $q.all([
        pizzaPit.takeOrder('Pepperoni'),
        saladBar.takeOrder('Fresh salad')
    ]);

    ordersDelivered.then(pawel.eat, pawel.beHungry);

    pizzaPit.deliverOrder();
    saladBar.problemWithOrder('no fresh lettuce');
    expect($log.warn.logs)
        .toContain(['Pawel is hungry because: no fresh lettuce']);
});
```

В случае неудачи объединенный отложенный результат получит то же значение, что и конкретный отложенный результат.

### Передача значений в виде отложенных результатов

Иногда возникают ситуации, когда одни и те же функции должны работать с результатами не только асинхронных, но и синхронных операций. В таких случаях часто проще интерпретировать все результаты асинхронными.

Метод `$q.when` позволяет вернуть объект JavaScript в виде отложенного результата.

Продолжим пример «с пиццей и салатом» объединения отложенных результатов и представим, что салат уже готов (синхронная операция), а пицца должна быть заказана и доставлена (асинхронная операция). При этом для нас все еще желательно подать оба блюда на стол одновременно. Ниже приводится пример, как с помощью методов `$q.when` и `$q.all` организовать это:

```
it('should illustrate promise aggregation with $q.when', function () {

    var ordersDelivered = $q.all([
        pizzaPit.takeOrder('Pepperoni'),
        $q.when('home made salad')
    ]);

ordersDelivered.then(pawel.eat, pawel.beHungry);

    pizzaPit.deliverOrder();
    expect($log.info.logs)
        .toContain(['Pawel is eating delicious Pepperoni,home
made salad']);
});
```

Метод `$q.when` возвращает отложенный результат, инициализированный значением его аргумента.

## Интеграция службы $q в AngularJS

Служба `$q` не только достаточно функциональная (и по-настоящему легковесная!) реализация Promise API, но очень тесно интегрирована с механизмом отображения в фреймворке AngularJS.

Во-первых, отложенный результат может быть передан непосредственно в контекст и автоматически отображен в пользовательском интерфейсе, как только будет получено фактическое значение. Это позволяет использовать отложенные результаты в качестве значений модели. Например, для следующего шаблона:

```
<h1>Hello, {{name}}!</h1>
```

и контроллера:

```
$scope.name = $timeout(function () {
    return "World";
}, 2000);
```

Текст «Hello, World!» появится, спустя две секунды без всякого вмешательства программиста.

Служба `$timeout` возвращает отложенный результат, который получит фактическое значение, возвращаемое функцией обратного вызова.

Несмотря на удобство такого подхода, его реализация в коде довольно сложно читается. Ситуация может стать еще более запутывающей, когда обнаруживается, что отложенный результат не отображается автоматически, если он возвращается функцией! Взгляните на следующий шаблон:

```
<h1>Hello, {{getName()}}!</h1>
```

и реализацию контроллера:

```
$scope.getName = function () {
    return $timeout(function () {
        return "World";
    }, 2000);
};
```

Этот код не выведет ожидаемый текст.

Мы не советуем использовать прием непосредственной передачи отложенных результатов в `$scope` и полагаться на автоматическое их отображение. Мы считаем, что он вносит путаницу, особенно из-за непоследовательности поведения, когда отложенный результат возвращается функцией.

# Promise API и служба $http

Теперь, после знакомства с отложенными результатами, мы можем сорвать покров таинственности с объектов ответов, возвращаемых методами службы `$http`. Если вы помните простой пример, приводившийся в начале главы, то вы должны вспомнить, что методы `$http`

возвращают объект, в котором можно зарегистрировать функции обратного вызова для обработки ситуаций успешного и неудачного выполнения запроса. В действительности, возвращаемый объект является самым настоящим объектом отложенного результата с двумя дополнительными вспомогательными методами: `success` и `error`. Как любой другой объект отложенного результата, он так же имеет метод `then`, с помощью которого можно зарегистрировать функции обратного вызова:

```
var responsePromise = $http.get('data.json');
responsePromise.then(function (response) {
    $scope.data = response.data;
},function (response) {
    throw new Error('Something went wrong...');
});
```

В отложенных результатах, возвращаемых службой `$http`, сохраняются объекты ответов, имеющие следующие свойства: `data`, `status`, `headers` и `config`.

Методы службы `$http` возвращают объекты отложенных результатов, которые обладают двумя дополнительными методами (`success` и `error`), чтобы упростить регистрацию функций обратного вызова.

Благодаря тому, что методы службы `$http` возвращают отложенные результаты, мы можем пользоваться всеми преимуществами Promise API для организации взаимодействий с серверной стороной: объединять функции обратного вызова, выстраивать цепочки запросов и предусматривать обработку ошибок любой сложности.

# Взаимодействие с конечными точками RESTful

**Передача репрезентативного состояния** (Representational State Transfer, REST) – популярное архитектурное решение для организации доступа к веб-службам. Интерфейс, предоставляемый службой `$http`, дает возможность без труда взаимодействовать с конечными точками RESTful из любых веб-приложений на основе AngularJS. Но разработчики фреймворка AngularJS сделали еще шаг вперед и реализовали службу `$resource`, еще больше упрощающую взаимодействия с конечными точками RESTful.

## Служба $resource

Конечные точки RESTful часто поддерживают операции CRUD, выполняемые в ответ на HTTP-запросы разных типов с похожими адресами URL. Писать код, взаимодействующий с такими конечными точками, часто достаточно просто, но утомительно. Служба `$resource` помогает избавиться от повторяющегося кода, пользоваться более высокоуровневыми абстракциями и интерпретировать операции с данными в терминах объектов (ресурсов) и вызовов методов, а не HTTP-запросов.

Реализация службы `$resource` находится в отдельном файле (`angular-resource.js`) и размещается в собственном модуле (`ngResource`). Чтобы воспользоваться службой `$resource`, необходимо подключить файл `angular-resource.js` и объявить зависимость прикладного модуля от модуля `ngResource`.

Чтобы увидеть, насколько просто с помощью службы `$resource` организовать взаимодействие с конечной точкой RESTful, реализуем слой абстракции для получения коллекции пользователей, возвращаемой веб-службой MongoLab:

```
angular.module('resource', ['ngResource'])
  .factory('Users', function ($resource) {
    return $resource(
      'https://api.mongolab.com/api/1/databases/ascrum/
collections/users/:id',
      {
        apiKey:'4fb51e55e4b02e56a67b0b66',
        id:'@_id.$oid'
      });
    })
```

Прежде всего нужно зарегистрировать рецепт (фабрику) для функции-конструктора `User`. Обратите внимание, что здесь не требуется писать код реализации конструктора. Его автоматически создаст служба `$resource`.

Служба `$resource` так же сгенерирует комплект методов, упрощающих взаимодействие с конечной точкой RESTful. Например, чтобы запросить список всех пользователей, достаточно выполнить вызов:

```
.controller('ResourceCtrl', function($scope, Users){
    $scope.users = Users.query();
});
```

При вызове метода `Users.query()` служба `$resource` сгенерирует код, вызывающий службу `$http`, и выполнит его. Когда в ответ сервер вернет строку в формате JSON, она будет преобразована в массив JavaScript с элементами типа `Users`.

При обращении к службе `$resource` возвращается сгенерированный конструктор, дополненный методами для взаимодействия с конечной точкой RESTful: `query`, `get`, `save` и `delete`.

Чтобы создать полноценный ресурс, фреймворку AngularJS требуется совсем немного информации. Давайте исследуем параметры метода `$resource` и посмотрим, какие исходные данные ему необходимы, и какие настройки нам доступны:

```
$resource(
    'https://api.mongolab.com/api/1/databases/ascrum/
collections/users/:id',
    {
        apiKey:'4fb51e55e4b02e56a67b0b66',
        id:'@_id.$oid'
    });
```

Первый аргумент – строка URL, точнее шаблон URL. Шаблон URL может содержать именованные параметры, начинающиеся с двоеточия. Мы можем определить только один шаблон, а это означает, что все HTTP-запросы должны иметь сходные URL.

Если для адресации серверной службы требуется указать номер порта, его следует экранировать в строке шаблона (например: `http://example.com\\:3000/api`). Это необходимо из-за того, что символ двоеточия в шаблонах URL имеет специальное значение.

Второй аргумент функции `$resource` позволяет определить параметры по умолчанию для передачи в каждом запросе. Имейте в виду, что здесь под термином «параметры» подразумеваются не только параметры, указанные в шаблоне URL, но и стандартные параметры, отправляемые в строке запроса. Фреймворк AngularJS попытается сначала выполнить подстановку параметров в шаблоне URL, а затем добавит остальные параметры в строку запроса URL.

Параметры по умолчанию могут быть либо статическими (указанными в вызове метода `factory`), либо динамическими, хранящимися в объекте ресурса. Значения динамических параметров должны предваряться символом `@`.

### Методы конструктора и методы экземпляра

Служба `$resource` автоматически генерирует два комплекта вспомогательных методов. Один комплект генерируется на уровне конструктора (класса) данного ресурса и используется для выполнения операций с коллекцией ресурсов или для обработки ситуации, когда не было создано ни одного экземпляра ресурса. Другой комплект предназначен для работы с конкретными экземплярами ресурса и служит для выполнения операций с одним ресурсом (одной записью в хранилище данных).

### Методы конструктора

Функция-конструктор, сгенерированная службой `$resource`, имеет комплект методов, соответствующих HTTP-запросам различных типов.

- `Users.query(params, successcb, errorcb)`: выполняет запрос HTTP GET и ожидает получить в ответ массив в формате JSON. Используется для получения коллекции элементов.
- `Users.get(params, successcb, errorcb)`: выполняет запрос HTTP GET и ожидает получить в ответ объект в формате JSON. Используется для получения единственного элемента.
- `Users.save(params, payloadData, successcb, errorcb)`: выполняет запрос HTTP POST, в теле которого передает данные из аргумента `payloadData`.
- `Users.delete(params,successcb, errorcb)` (и его псевдоним: `Users.remove`): выполняет запрос HTTP DELETE.

Все методы, перечисленные выше, принимают в аргументах `successcb` и `errorcb` функции обратного вызова для обработки успешного выполнения запроса или ошибки, соответственно. В аргументе `params` можно передавать параметры для данной конкретной операции – они будут использованы либо в качестве значений параметров шаблона URL, либо добавлены в строку запроса. Наконец, в аргументе `payloadData` можно указать данные для передачи в теле HTTP-запроса (POST и PUT).

### Методы экземпляра

В дополнение к методам конструктора, служба `$resource` генерирует также методы прототипа (экземпляра). Методы экземпляра действуют подобно одноименным методам класса, но оказывают влияние только на конкретный экземпляр. Например, удалить учетную запись пользователя можно либо вызовом метода конструктора:

```
Users.delete({}, user);
```

либо вызовом метода экземпляра:

```
user.$delete();
```

Методы экземпляра очень удобны в использовании и позволяют писать краткий и выразительный код управления ресурсами. Взгляните на следующий фрагмент, сохраняющий информацию о новом пользователе:

```
var user = new Users({
    name:'Superhero'
});
user.$save();
```

Его можно было бы записать иначе, применив метод конструктора:

```
var user = {
    name:'Superhero'
};
Users.save(user);
```

Фабричный метод службы `$resource` генерирует методы класса и методы экземпляра. Имена методов экземпляра начинаются с символа `$`. Соответствующие методы обоих типов обладают идентичной функциональностью, поэтому вы можете выбрать форму, наиболее удобную для вас.

### Собственные методы

По умолчанию метод `factory` службы `$resource` генерирует комплект методов, достаточных для большинства приложений. Однако, если серверная служба потребует использовать для некоторых операций HTTP-запросы других типов (например, PUT или PATCH), совсем несложно реализовать собственные методы на уровне ресурса.

По умолчанию фабричный метод службы `$resource` не генерирует методы, соответствующие запросам HTTP PUT. Если серверная служба предоставляет возможность выполнения некоторых операций в ответ на запросы HTTP PUT, вы можете добавить соответствующие методы вручную.

Например, MongoLab REST API в ответ на запрос HTTP POST создает новый элемент, а чтобы изменить существующий элемент,

следует использовать запрос HTTP PUT. Давайте посмотрим, как определить собственный метод `update` (на обоих уровнях, класса и экземпляра):

```
.factory('Users', function ($resource) {
  return $resource(
    'https://api.mongolab.com/api/1/databases/ascrum/
collections/users/:id',
    {
      apiKey:'4fb51e55e4b02e56a67b0b66',
      id:'@_id.$oid'
    }, {
      update: {method:'PUT'}
    });
  })
```

Как видите, новый метод определяется очень просто – достаточно передать функции `factory` третий аргумент, который должен быть объектом со следующей структурой:

```
action: {method:?, params:?, isArray:?, headers:?}
```

Ключ `action` – имя нового метода, который должен быть сгенерирован. Сгенерированный метод будет запускать HTTP-запрос указанного типа `method`, с параметрами по умолчанию `params`. Аргумент `isArray` определяет вид данных, возвращаемых в ответ на запрос – массив или единственный объект. Имеется также возможность указать собственные HTTP-заголовки в аргументе `headers`.

Служба `$resource` может принимать со стороны сервера только массивы и объекты JavaScript. Единственные значения (простых типов) не поддерживаются. Методы, возвращающие коллекции (отмеченные ключом `isArray`) должны возвращать массивы JavaScript. Массивы, заключенные в объекты, обрабатываются не так, как можно было бы ожидать.

### Расширение возможностей объектов ресурсов

Фабрика службы `$resource` генерирует функции-конструкторы, которые могут использоваться, как и любые другие конструкторы в JavaScript, для создания новых экземпляров ресурса с применением ключевого слова `new`. Но мы можем также расширять прототипы конструкторов, добавляя новые методы для экземпляров ресурсов. Представьте, что требуется добавить новый метод на уровне экземпляра, который выводил бы полное имя пользователя, исходя из имени и фамилии. Ниже демонстрируется рекомендуемый способ реализации такого метода:

```
.factory('Users', function ($resource) {
  var Users = $resource(
    'https://api.mongolab.com/api/1/databases/ascrum/
collections/users/:id',
    {
      apiKey:'4fb51e55e4b02e56a67b0b66',
      id:'@_id.$oid'
    }, {
      update: {method:'PUT'}
    });

  Users.prototype.getFullName = function() {
    return this.firstName + ' ' + this.lastName;
  };

  return Users;
})
```

Так же поддерживается возможность добавления новых методов класса (конструктора). Так как в языке JavaScript функции являются самыми обычными объектами, мы можем определять новые методы для функций-конструкторов. То есть, можно добавить собственные методы «вручную», не полагаясь на механизм автоматического создания методов. Это может пригодиться при необходимости реализовать нестандартную логику в одном из методов ресурса. Например, MongoLab REST API требует, чтобы идентификатор объекта исключался из передаваемых данных при выполнении HTTP-запросов методом `PUT`.

### Служба $resource создает асинхронные методы

Вернемся к примеру использования метода `query`:

```
$scope.users = Users.query();
```

У кого-то может сложиться впечатление, что сгенерированные ресурсы действуют синхронно (здесь мы не используем функции обратного вызова или отложенные результаты). В действительности же метод `query` является асинхронным, а чтобы его вызов выглядел как синхронный, AngularJS проделывает специальный трюк.

В данном случае метод `Users.query()` немедленно вернет пустой массив в качестве результата. Затем, когда асинхронный вызов завершится успехом и со стороны сервера поступят фактические данные, они будут сохранены в данном массиве. AngularJS просто сохранит ссылку на массив и заполнит его позднее. В результате содержимое массива изменится и шаблон, отображающий его, обновится автоматически.

Но не забывайте об асинхронной природе методов службы `$resource`. Подобная забывчивость часто становится источником ошибок, как в следующем фрагменте:

```
$scope.users = Users.query();
console.log($scope.users.length);
```

который работает не так как можно было бы ожидать!

К счастью в методах, сгенерированных службой `$resource`, всегда можно задействовать функции обратного вызова и переписать предыдущий фрагмент, как показано ниже:

```
Users.query(function(users){
    $scope.users = users;
    console.log($scope.users.length);
});
```

Методы, сгенерированные службой `$resource`, являются асинхронными, несмотря на то, что код (благодаря уловкам, реализованным внутри фреймворка AngularJS) выглядит как синхронный.

### Ограничения службы $resource

Служба `$resource` очень удобна и позволяет приступить к взаимодействиям с конечными точками RESTful, практически не затрачивая времени на реализацию. Но беда в том, что служба `$resource` слишком универсальна; она не учитывает специфических особенностей серверных веб-служб и опирается на некоторые допущения, которые могут оказаться ошибочными в том или ином случае.

Хорошо, когда служба `$resource` оказывается в состоянии взаимодействовать с серверной частью веб-приложения. Часто возможностей службы `$resource` оказывается вполне достаточно, но иногда предпочтительнее бывает использовать более низкоуровневую службу `$http`.

## Взаимодействия с веб-службами REST с помощью $http

Служба `$resource` очень удобна в использовании, но если вы столкнетесь со свойственными ей ограничениями, вы относительно легко сможете создать собственную службу, похожую на `$resource`, основанную на службе `$http`. В обмен на необходимость потратить некоторое время на реализацию собственной службы вы получите полный

контроль над предварительной и заключительной обработкой URL и данных. Как дополнительное преимущество, отпадает необходимость подключать файл `angular-resource.js` и тем самым сэкономить несколько килобайт трафика.

Ниже демонстрируется пример реализации службы для взаимодействия с MongoLab RESTful API. В изучении этого примера вам пригодится знание Promise API:

```
angular.module('mongolabResource', [])

    .factory('mongolabResource', function ($http, MONGOLAB_CONFIG) {

        return function (collectionName) {

            // основные настройки
            var collectionUrl =
                'https://api.mongolab.com/api/1/databases/' +
                MONGOLAB_CONFIG.DB_NAME +
                '/collections/' + collectionName;

            var defaultParams = {apiKey:MONGOLAB_CONFIG.API_KEY};

            // вспомогательные методы
            var getId = function (data) {
                return data._id.$oid;
            };

            // конструктор для создания новых ресурсов
            var Resource = function (data) {
                angular.extend(this, data);
            };

            Resource.query = function (params) {
                return $http.get(collectionUrl, {
                    params:angular.extend({q:JSON.stringify({}
|| params)},
                        defaultParams)
                }).then(function (response) {
                    var result = [];
                    angular.forEach(response.data, function
(value, key) {
                        result[key] = new Resource(value);
                    });
                    return result;
                });
            };

            Resource.save = function (data) {
                return $http.post(collectionUrl, data,
{params:defaultParams})
```

```
                    .then(function (response) {
                        return new Resource(data);
                    });
            };

            Resource.prototype.$save = function (data) {
                return Resource.save(this);
            };

            Resource.remove = function (data) {
                return $http.delete(collectionUrl + '',
defaultParams)
                    .then(function (response) {
                        return new Resource(data);
                    });
        };

            Resource.prototype.$remove = function (data) {
                return Resource.remove(this);
            };

            // здесь находится реализация других методов CRUD

            // дополнительные методы
            Resource.prototype.$id = function () {
                return getId(this);
            };
            return Resource;
    };
});
```

Пример начинается с объявления нового модуля (`mongolabResource`) и фабрики (`mongolabResource`), принимающей объект с настройками (`MONGOLAB_CONFIG`). Все это должно быть вам уже знакомо. На основе объекта с настройками можно подготовить URL для использования ресурсом. Здесь мы имеем полный контроль над строкой URL.

Далее объявляется конструктор `Resource`, с помощью которого можно создавать объекты ресурсов на основе имеющихся данных. Далее следуют определения нескольких методов: `query`, `save` и `remove`. Эти методы определены на уровне конструктора (класса), но точно так же можно объявить методы на уровне экземпляра, следуя тем же соглашениям, что приняты в оригинальной реализации службы `$resource`. Методы экземпляра могут делегировать выполнение операций соответствующим методам класса:

```
Resource.prototype.$save = function (data) {
    return Resource.save(this);
};
```

Использование приема составления цепочек из отложенных результатов составляет самую важную часть реализации ресурса. Метод `then` всегда вызывается относительно объекта отложенного результата, возвращаемого службой `$http`. Функция обратного вызова обработки успешного завершения запроса используется здесь для реализации логики постобработки. Например, в методе `query` реализована постобработка данных в формате JSON, возвращаемых серверной стороной, и создание экземпляров ресурсов. Здесь мы также получаем полный контроль над процессом извлечения данных из ответа.

Давайте посмотрим, как можно было бы использовать новую фабрику ресурсов:

```
angular.module('customResourceDemo', ['mongolabResource'])
    .constant('MONGOLAB_CONFIG', {
        DB_NAME: 'ascrum',
        API_KEY: '4fb51e55e4b02e56a67b0b66'
    })

    .factory('Users', function (mongolabResource) {
        return mongolabResource('users');
    })

    .controller('CustomResourceCtrl', function ($scope, Users,
Projects) {
        Users.query().then(function(users){
        $scope.users = users;
    });
});
```

Порядок использования собственных ресурсов ничем не отличается от использования службы `$resource`. Сначала необходимо объявить зависимость от модуля, где находится реализация фабрики ресурсов (`mongolabResource`). Затем следует передать параметры настройки в форме константы. И по завершении инициализации можно приступать к созданию фактических ресурсов – для этого достаточно вызвать фабричную функцию `mongolabResource` и передать ей имя коллекции MongoDB.

Новый конструктор ресурсов (здесь: `Users`) может внедряться, как любые другие зависимости и использоваться для вызова методов, класса или экземпляра. Следующий фрагмент демонстрирует вызов метода экземпляра:

```
$scope.addSuperhero = function () {
    new Users({name: 'Superhero'}).$save();
};
```

Самое большое преимущество собственной фабрики ресурсов, реализованной на основе службы `$http`, заключается в возможности использовать всю мощь Promise API.

# Дополнительные возможности $http

Служба `$http` – чрезвычайно гибкая и мощная. Ее мощь обусловлена ясным и гибким API, а также использованием Promise API. В этом разделе мы познакомимся с некоторыми дополнительными возможностями службы `$http`.

## *Обработка ответов*

Встроенная в AngularJS служба `$http` позволяет регистрировать обработчики, которые будут вызываться для каждого запроса. Такие обработчики могут пригодиться в ситуациях, когда требуется выполнить специальную обработку большинства (возможно всех) запросов.

Для примера предположим, что нам нужно организовать повторную отправку запросов, потерпевших неудачу. Для этого определим обработчик, который будет исследовать полученный ответ и пытаться повторить запрос, если в ответе получен код HTTP Service Unavailable (503). Ниже представлена одна из возможных реализаций обработчика:

```
angular.module('httpInterceptors', [])

    .config(function($httpProvider){
        $httpProvider.responseInterceptors.push('retryInterce
ptor');
    })

    .factory('retryInterceptor', function ($injector, $q) {

        return function(responsePromise) {
            return responsePromise.then(null,
function(errResponse) {
                if (errResponse.status === 503) {
                    return $injector.get('$http')(errResponse.
config);
                } else {
                    return $q.reject(errResponse);
```

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

Обработчик – это функция, принимающая отложенный результат оригинального запроса и возвращающая другой отложенный результат, инициализированный значением исходного отложенного результата. Здесь мы проверяем код `errResponse.status` и если в результате проверки определяется ошибочное состояние, после которого можно попытаться восстановить нормальную работу приложения, мы возвращаем отложенный результат, полученный в результате нового вызова `$http` с тем же конфигурационным объектом. Если обнаружена ошибка, восстановление после которой невозможно, мы просто разрешаем ей распространяться дальше (вызовом метода `$q.reject`).

Обработчики в AngularJS используют Promise API, и именно это делает их такими мощными. В примере выше, с помощью службы `$http`, выполняется повторный HTTP-запрос совершенно прозрачным для клиента способом. В главе 7 приводится более полный пример использования обработчиков в реализации механизма безопасности приложения.

Регистрация новых обработчиков выполняется достаточно просто и сводится к добавлению ссылки на новый обработчик (здесь служба AngularJS создается вызовом метода `factory`) в массив обработчиков, который управляется провайдером `$httpProvider`. Обратите внимание, что здесь для регистрации новых обработчиков используется провайдер, а провайдеры доступны только в блоках настройки.

# Тестирование кода, осуществляющего взаимодействия с помощью $http

Тестирование кода, обращающегося к внешним службам HTTP, часто представляет серьезную проблему из-за задержек в сети и непостоянства получаемых данных. Тогда как нам требуется, чтобы тесты выполнялись быстро и с предсказуемым результатом. К счастью в AngularJS имеются замечательные фиктивные объекты для имитации получения HTTP-ответов.

Служба `$http` в AngularJS зависит от другой, более низкоуровневой службы `$httpBackend`, которую можно считать тонкой оберткой вокруг объекта `XMLHttpRequest`. Эта обертка маскирует несовместимость браузеров и обеспечивает поддержку запросов JSONP.

Прикладной код никогда не должен вызывать службу `$httpBackend` непосредственно, потому что служба `$http` является более удачной абстракцией. Но наличие отдельной службы `$httpBackend` означает возможность ее подмены фиктивным объектом для нужд тестирования.

Чтобы познакомиться с фиктивным объектом, действующим в тестах взамен `$httpBackend`, рассмотрим модульный тест для простого контроллера, генерирующего запрос GET посредством службы `$http`. Ниже приводится код самого контроллера:

```
.controller('UsersCtrl', function ($scope, $http) {

    $scope.queryUsers = function () {
        $http.get('http://localhost:3000/databases/ascrum/
collections/users')

            .success(function (data, status, headers, config) {
                $scope.users = data;
            }).error(function (data, status, headers, config) {
                throw new Error('Something went wrong...');
            });
    };
});
```

Этот контроллер можно протестировать с помощью следующего кода:

```
describe('$http basic', function () {

    var $http, $httpBackend, $scope, ctrl;
    beforeEach(module('test-with-http-backend'));
    beforeEach(inject(function (_$http_, _$httpBackend_) {
        $http = _$http_;
        $httpBackend = _$httpBackend_;
    }));
    beforeEach(inject(function (_$rootScope_, _$controller_) {
        $scope = _$rootScope_.$new();
        ctrl = _$controller_('UsersCtrl', {
            $scope : $scope
        });
    }));
```

```
    it('should return all users', function () {

        // настройка ожидаемого запроса и ответа
        $httpBackend
            .whenGET('http://localhost:3000/databases/ascrum/
collections/users')
            .respond([{name: 'Pawel'}, {name: 'Peter'}]);

        // вызвать тестируемый код
        $scope.queryUsers();

        // имитировать ответ
        $httpBackend.flush();

        // проверить результат
        expect($scope.users.length).toEqual(2);
    });

    afterEach(function() {
        $httpBackend.verifyNoOutstandingExpectation();
        $httpBackend.verifyNoOutstandingRequest();
    });
});
```

Прежде всего обратите внимание, как фиктивный объект `$httpBackend` позволяет определять ожидаемые запросы (`whenGET(...)`) и ответы (`respond(...)`). Существует целое семейство методов `whenXXX`, по одному для каждого типа запросов HTTP. Методы семейства `whenXXX` имеют очень гибкую сигнатуру и позволяют определять адреса URL в виде регулярных выражений.

Для имитации данных, обычно возвращаемых серверной службой, можно использовать метод `respond`. Кроме всего прочего он позволяет также имитировать HTTP-заголовки.

Самой замечательной чертой фиктивного объекта `$httpBackend` является возможность полного контроля над содержимым ответов и временем их передачи. Благодаря методу `flush()` мы можем выбирать момент получения сымитированного HTTP-ответа. Модульные тесты, использующие фиктивный объект `$httpBackend`, могут действовать синхронно, даже при том, что сама служба `$http` имеет асинхронную организацию. Эти особенности обеспечивают предсказуемость модульных тестов и высокую скорость их выполнения.

Метод `verifyNoOutstandingExpectation` проверяет, все ли ожидаемые запросы были выполнены (вызваны методы `$http` и отправлены ответы), а метод `verifyNoOutstandingRequest` проверяет, не выполнил ли тестируемый код неожиданные запросы XHR. С помо-

щью этих двух методов можно убедиться, что тестируемый код вызывает все необходимые и только ожидаемые методы.

# В заключение

В этой главе мы исследовали различные способы взаимодействий с серверной стороной с целью получения данных. Сначала мы познакомились с прикладным интерфейсом службы `$http`, основного механизма выполнения запросов XHR и JSONP в AngularJS. При этом мы не просто рассмотрели API службы `$http`, но и детально изучили разные способы выполнения междоменных запросов.

Многие асинхронные службы в AngularJS в своей работе опираются на использование Promise API. Служба `$http` не является исключением и также в значительной степени основана на применении отложенных вычислений, из-за чего возникла необходимость детально изучить реализацию этого механизма в AngularJS. Мы узнали, что многоцелевой интерфейс Promise API реализован в виде службы `$q`, которая тесно интегрирована с механизмом отображения. Знание особенностей реализации Promise API в службе `$q` позволило нам в полной мере понять, как методы службы `$http` возвращают значения.

Фреймворк AngularJS легко настраивается на взаимодействие с конечными точками RESTful. В нем имеется специализированная фабрика `$resource`, значительно упрощающая реализацию подобных взаимодействий. Фабрика `$resource` очень удобна, но она слишком универсальна, из-за чего не всегда может соответствовать нашим потребностям. По этой причине не следует избегать создания собственных `$resource`-подобных фабрик, опирающихся на использование службы `$http`.

Ближе к концу главы мы рассмотрели некоторые дополнительные приемы использования службы `$http` и обработчиков HTTP-ответов.

Наконец, методы, использующие службу `$http`, должны подвергаться тестированию, как любой другой код на JavaScript, для чего AngularJS предоставляет замечательный фиктивный объект, упрощающий создание модульных тестов для проверки кода, взаимодействующего с сервером.

Когда все необходимые данные будут загружены клиентом и окажутся доступны программному коду на JavaScript, можно приступать к их отображению с применением различных средств AngularJS. Этой теме посвящена следующая глава, рассказывающая о шаблонах, директивах и механизмах отображения.

# ГЛАВА 4. Отображение и форматирование данных

Теперь, когда мы знаем, как получить данные со стороны сервера, можно сосредоточиться на вопросах обработки и отображения данных с применением средств фреймворка AngularJS. Эта глава начнется с обзора встроенных директив AngularJS, используемых в шаблонах. После знакомства с основами мы посмотрим как действуют разные директивы и обсудим варианты их применения. А затем погрузимся в изучение фильтров AngularJS.

В этой главе вы познакомитесь с:

- соглашениями по именованию директив в AngularJS;
- приемами отображения или сокрытия блоков разметки исходя из некоторого условия;
- особенностями применения директивы повторения (`ng-repeat`);
- возможностью регистрации обработчиков событий DOM, чтобы пользователи могли взаимодействовать с приложением;
- ограничениями языка шаблонов фреймворка AngularJS и возможными обходными решениями;
- фильтрами: их назначением и примерами использования; мы пройдемся по встроенным фильтрам, а также посмотрим, как создавать и тестировать собственные фильтры.

## Знакомство с директивами

Прежде чем приступать к рассмотрению примеров использования встроенных директив AngularJS, необходимо познакомиться с различными соглашениями по именованию директив и обращению к ним в разметке HTML.

В документации с описанием AngularJS имена всех директив указаны в «верблюжьей» нотации (например, `ngModel`). В шаблонах, однако, требуется использовать нотацию, в которой используются только символы нижнего регистра, а слова и префиксы в именах отделяются друг от друга символом дефиса (`ng-model`), двоеточием (`ng:model`) или символом подчеркивания (`ng_model`). Кроме того, ссылки на директивы можно предварять символом `x` или словом `data`.

К любой директиве можно обратиться с использованием множества разных (но эквивалентных) имен. Если в качестве примера взять директиву `ngModel`, для обращения к ней в шаблонах можно использовать имена: `ng-model`, `ng:model`, `ng_model`, `x-ng-model`, `x-ng:model`, `x-ng_model`, `data-ng-model`, `data-ng:model`, `data-ng_model`, `x:ng-model`, `data_ng-model` и так далее.

Префикс `data` очень удобно использовать, чтобы придать HTML-документам совместимость со стандартом HTML5 и обеспечить прохождение проверок валидаторами HTML5. Если перед вами не стоит такой цели, можно выбрать любую схему именования – все они полностью эквивалентны.

Во всех примерах в этой книге используется форма записи с дефисом (`ng-model`). На наш взгляд это наиболее краткий и читаемый синтаксис.

# Отображение результатов вычисления выражений

Фреймворк AngularJS поддерживает разные способы отображения данных. Но конечный результат всегда один и тот же – содержимое модели отображается на экране. Однако имеется ряд тонкостей, о которых следует упомянуть.

## Директива интерполяции

**Директива интерполяции** является самой основной директивой отображения данных модели. Она принимает выражение, ограниченное парами фигурных скобок:

```
<span>{{expression}}</span>
```

вычисляет выражение и отображает результат на экране.

Ограничители, используемые в шаблонах, являются настраиваемыми – это может пригодиться, если вы собираетесь смешивать AngularJS с другими языками шаблонов на стороне сервера. Настройка ограничителей выполняется очень просто и сводится к установке значений свойств в `$interpolateProvider`:

```
myModule.config(function($interpolateProvider) {
    $interpolateProvider.startSymbol('[[');
    $interpolateProvider.endSymbol(']]');
});
```

Здесь мы заменили ограничители `{{}}` на `[[]]` и теперь можем использовать в шаблонах синтаксис:

```
[[expression]]
```

## Отображение значений с помощью ngBind

Директива интерполяции имеет эквивалентную форму: `ng-bind`, которую можно использовать как HTML-атрибут:

```
<span ng-bind="expression"></span>
```

Форма с фигурными скобками проще в использовании, но иногда форма `ng-bind` оказывается удобнее. Обычно она используется, чтобы скрыть выражения до момента, когда AngularJS получит возможность обработать их на этапе начальной загрузки страницы. Это помогает избавиться от эффекта мерцания элементов пользовательского интерфейса и усилить благоприятные впечатления у пользователя. Тема оптимизации начальной загрузки страницы более подробно рассматривается в главе 12.

## Включение разметки HTML в выражения

По умолчанию AngularJS экранирует все элементы разметки HTML, присутствующие в выражениях, (моделях) вычисляемых директивой интерполяции. Например, если имеется модель:

```
$scope.msg = 'Hello, <b>World</b>!';
```

и разметка:

```
<p>{{msg}}</p>
```

механизм отображения выполнит экранирование тегов `<b>` так, что они будут отображаться как простой текст, а не как разметка:

```
<p>Hello, &lt;b&gt;World&lt;/b&gt;!</p>
```

Директива выполняет экранирование разметки HTML, чтобы помочь предотвратить нападения типа «инъекция HTML».

Если по каким-то причинам разметка HTML, содержащаяся в модели, должна интерпретироваться и отображаться браузером именно как разметка, используйте директиву `ng-bind-html-unsafe`, отключающую операцию экранирования тегов HTML:

```
<p ng-bind-html-unsafe="msg"></p>
```

Применив директиву `ng-bind-html-unsafe`, мы получим фрагмент HTML с тегами `<b>`.

Будьте осторожны, применяя директиву `ng-bind-html-unsafe`. Используйте ее только с содержимым, не вызывающим сомнений, или с выражениями, целиком и полностью находящимися под вашим контролем. В противном случае злонамеренный пользователь сможет внедрить в вашу страницу любую разметку HTML.

В AngularJS существует еще одна директива, которая выборочно экранирует некоторые теги HTML: `ng-bind-html`. Она используется подобно директиве `ng-bind-html-unsafe`:

```
<p ng-bind-html="msg"></p>
```

В терминах экранирования, директиву `ng-bind-html` можно рассматривать как компромисс между директивой `ng-bind-html-unsafe` (позволяющей использовать любые теги HTML) и директивой интерполяции (экранирующей все теги HTML). Она служит отличной альтернативой в случаях, когда необходимо использовать некоторые теги HTML, вводимые пользователем.

Директива `ng-bind-html` находится в отдельном модуле (`ngSanitize`) и для ее использования требуется подключить дополнительный исходный файл: `angular-sanitize.js`.

Не забудьте объявить зависимость от модуля `ngSanitize`, если собираетесь использовать директиву `ng-bind-html`:

```
angular.module('expressionsEscaping', ['ngSanitize'])
    .controller('ExpressionsEscapingCtrl', function ($scope) {
        $scope.msg = 'Hello, <b>World</b>!';
    });
```

Если только вы не собираетесь использовать уже находящиеся в эксплуатации системы (CMS, серверные компоненты, передающие клиенту разметку HTML, и так далее), избегайте включения размет-

ки в свои модели. Такая разметка не может содержать директивы AngularJS и требует применения директивы `ng-bind-html-unsafe` или `ng-bind-html` для получения желаемого результата.

# Отображение по условию

Отображение и сокрытие элементов DOM, исходя из некоторых условий, является вполне распространенным требованием. В состав AngularJS входит четыре разных набора директив для этой цели (`ng-show`/`ng-hide`, `ng-switch-*`, `ng-if` и `ng-include`).

Семейство директив `ng-show`/`ng-hide` можно использовать для сокрытия (применяя CSS-правило `display`) элементов дерева DOM, опираясь на результат вычисления выражения:

```
<div ng-show="showSecret">Secret</div>
```

Предыдущий фрагмент можно переписать, задействовав в нем директиву `ng-hide`:

```
<div ng-hide="!showSecret">Secret</div>
```

Директивы `ng-show`/`ng-hide` достигают желаемого эффекта простым применением стиля `style="display: none;"`, когда требуется скрыть элементы DOM. Сами элементы при этом не удаляются из дерева.

Если понадобится физически удалить или добавить узлы в дерево DOM, используйте семейство директив `ng-switch` (`ng-switch`, `ng-switch-when`, `ng-switch-default`):

```
<div ng-switch on="showSecret">
    <div ng-switch-when="true">Secret</div>
    <div ng-switch-default>Won't show you my secrets!</div>
</div>
```

Директива `ng-switch` своим действием близко напоминает инструкцию `switch` в языке JavaScript, и может включать несколько экземпляров `ng-switch-when`.

Главное отличие директивы `ng-switch` от директив `ng-show`/`ng-hide` заключается в способе обработки элементов DOM. Директива `ng-switch` добавляет/удаляет элементы DOM, тогда как директивы `ng-show`/`ng-hide` просто применяют стиль `style="display: none;"` для сокрытия элементов. Кроме того, директива `ng-switch` создает новый контекст.

Директивы ng-show/ng-hide просты в использовании, но могут отрицательно сказываться на производительности, при применении к большому количеству элементов DOM. При обнаружении проблем с производительностью, связанных с размером дерева DOM, подумайте о возможности использовать директивы семейства ng-switch.

Проблема семейства директив ng-switch заключается в том, что они способствуют чрезмерному разбуханию кода даже в самых простых случаях. К счастью в арсенале AngularJS имеется еще одна директива: ng-if. Она действует подобно директиве ng-switch (в том смысле, что добавляет/удаляет элементы дерева DOM) но имеет более краткий синтаксис:

```
<div ng-if="showSecret">Secret</div>
```

Директива ng-if доступна только в самых последних версиях AngularJS: 1.1.x и 1.2.x.

## Включение блоков содержимого по условию

Директива ng-include, хотя и не является прямым эквивалентом инструкции if/else, также может использоваться для динамического отображения блоков содержимого по условию. Данная директива имеет одну весьма примечательную особенность – она способна загружать и отображать фрагменты содержимого, опираясь на результат вычисления выражения. Это дает возможность легко создавать очень динамичные страницы. Например, с ее помощью можно включать различные формы, в зависимости от роли, присвоенной пользователю. Следующий фрагмент загружает разные фрагменты для обычных пользователей и пользователей, наделенных привилегиями администратора:

```
<div ng-include="user.admin && 'edit.admin.html' || 'edit.user.html'">
</div>
```

Директива ng-include создает новый контекст для каждого включенного с ее помощью фрагмента.

Кроме того, директива ng-include является отличным средством компоновки страниц из небольших фрагментов разметки.

Директива `ng-include` принимает выражение, поэтому, если вы собираетесь использовать фиксированные значения, ей должна передаваться строка в кавычках, указывающая на фрагмент, например: `<div ng-include="'header.tpl.html'"></div>`.

# Отображение коллекций с помощью директивы ngRepeat

Директива `ng-repeat` является, пожалуй, одной из самых часто используемых и самых мощных директив. Она выполняет итерации по элементам коллекции и для каждого из них создает новый элемент DOM. Но директива `ng-repeat` обладает гораздо более широкими возможностями, чем простое отображение элементов коллекций. Она постоянно следит за источником данных и повторно отображает шаблон, обнаружив какие-либо изменения.

Директива повторения имеет высоко оптимизированную реализацию и стремится минимизировать количество изменений в дереве DOM при отображении данных.

Внутренняя реализация `ng-repeat` может перемещать узлы дерева DOM с места на место (при перемещении элемента в исходном массиве), удалять их (при удалении соответствующих элементов из массива) и вставлять новые узлы (при включении новых элементов в массив). Но, независимо от стратегий, выбираемых внутренней реализацией, важно понять, что эта директива не просто цикл `for`, который выполняется однократно. Она действует подобно механизму наблюдения за данными, который пытается своевременно отобразить исходную коллекцию с данными в коллекцию узлов DOM. Процесс наблюдения за данными протекает непрерывно.

## Знакомство с директивой ngRepeat

В простейшем случае использования директива `ng-repeat` имеет несложный синтаксис:

```
<table class="table table-bordered">
  <tr ng-repeat="user in users">
    <td>{{user.name}}</td>
    <td>{{user.email}}</td>
```

```
    </tr>
  </table>
```

Здесь массив `users` определен в текущем контексте и содержит типичные объекты с информацией о пользователях в виде таких свойств, как: `name`, `email` и так далее. Директива `ng-repeat` выполнит итерации по элементам коллекции `users` и для каждого из них создаст элемент `<tr>`.

Директива создает новый контекст для каждого элемента коллекции.

## Специальные переменные

Инструкция повторения в AngularJS объявляет множество специальных переменных в контексте, создаваемом для каждого элемента коллекции. Эти переменные могут использоваться для определения позиции элемента в коллекции:

- `$index`: индекс элемента в коллекции (нумерация начинается с 0);
- `$first`, `$middle`, `$last`: логическое значение, соответствующее позиции элемента.

Упомянутые переменные могут с успехом использоваться во многих ситуациях. Например, в примере приложения SCRUM, переменную `$last` можно использовать для отображения ссылок в элементе навигации. Последняя (текущая) часть пути не должна отображаться как ссылка, тогда как остальные элементы пути должны быть оформлены как элементы `<a>`, как показано на рис. 4.1.

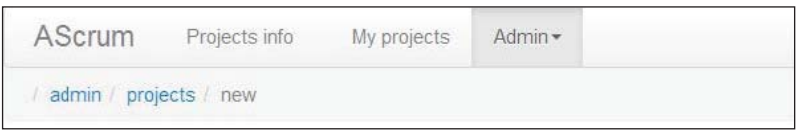


**Рис. 4.1.** Последний элемент пути не должен отображаться как ссылка

Смоделировать такой пользовательский интерфейс можно с помощью следующего кода:

```
<li ng-repeat="breadcrumb in breadcrumbs.getAll()">
    <span class="divider">/</span>
    <ng-switch on="$last">
```

```
            <span ng-switch-when="true">{{breadcrumb.name}}</span>
            <span ng-switch-default>
                <a href="{{breadcrumb.path}}">{{breadcrumb.name}}</a>
            </span>
        </ng-switch>
    </li>
```

## Итерации по свойствам объекта

Обычно директива `ng-repeat` используется для отображения элементов массивов JavaScript. Однако ее можно также использовать для обхода свойств объектов. В этом случае ее синтаксис немного отличается:

```
<li ng-repeat="(name, value) in user">
    Property {{$index}} with {{name}} has value {{value}}
</li>
```

Этот пример отображает все свойства объекта `user` в виде маркированного списка. Обратите внимание, что необходимо указывать переменные для хранения имени и значения свойств, используя форму записи с круглыми скобками `(name, value)`.

Перед выводом результатов директива `ng-repeat` отсортирует их в алфавитном порядке по именам свойств. Это поведение нельзя изменить, поэтому в данном случае нет никаких средств управления порядком итераций.

Переменную `$index` все так же можно использовать для получения номера позиции свойства в отсортированном списке всех свойств.

Механизм итераций по свойствам объектов имеет некоторые ограничения. Главная проблема в отсутствии возможности управления порядком итераций.

Если порядок итераций по свойствам имеет значение, их следует отсортировать в нужном порядке в контроллере и поместить в массив.

## Приемы использования директивы ngRepeat

В этом разделе мы рассмотрим некоторые наиболее типичные варианты отображения и способы их реализации с помощью AngularJS.

В частности, мы рассмотрим списки с подробной информацией и приемы применения классов CSS к элементам списка.

## Списки

Очень часто используется прием вывода списков, элементы которых могут распахиваться щелчком мыши для отображения дополнительных сведений. Этот прием имеет два варианта: когда распахиваться может единственный элемент или несколько элементов. Скриншот на рис. 4.2 иллюстрирует такой пользовательский интерфейс.

| Name | e-mail |
|---|---|
| Pawel | pawel@domain.com |
| Pawel details go here... | |
| Peter | peter@domain.com |

**Рис. 4.2.** Раскрывающийся элемент списка

## Отображение единственной строки с дополнительной информацией

Распахивание единственного элемента легко можно реализовать, как показано ниже:

```
<table class="table table-bordered"  ng-controller="ListAndOn
eDetailCtrl">

  <tbody ng-repeat="user in users" ng-click="selectUser(user)"
         ngswitch-on="isSelected(user)">
    <tr>
      <td>{{user.name}}</td>
      <td>{{user.email}}</td>
    </tr>
    <tr ng-switch-when="true">
      <td colspan="2">{{user.desc}}</td>
    </tr>
  </tbody>
</table>
```

Здесь дополнительная строка с более подробной информацией о пользователе отображается, только если данный пользователь был выбран. Процедура выбора очень проста и реализуется функциями `selectUser` и `isSelected`:

```
.controller('ListAndOneDetailCtrl', function ($scope, users) {
    $scope.users = users;

    $scope.selectUser = function (user) {
```

```
        $scope.selectedUser = user;
    };

    $scope.isSelected = function (user) {
        return $scope.selectedUser === user;
    };
})
```

Эти две функции используют тот факт, что существует единый контекст (определяется в начале элемента DOM таблицы), где можно сохранить указатель (`selectedUser`) на активный элемент списка.

## Отображение множества строк с дополнительной информацией

Если потребуется обеспечить возможность распахивания нескольких строк с дополнительной информацией, необходимо изменить стратегию. На этот раз сведения о выборе следует сохранять на уровне элементов. Как вы наверняка помните, директива `ng-repeat` создает новый контекст для каждого элемента коллекции. Мы можем воспользоваться этим обстоятельством и сохранять состояние выбора в каждом элементе:

```
<table class="table table-bordered">
  <tbody ng-repeat="user in users" ng-controller="UserCtrl"
         ng-click="toggleSelected()" ng-switch on="isSelected()">
    <tr>
      <td>{{user.name}}</td>
      <td>{{user.email}}</td>
    </tr>
    <tr ng-switch-when="true">
      <td colspan="2">{{user.desc}}</td>
    </tr>
  </tbody>
</table>
```

Этот пример отличается тем, что в нем директива `ng-controller` применяется к каждому элементу. Используемый контроллер может расширять объект контекста дополнительными функциями и свойствами для управления состоянием выбора:

```
.controller('UserCtrl', function ($scope) {

    $scope.toggleSelected = function () {
        $scope.selected = !$scope.selected;
    };

    $scope.isSelected = function () {
        return $scope.selected;
```

```
    };
});
```

Запомните, если контроллер определяется на уровне того же элемента DOM, что и директива `ng-repeat`, он будет действовать в рамках нового контекста, создаваемого директивой повторения. С практической точки зрения это означает, что в контроллере можно реализовать логику управления отдельными элементами коллекции. Это очень мощный прием, дающий возможность инкапсулировать переменные и функции для работы с отдельными элементами.

## Изменение таблиц, строк и применение классов CSS

Для улучшения читаемости, к спискам часто применяют прием оформления чередования цвета. В AngularJS имеется пара директив (`ngClassEven` и `ngClassOdd`) делающих решение этой задачи тривиальным делом:

```
<tr ng-repeat="user in users"
    ng-class-even="'light-gray'" ng-class-odd="'dark-gray'">
  ...
</tr>
```

Директивы `ngClassEven` и `ngClassOdd` являются всего лишь специализированными версиями более универсальной директивы `ngClass`, которая отличается большой гибкостью и может применяться в самых разных ситуациях. Для демонстрации перепишем предыдущий пример, как показано ниже:

```
<tr ng-repeat="user in users"
    ng-class="{'dark-gray' : !$index%2, 'light-gray' :
$index%2}">
```

Здесь директиве `ngClass` в качестве аргумента передается объект. Ключами этого объекта являются имена классов, а значениями – условные выражения. Класс, определяемый ключом, будет добавлен в элемент или удален из него, в зависимости от значения соответствующего выражения.

Директива `ngClass` может также принимать строковые аргументы и массивы. В обоих этих случаях аргументы могут содержать список классов CSS (в строках – разделенных запятыми) для добавления в данный элемент.

# Обработчики событий DOM

Пользовательский интерфейс не особенно полезен, если не дает пользователю возможность взаимодействовать с ним (с помощью мыши, клавиатуры или путем касаний сенсорного экрана). Для тех, кого волнует эта проблема, AngularJS приготовил приятный сюрприз, превратив регистрацию обработчиков событий в детскую забаву! Например, ниже приводится реализация реакции на щелчок мыши:

```
<button ng-click="clicked()">Click me!<button>
```

Выражение `clicked()` вычисляется относительно текущего объекта контекста `$scope`, что дает возможность вызвать любой метод этого объекта. При этом мы не ограничены простыми вызовами функций; допускается использовать выражения любой сложности, включая выражения с аргументами:

```
<input ng-model="name">
<button ng-click="hello(name)">Say hello!<button>
```

Разработчики, только начинающие осваивать AngularJS, часто пытаются регистрировать обработчики событий, как `ng-click="{{clicked()}}"` или `ng-click="sayHello({{name}})"`, то есть, используя внутри значения атрибута директиву интерполяции. В этом нет необходимости, и к тому же такой способ приведет к ошибочным результатам, потому что в этом случае AngularJS вычислит директиву интерполяции в процессе обработки дерева DOM и полученный результат использует в качестве обработчика события!

В AngularJS имеются встроенные директивы поддержки самых разных событий:

- **события щелчка мышью**: `ngClick` и `ngDblClick`;
- **события мыши**: `ngMousedown`, `ngMouseup`, `ngMouseenter`, `ngMouseleave`, `ngMousemove` и `ngMouseover`;
- **события клавиатуры**: `ngKeydown`, `ngKeyup` и `ngKeypress`;
- **события ввода** (`ngChange`): директива `ngChange` действует совместно с директивой `ngModel` и позволяет реализовать реакцию на изменения в модели, вызванные действиями пользователя.

Упомянутые обработчики событий DOM могут принимать специальный аргумент `$event`, представляющий низкоуровневое событие DOM. Это обеспечивает доступ к низкоуровневым свойствам собы-

тий, позволяет предотвратить выполнение действий, предусмотренных для данного события по умолчанию, остановить распространение события, и так далее. Например, ниже показано, как определить позицию элемента, на котором выполнен щелчок:

```
<li ng-repeat="item in items" ng-click="logPosition(item, $event)">
    {{item}}
</li>
```

где функция `logPosition` определена в объекте контекста, как показано ниже:

```
$scope.readPosition = function (item, $event) {
    console.log(item + ' was clicked at: ' + $event.clientX + ',' +
                $event.clientY);
};
```

Несмотря на то, что специальная переменная `$event` передается обработчикам событий, ее не следует использовать для дорогостоящих манипуляций с деревом DOM. Как уже говорилось в главе 1, при создании пользовательского интерфейса следует придерживаться декларативного подхода и выполнять манипуляции с деревом DOM только в директивах. Именно поэтому аргумент `$event` часто используется только в коде директив.

# Увеличение эффективности с помощью шаблонов на основе DOM

Очень редко встречаются системы шаблонов, использующие «живое» дерево DOM и разметку HTML, но, как оказывается, такой подход на удивление эффективен. Мы привыкли использовать другие механизмы шаблонов, основанные на строках, но стоит попробовать использовать шаблоны на основе DOM, как они быстро превращаются в основной инструмент. Нужно лишь помнить некоторые его недостатки.

## *Избыточный синтаксис*

Во-первых, синтаксис некоторых конструкций может оказаться несколько избыточным. Примером может служить раздражающий синтаксис семейство директив `ng-switch`, требующих от разработчика

ввести массу кода даже в самых простых случаях использования. Рассмотрим простой пример отображения разных сообщений, в зависимости от количества элементов в списке:

```
<div ng-switch on="items.length>0">
    <span ng-switch-when="true">
        There are {{items.length}} items in the list.
    </span>
    <span ng-switch-when="false">
        There are no items in the list.
    </span>
</div>
```

Конечно, логику подготовки текста сообщения можно было бы переместить в контроллер, чтобы избавиться от инструкции `switch` в шаблоне, но подобная простая логика выбора более уместна в слое реализации отображения.

К счастью в последних версиях AngularJS появилась встроенная директива `ng-if`, очень удобная в ситуациях, когда не требуется вся мощь конструкции множественного выбора.

## Применение директивы ngRepeat к множеству элементов DOM

Несколько более серьезная проблема связана с тем, что в простейшей своей форме, директива повторения `ng-repeat` способна выполнять итерации только с одним элементом (вместе с его потомками). Это означает, что директива `ng-repeat` не способна управлять группой братских элементов.

Для иллюстрации проблемы предположим, что имеется список, каждый элемент которого содержат название и описание. Допустим также, что нам требуется вывести этот список в виде таблицы, где названия и описания должны находиться в разных строках (`<tr>`). Чтобы решить эту задачу, необходимо добавить тег `<tbody>`, только чтобы поместить в него директиву `ng-repeat`:

```
<table>
    <tbody ng-repeat="item in items">
        <tr>
            <td>{{item.name}}</td>
        </tr>
        <tr>
            <td>{{item.description}}</td>
        </tr>
    </tbody>
</table>
```

Потребность директивы ng-repeat в контейнерном элементе вынуждает нас создать определенную структуру HTML. Это может оказаться проблематичным, в зависимости от того, насколько строго требуется придерживаться структуры разметки, разработанной дизайнером.

В предыдущем примере проблема стоит не так остро, потому что у нас в запасе оказался соответствующий контейнер HTML (<tbody>), но иногда ситуация складывается так, что у нас нет допустимого элемента HTML, куда можно было бы поместить директиву ng-repeat. Например, в следующем примере было бы желательно получить на выходе такую разметку HTML:

```
<ul>
    <!-- директиву повторения было бы желательно поместить сюда -->
    <li><strong>{{item.name}}</strong></li>
    <li>{{item.description}}</li>
    <!-- и чтобы ее действие заканчивалось здесь -->
</ul>
```

В грядущей версии AngularJS (1.2.x) базовый синтаксис директивы ngRepeat будет расширен возможностью выбора элементов DOM, участвующих в итерациях. В этой версии появится возможность писать такой код:

```
<ul>
    <li ng-repeat-start="item in items">
        <strong>{{item.name}}</strong>
    </li>
    <li ng-repeat-end>{{item.description}}</li>
</ul>
```

С помощью атрибутов ng-repeat-start и ng-repeat-end станет возможно определять группы братских элементов DOM, участвующих в итерациях.

## Элементы и атрибуты не могут изменяться во время выполнения

Так как фреймворк AngularJS оперирует «живым» деревом DOM в браузере, он может делать только то, что позволит браузер. Оказывается, что в некоторых ситуациях отдельные браузеры отвергают любые попытки изменения элементов и их атрибутов, предпринимаемые после того, как эти элементы будут вставлены в дерево DOM.

Чтобы увидеть следствия этих ограничений на примере, рассмотрим достаточно типичный случай динамического изменения атрибу-

та `type` элемента ввода. Многие разработчики пытаются (безуспешно!) писать такой код:

```
<input type="{{myinput.type}}" ng-model="myobject[myinput.
model]">
```

Проблема в том, что некоторые браузеры (да! вы угадали, это Internet Explorer!) не позволяют изменить тип созданного элемента ввода. Такие браузеры интерпретируют выражение `{{myinput.type}}` (не вычисленное) как тип элемента, а поскольку этот тип не известен браузеру, тип интерпретируется как `type="text"`.

Существует несколько способов решения проблемы, описанной выше, но прежде чем обсуждать их, нам необходимо познакомиться с приемами создания собственных директив. Одно из возможных решений приводится в главе 9. Еще одно простое решение основано на применении встроенной директивы `ng-include`, инкапсулирующей статические шаблоны с элементами ввода разных типов:

```
<ng-include src="'input'+myinput.type+'.html'"></ng-include>
```

где подключаемый фрагмент определяется статической строкой.

При использовании этого приема обращайте внимание на проблемы, связанные с контекстом, так как директива `ng-include` создает новый контекст.

### Нестандартные элементы HTML и старые версии IE

В заключение хочется напомнить, что нестандартные элементы и атрибуты HTML плохо поддерживаются версиями Internet Explorer 8 и ниже. Чтобы получить полную отдачу от директив AngularJS в IE8 и IE7, требуется предпринять дополнительные шаги, которые подробно описаны в главе 12, посвященной сценариям развертывания приложений.

## Преобразование моделей с помощью фильтров

Выражения AngularJS могут иметь очень сложный вид и содержать вызовы функций. Эти функции могут служить разным целям, из ко-

торых наиболее типичными являются преобразование и форматирование моделей. Для достижения этих типичных целей выражения AngularJS поддерживают специальные функции форматирования (преобразования), называемые фильтрами:

```
{{user.signedUp| date:'yyyy-MM-dd'}}
```

В данном примере `date` – это фильтр, используемый для форматирования даты регистрации пользователя.

Фильтры являются не более чем глобальными именованными функциями, которые вызываются в представлении с помощью символа конвейера (|) и с параметрами, отделенными от имени функции двоеточием (:). Фактически, строку выше можно переписать иначе (при условии, что функция `formatDate` определена в объекте контекста):

```
{{formatDate(user.signedUp, 'yyyy-MM-dd')}}
```

Применение фильтров дает двойное преимущество: они не требуют добавлять функции в объект контекста (так как по умолчанию доступны в любом шаблоне) и позволяют использовать синтаксис, более выразительный, чем синтаксис вызова функции.

Этот простой пример также показывает, что фильтры могут быть параметризованными (иными словами, могут принимать аргументы): здесь фильтру `date` передается аргумент с форматом представления даты.

Некоторые фильтры могут объединяться в цепочки для реализации конвейерных преобразований. Например, ниже показано, как ограничить длину строки 80 символами и преобразовать их в нижний регистр:

```
{{myLongString | limitTo:80 | lowercase}}
```

## Применение встроенных фильтров

В состав базовой библиотеки AngularJS входит несколько встроенных фильтров. Вообще говоря, встроенные фильтры можно разделить на две группы: фильтры форматирования и фильтры преобразования массивов.

### Фильтры форматирования

Ниже следует список фильтров форматирования, назначение и область применения которых легко определяется по их названиям:

- `currency`: форматирует числа с двумя десятичными знаками после запятой и символом валюты;
- `date`: форматирует дату в соответствии с указанным форматом; модель может содержать дату в виде объекта `Date` или строки (строка сначала будет преобразована в объект `Date` и только потом отформатирована);
- `number`: форматирует исходное число, оставляя число десятичных знаков, указанное в аргументе;
- `lowercase` и `uppercase`: как следует из их имен, эти фильтры можно использовать для преобразования символов в строках в нижний или верхний регистр;
- `json`: этот фильтр чаще применяется при отладке, так как позволяет выводить объекты JavaScript в удобочитаемом виде; обычно используется в виде: `<pre>{{someObject | json}}</pre>`.

### Фильтры преобразования массивов

По умолчанию в состав AngularJS входит три фильтра для работы с массивами:

- `limitTo`: возвращает массив, уменьшенный до указанного размера; позволяет сохранять элементы в начале или в конце коллекции (в последнем случае в аргументе фильтру должно передаваться отрицательное число);
- `filter`: многоцелевая и очень гибкая утилита, поддерживает множество параметров настройки отбора элементов коллекций;
- `orderBy`: фильтр сортировки, может использоваться для упорядочения элементов в массиве, опираясь на указанный критерий.

Перечисленные фильтры работают только с массивами (исключение составляет фильтр `limitTo`, способный также работать со строками). При применении к объектам, отличным от массивов, эти фильтры не выполняют ни каких действий и просто возвращают исходный объект.

Фильтры из этой группы часто используются совместно с директивой `ng-repeat` для отображения отфильтрованных результатов. В следующих разделах мы создадим полноценный пример таблицы с поддержкой сортировки, фильтрации и постраничного просмотра. Примеры иллюстрируют работу со списком заданий в приложении

SCRUM и наглядно показывают, как можно объединять фильтры и директиву повторения.

### Фильтрация с помощью filter

Сначала познакомимся поближе с фильтром `filter`. Имя фильтра `filter` выбрано несколько неудачно, так как слово «filter» может обозначать вообще любой фильтр (функцию преобразования).

Фильтр `filter` – универсальная функция фильтрации, которая может использоваться для выбора подмножества элементов массива (или, говоря иначе, для исключения некоторых элементов). Этот фильтр может принимать несколько параметров, управляющих процессом выбора элементов. В простейшем случае можно передать фильтру строку, и он оставит в массиве только элементы, содержащие эту строку.

В качестве примера рассмотрим реализацию вывода списка заданий, который можно фильтровать на основе некоторого критерия. Пользователям будет предоставлено поле ввода, где они смогут вводить свои критерии, а результирующий список должен включать только элементы, в которых хотя бы одно поле содержит указанную подстроку. Скриншот на рис. 4.3 демонстрирует пользовательский интерфейс, который должен получиться.

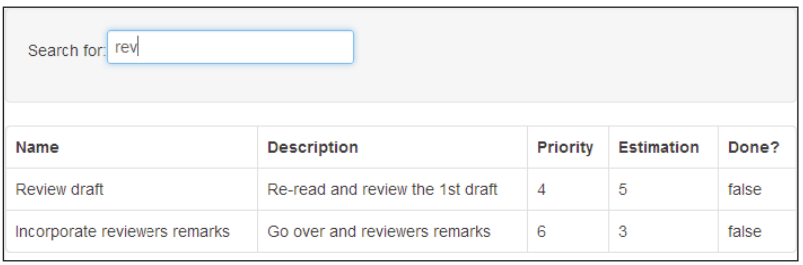


**Рис. 4.3.** Список с поддержкой фильтрации

Если предположить, что модель имеет следующие свойства: `name`, `desc`, `priority`, `estimation` и `done`, шаблон пользовательского интерфейса можно было бы реализовать, как показано ниже:

```
<div class="well">
<label>
    Search for:<input type="text" ng-model="criteria">
</label>
</div>
<table class="table table-bordered">
    <thead>
```

```
            <th>Name</th>
            <th>Description</th>
            ...
        </thead>
        <tbody>
            <tr ng-repeat="backlogItem in backlog | filter:criteria">
                <td>{{backlogItem.name}}</td>
                <td>{{backlogItem.desc}}</td>
                ...
            </tr>
        </tbody>
    </table>
```

Как видите, добавить фильтр на основе ввода пользователя проще простого; нужно лишь связать значение поля ввода с аргументом фильтра. Об остальном позаботятся механизмы автоматического связывания данных и обновления фреймворка AngularJS.

Условие критерия можно инвертировать, добавив префиксный оператор `!`.

В предыдущем примере выполняется поиск указанной подстроки по всем свойствам исходных объектов. Если необходимо более точно управлять списком свойств, в которых должен выполняться поиск, можно передать фильтру дополнительный аргумент с объектом. Этот объект будет действовать как «образец запроса». Следующий фрагмент ограничивается поиском подстроки только в свойстве `name` и включает в конечный список только элементы, свойство `done` которых содержит значение `false`:

```
ng-repeat="item in backlog | filter:{name: criteria, done: false}"
```

В этом фрагменте объект в аргументе включает все свойства, значения которых должны соответствовать требованиям. Можно сказать, что условия, выраженные с помощью отдельных свойств, объединяются с использованием логического оператора «И» (AND).

Кроме всего прочего в AngularJS существует имя, соответствующее любым именам свойств: `$`. Используя этот шаблонный символ в качестве имени свойства, мы можем реализовать объединение с помощью логических операторов «И» (AND) и «ИЛИ» (OR). Допустим, что требуется найти объекты, имеющие совпадение с введенной подстрокой хотя бы в одном из их свойств, но включать в список только элементы, свойство `done` которых имеет значение `false`. Для этого выражение фильтрации можно переписать, как показано ниже:

```
ng-repeat="item in backlog | filter:{$: criteria, done: false}"
```

Может так получиться, что комбинация критериев будет настолько сложной, что ее нельзя будет выразить с использованием синтаксиса объектов. В таких случаях фильтру можно передать функцию (функцию-предикат). Такая функция будет вызываться для каждого отдельного элемента в исходной коллекции. Результирующий массив будет содержать только элементы, для которых функция-предикат вернет `true`. Представьте нужно вывести только задания, которые уже выполнены и потребовали более 20 единиц усилий. Функцию фильтрации для этого примера не только легко написать:

```
$scope.doneAndBigEffort = function (backlogItem) {
    return backlogItem.done && backlogItem.estimation > 20;
};
```

но и использовать:

```
ng-repeat="item in backlog | filter:doneAndBigEffort"
```

### Подсчет элементов после фильтрации

Иногда бывает желательно показать количество пунктов в коллекции. Обычно для этого можно использовать выражение `{{myArray.length}}`. Однако проблема усложняется, когда возникает необходимость вывести количество элементов в отфильтрованном массиве. В простейшем случае можно было бы продублировать фильтр, определив его в директиве повторения и в выражении вычисления количества элементов. Если взять за основу наш последний пример:

```
<tr ng-repeat="item in backlog | filter:{$: criteria, done: false}">
```

подсчитать количество элементов можно было бы так:

```
Total: {{(backlog | filter:{$: criteria, done: false}).length}}
```

Очевидно, что такой прием имеет несколько недостатков. При его использовании придется не только повторить программный код, но и применить один и тот же фильтр несколько раз в двух разных местах, что не самым лучшим образом скажется на производительности.

Чтобы исправить ситуацию, можно создать промежуточную переменную (`filteredBacklog`) для хранения отфильтрованного массива:

```
ng-repeat=
  "item in filteredBacklog = (backlog | filter:{$: criteria,
done: false})"
```

и затем просто отобразить на экране размер массива в этой переменной:

```
Total: {{filteredBacklog.length}}
```

Такой подход к подсчету количества элементов в отфильтрованном массиве позволяет определить логику фильтрации в одном месте.

Другое возможное решение состоит в том, чтобы переместить логику фильтрации в контроллер и сохранять в объекте контекста уже отфильтрованные данные. Этот способ обладает еще одним серьезным преимуществом: перенос логики фильтрации в контроллер упрощает ее тестирование. Чтобы воспользоваться этим решением, необходимо знать, как обращаться к фильтрам из программного кода на JavaScript, о чем мы и поговорим далее в этой главе.

## Сортировка с помощью фильтра orderBy

Очень часто пользователям предоставляется возможность сортировки табличных данных. Обычно пользователю достаточно щелкнуть на заголовке той или иной колонки, чтобы выбрать данное поле для сортировки, и щелкнуть еще раз, чтобы отсортировать данные по этому полю в обратном порядке. В этом разделе мы реализуем такой способ сортировки с применением средств AngularJS.

Основным нашим инструментом решения этой задачи будет фильтр `orderBy`. По завершении наш пример с таблицей заданий будет дополнен пиктограммами управления сортировкой, как показано на рис. 4.4.

| Name ▲ | Description | Priority | Estimation | Done? |
|---|---|---|---|---|
| Incorporate reviewers remarks | Go over and reviewers remarks | 6 | 3 | false |
| Prepare outline | Prepare book outline with estimated page count | 2 | 2 | true |
| Prepare samples | Think of code samples | 3 | 5 | true |
| Review draft | Re-read and review the 1st draft | 4 | 5 | false |
| Total: 4 | | | | |

**Рис. 4.4.** Список с поддержкой сортировки

Порядок использования фильтра `orderBy` прост и понятен, поэтому мы можем сразу же перейти к знакомству с кодом примера, не тратя много времени на знакомство с теорией. Сначала мы добавим поддержку сортировки, а затем – индикаторы направления сортировки. Ниже приводится фрагмент разметки, имеющий отношение к сортировке:

```
<thead>
    <th ng-click="sort('name')">Name</th>
    <th ng-click="sort('desc')">Description</th>
    ...
</thead>
<tbody>
    <tr ng-repeat="item in filteredBacklog = (backlog |
        filter:criteria | orderBy:sortField:reverse)">
        <td>{{item.name}}</td>
        <td>{{item.desc}}</td>
        ...
    </tr>
</tbody>
```

Фактическая сортировка выполняется фильтром `orderBy`, которому в нашем примере передается два аргумента:

- `sortField`: имя свойства, по которому выполняется сортировка;
- `reverse` (направление сортировки): этот аргумент определяет направление сортировки – по возрастанию или по убыванию.

Для обработки события щелчка мышью на заголовке колонки таблицы вызывается функция `sort`. Она производит выбор поля для сортировки и определяет направление сортировки. Ниже приводится фрагмент контроллера, имеющего отношение к сортировке:

```
$scope.sortField = undefined;
$scope.reverse = false;

$scope.sort = function (fieldName) {
    if ($scope.sortField === fieldName) {
        $scope.reverse = !$scope.reverse;
    } else {
        $scope.sortField = fieldName;
        $scope.reverse = false;
    }
};
```

Наш пример реализации сортировки основан на предыдущем примере фильтрации, поэтому сейчас список заданий поддерживает и фильтрацию, и сортировку. Оба фильтра AngularJS удивительно легко поддаются объединению.

Фильтр `orderBy` был преднамеренно помещен после фильтра `filter`, выполняющего фильтрацию. Причина тому – производительность: сортировка является более дорогостоящей операцией, чем фильтрация, поэтому алгоритм упорядочения лучше применять как можно к меньшему множеству данных.

Теперь, когда с сортировкой покончено, нам осталось добавить пиктограммы, указывающие на поле, по которому отсортирован список, и направление сортировки. Здесь нам снова пригодится директива `ng-class`. Ниже следует пример реализации поддержки индикатора для колонки «name»:

```
<th ng-click="sort('name')">Name
<i ng-class="{'icon-chevron-up' : isSortUp('name'),
              'icon-chevrondown': isSortDown('name')}"></i>
</th>
```

Функции `isSortUp` и `isSortDown` очень просты в реализации:

```
$scope.isSortUp = function (fieldName) {
    return $scope.sortField === fieldName && !$scope.reverse;
};

$scope.isSortDown = function (fieldName) {
    return $scope.sortField === fieldName && $scope.reverse;
};
```

Конечно, существует множество способов отображения индикаторов сортировки, но данный отличается тем, что старается отделить классы CSS от кода на JavaScript. Благодаря этому представление легко можно изменить простым изменением шаблона.

## Создание собственных фильтров – реализация постраничного вывода

К настоящему моменту мы реализовали вывод списка заданий в виде таблицы с поддержкой фильтрации и сортировки. Постраничный вывод – еще один распространенный прием организации пользовательского интерфейса, часто используемый при отображении больших наборов данных.

Фреймворк AngularJS не имеет встроенных фильтров, которые помогли бы нам точно выделить подмножество элементов массива, указав начальный и конечный индексы. Поэтому для поддержки постраничного вывода нам потребуется реализовать собственный фильтр, и это хорошая возможность познакомиться с особенностями создания собственных фильтров.

Чтобы получить представление об интерфейсе нового фильтра (назовем его `pagination`) давайте сначала напишем заготовку разметки:

```
<tr ng-repeat="item in filteredBacklog = (backlog |
    pagination:pageNo:pageSize">
```

```
    <td>{{item.name}}</td>
    ...
</tr>
```

Новый фильтр `pagination` должен иметь два параметра: номер (индекс) отображаемой страницы и ее размер (количество элементов на странице).

Далее следует самая первая, упрощенная реализация фильтра (обработка ошибок была преднамеренно опущена, чтобы вы могли сосредоточить все свое внимание на особенностях создания фильтров):

```
angular.module('arrayFilters', [])

    .filter('pagination', function(){

        return function(inputArray, selectedPage, pageSize) {
            var start = selectedPage*pageSize;
            return inputArray.slice(start, start + pageSize);
        };
    });
```

Фильтр, как и любой другой провайдер, должен быть зарегистрирован в экземпляре модуля. Для этого нужно вызвать метод `filter` и передать ему имя фильтра и фабричную функцию, которая будет вызываться для создания новых экземпляров фильтра. Зарегистрированная функция должна возвращать фактическую функцию фильтра.

Первый аргумент функции фильтра `pagination` представляет исходную коллекцию для фильтрации, а последующие – параметры, управляющие фильтрацией.

Фильтры очень легко поддаются тестированию; они обрабатывают входные данные и не должны иметь побочных эффектов. Ниже приводится пример теста для нашего фильтра `pagination`:

```
describe('pagination filter', function () {

    var paginationFilter;
    beforeEach(module('arrayFilters'));
    beforeEach(inject(function (_paginationFilter_) {
        paginationFilter = _paginationFilter_;
    }));

    it('should return a slice of the input array', function () {

        var input = [1, 2, 3, 4, 5, 6];
        expect(paginationFilter(input, 0, 2)).toEqual([1, 2]);
        expect(paginationFilter(input, 2, 2)).toEqual([5, 6]);
```

```
    });

    it('should return empty array for out-of bounds', function () {

        var input = [1, 2];
        expect(paginationFilter(input, 2, 2)).toEqual([]);
    });
});
```

Тестирование фильтра выполняется так же просто, как тестирование обычной функции. Структура только что представленного теста должна быть понятна вам, так как в нем не использовались никакие новые конструкции. Единственное, что требует пояснений, это способ доступа к экземплярам фильтра и кода на JavaScript.

## Доступ к фильтрам из кода на JavaScript

Фильтры обычно вызываются из разметки (в выражениях, с помощью символа вертикальной черты), однако с тем же успехом доступ к экземплярам фильтров можно получить и в программном коде на JavaScript (в контроллерах, службах, других фильтрах и так далее). Благодаря этому имеется возможность комбинировать существующие фильтры при реализации новой функциональности.

**Система внедрения зависимостей** AngularJS может внедрять фильтры в любые объекты. Выразить зависимость от фильтра можно любым из двух способов:

- с помощью службы `$filter`;
- снабдив имя фильтра окончанием `Filter`.

Служба `$filter` – это функция поиска, позволяющая получить экземпляр фильтра по его имени. Чтобы увидеть, как она действует, напишем простой фильтр, действующий подобно фильтру `limitTo` и способный усекать длинные строки. Кроме того, наш собственный фильтр будет добавлять `"..."` в конец усеченной строки. Соответствующая реализация приводится ниже:

```
angular.module('trimFilter', [])
    .filter('trim', function($filter){

        var limitToFilter = $filter('limitTo');

        return function(input, limit) {
            if (input.length > limit) {
                return limitToFilter(input, limit-3) + '...';
            }
            return input;
```

```
        };
    });
```

Вызов `$filter('limitTo')` позволяет нам получить экземпляр фильтра по его имени.

Предыдущий пример прекрасно справляется со своей задачей, однако существует альтернативный способ, который часто не только работает быстрее но и лучше воспринимается при чтении кода:

```
.filter('trim', function(limitToFilter){

    return function(input, limit) {
        if (input.length > limit) {
            return limitToFilter(input, limit-3) + '...';
        }
        return input;
    };
});
```

Во втором примере оказалось достаточно объявить зависимость в виде `[filter_name]Filter`, где `[filter_name]` – имя фильтра, который требуется получить.

Организация доступа к фильтрам с применением службы `$filter` имеет не самый удачный синтаксис, именно поэтому мы предпочитаем использовать форму с использованием окончания `Filter`. Единственный случай, где может потребоваться использовать службу `$filter`, – когда необходимо получить несколько фильтров в одной функции или извлечь экземпляр фильтра по имени, хранящемуся в переменной, например, `$filter(filterName)`.

## Правила использования фильтров

Фильтры незаменимы, когда используются в шаблонах для форматирования и преобразования данных, так как имеют краткий и выразительный синтаксис. Но фильтры – это специализированный инструмент и как любой другой инструмент могут плохо подходить для выполнения другой работы. В этом разделе описываются ситуации, когда следует избегать применения фильтров и использовать другие, более удачные решения.

### Фильтры и операции с деревом DOM

Иногда возникает соблазн вернуть из фильтра разметку HTML. В действительности в AngularJS имеется один такой фильтр: `linky` (находящийся в отдельном модуле `ngSanitize`).

Однако с практической точки зрения такие фильтры, возвращающие разметку HTML, являются далеко не лучшим решением. Главная проблема в том, что для отображения результатов работы таких фильтров необходимо использовать одну из директив связывания, описанных выше: `ngBindUnsafeHtml` или `ngBindHtml`. Это не только усложняет синтаксис (в сравнении с простым использованием выражений вида `{{expression}}`), но и делает веб-страницу уязвимой для атак, основанных на внедрении HTML.

Чтобы увидеть некоторые проблемы, сопровождающие фильтры, которые возвращают разметку HTML, рассмотрим реализацию простого фильтра `highlight`:

```
angular.module('highlight', [])

    .filter('highlight', function(){
        return function(input, search) {
            if (search) {
                return input.replace(new RegExp(search, 'gi'),
                        '<strong>$&</strong>');
            } else {
                return input;
            }
        };
    });
```

В этом примере сразу бросается в глаза жестко «зашитая» разметка HTML. Как результат, у нас не получится использовать такой фильтр в директиве интерполяции и придется писать такой шаблон:

```
<input ng-model="search">
<span ng-bind-html="phrase | highlight:search"></span>
```

Кроме того, разметка HTML, возвращаемая фильтром, не может содержать директивы AngularJS, так как они не будут интерпретироваться фреймворком.

Ту же самую задачу с большим успехом можно решить с помощью собственной директивы, не рискуя безопасностью веб-приложения. Подробнее о директивах рассказывается в главах 9 и 10.

### Дорогостоящие преобразования данных в фильтрах

Фильтры, когда используются в шаблонах, становятся неотделимой частью выражений AngularJS и так же часто вызываются. Фактически такие фильтры вызываются множество раз в каждом цикле отображения. В этом легко можно убедиться на практике, создав журналирующую обертку вокруг фильтра:

```
angular.module('filtersPerf', [])
    .filter('logUppercase', function(uppercaseFilter){
        return function(input) {
            console.log('Calling uppercase on: '+input);
            return uppercaseFilter(input);
        };
    });
```

Задействовав этот новый фильтр в следующей разметке:

```
<input ng-model="name"> {{name | logUppercase}}
```

можно обнаружить, что инструкция `log` выполняется как минимум один (чаще два) раз в ответ на каждое нажатие клавиши! Одного этого эксперимента достаточно, чтобы убедиться, насколько часто вызываются фильтры, и понять, почему они должны действовать максимально быстро.

Не удивляйтесь, увидев, что фильтр многократно вызывается в одной строке; это действует механизм проверки изменений в AngularJS. Стремитесь писать свои фильтры так, чтобы они работали по-настоящему молниеносно.

### Нестабильные фильтры

Так как фильтры вызываются множество раз, вполне оправданно ожидать, что при вызове с теми же исходными данными фильтр будет возвращать те же результаты. Такие функции называют стабильными в отношении их параметров.

Ситуация легко может выйти из-под контроля, если фильтр не обладает таким качеством. Чтобы увидеть разрушительное действие нестабильных фильтров, напишем фильтр, выбирающий случайный элемент из входного массива:

```
angular.module('filtersStability', [])

    .filter('random', function () {

        return function (inputArray) {
            var idx = Math.floor(Math.random() * inputArray.length);
            return inputArray[idx];
        };
    })
```

Если предположить, что в объекте контекста имеется переменная `items`, хранящая массив с различными элементами, тогда фильтр

random можно было бы использовать в шаблонах, как показано ниже:

```
{{items | random}}
```

В процессе выполнения этот шаблон будет выводить случайные значения, поэтому может показаться, что фильтр действует правильно. Но стоит открыть консоль браузера, и вы сразу убедитесь в обратном:

```
Uncaught Error: 10 $digest() iterations reached. Aborting!
```

Это сообщение говорит о том, что при каждой попытке вычислить выражение, оно возвращало разные результаты. AngularJS постоянно следит за изменениями в моделях и повторно вычисляет выражения, надеясь, что результаты стабилизируются. Когда количество таких попыток достигает десяти, цикл обновления прерывается, последний результат выводится на экран, а в консоль – сообщение об ошибке. В главе 11 вы найдете более подробное обсуждение этой темы, а также описание особенностей работы внутренних механизмов AngularJS, что поможет вам понять причины, породившие эту ошибку.

В подобных ситуациях случайное значение лучше вычислять в контроллере, перед отображением шаблона:

```
.controller('RandomCtrl', function ($scope) {

    $scope.items = new Array(1000);
    for (var i=0; i<$scope.items.length; i++) {
        $scope.items[i] = i;
    }

    $scope.randomValue = Math.floor(Math.random() * $scope.items.
length);
});
```

Здесь случайное значение вычисляется перед обработкой шаблона, благодаря чему мы можем безопасно использовать выражение {{randomValue}} для вывода подготовленного значения.

Если ваша функция способна возвращать разные результаты для одних и тех же исходных данных, она является не самым лучшим кандидатом на звание фильтра. Используйте такую функцию в контроллере и передавайте фреймворку AngularJS уже готовое для отображения значение.

# В заключение

В этой главе мы увидели множество приемов отображения данных, содержащихся в модели.

В начале мы познакомились с соглашениями по именованию и затем перешли к обзору встроенных директив AngularJS. Особое внимание мы уделили директиве `ng-repeat`, как одной из самых мощных и наиболее часто используемых.

Как мы уже знаем, фреймворк AngularJS опирается на использование декларативных шаблонов на основе DOM, тем не менее, в некоторых пограничных ситуациях этот подход имеет определенные ограничения. Поэтому очень важно уметь распознавать такие ситуации и быть готовыми к необходимости изменять разметку, когда это потребуется.

Фильтры обеспечивают поддержку очень удобного синтаксиса форматирования и преобразования данных перед выводом в пользовательском интерфейсе. Мы видели, что AngularJS уже имеет несколько полезных фильтров, но когда возникают специфические потребности, мы легко можем создавать собственные фильтры. Не следует злоупотреблять функциями фильтрации и всегда желательно рассматривать варианты альтернативных решений.

Директивы, фильтры и шаблоны, рассматривавшиеся в этой главе, главной своей целью имели отображение данных. Но прежде чем отображать данные, необходимо иметь возможность вводить их с помощью различных элементов ввода. В следующей главе мы поближе познакомимся с особенностями использования элементов форм в AngularJS и проблемами ввода данных.

# ГЛАВА 5. Создание улучшенных форм

Фреймворк AngularJS основан на использовании стандартных форм HTML и элементов ввода. Это означает, что вы можете продолжать создавать свои формы, используя привычные и понятные элементы HTML, а также знакомые инструменты создания разметки HTML.

К настоящему моменту в нашем примере приложения SCRUM уже создано несколько форм с элементами ввода, связанными с данными в моделях и кнопками, позволяющих сохранять и удалять данные. Фреймворк AngularJS включает все необходимое для поддержки связей элемент-модель и событие-обработчик.

В этой главе мы подробно рассмотрим, как работают формы в AngularJS, а затем добавим в наши прикладные формы поддержку проверки ввода и возможность динамического взаимодействия с пользователем.

Эта глава охватывает следующие темы:

- связывание данных в моделях и директивы ввода;
- проверка введенных данных в формах;
- вложенные и повторяющиеся формы;
- отправка форм на сервер;
- сброс форм в исходное состояние.

## Сравнение традиционных форм с формами AngularJS

Прежде чем приступать к улучшению форм в нашем приложении, необходимо познакомиться с особенностями работы форм AngularJS. В этом разделе мы познакомимся с различиями между стандартными HTML-элементами ввода и директивами ввода в AngularJS. Пос-

мотрим, как AngularJS изменяет и расширяет поведение HTML-элементов ввода, и как управляет обновлением информацией в моделях данных.

В стандартной форме HTML значением элемента ввода является значение, которое будет отправлено на сервер в составе формы, как показано на рис. 5.1.

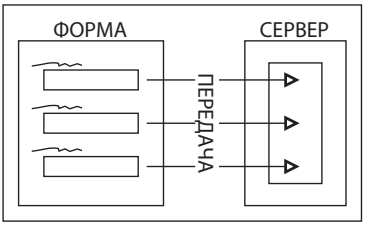


**Рис. 5.1.** Передача значений на сервер в составе формы

Проблема заключается в том, что мы вынуждены работать со значениями в элементах ввода, так как они отображаются на экране. Это часто не совпадает с нашими желаниями. Например, поле ввода даты может позволять пользователю вводить строку в некотором предопределенном формате, например, «12 марта 2013». Но в программном коде было бы желательно работать с этим значением в виде объекта `Date`. Однако постоянное перекодирование из одного формата в другой слишком утомительное занятие и чревато ошибками.

Фреймворк AngularJS отделяет модель от представления. Вы позволяете директивам ввода заботиться об отображении значений, а AngularJS позаботится об обновлении модели в случае изменения значений. Это дает возможность работать с моделями, например, в контроллерах, не беспокоясь о том, как данные отображаются или вводятся.

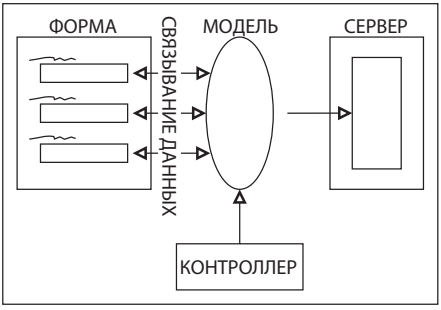


**Рис. 5.2.** AngularJS отделяет модель от представления

Для достижения такого разделения AngularJS расширяет формы HTML директивами `form` и `input`, директивами проверки ввода и контроллерами. Эти директивы и контроллеры переопределяют поведение по умолчанию форм HTML, но с точки зрения случайного наблюдателя формы в AngularJS выглядят очень похожими на стандартные формы HTML.

Прежде всего познакомимся с директивой `ngModel`, позволяющей определять связи между элементами ввода и моделью.

## Введение в директиву ngModel

Мы уже видели, как AngularJS создает связи между полями объекта контекста и элементами HTML. Настройку связей можно выполнить с помощью фигурных скобок, `{{}}`, или директив, таких как `ngBind`. Но такие связи являются однонаправленными. Для связывания значения с элементом ввода используется другая директива – `ngModel`:

```
<div>Hello <span ng-bind="name"/></div>
<div>Hello <input ng-model="name"/></div>
```

Попробуйте ее в действии по адресу: http://bit.ly/Zm55zM.

В первом элементе `div` производится связывание свойства `scope.name` текущего объекта контекста с текстом в элементе `span`. Это – однонаправленная связь: если изменится значение свойства `scope.name`, изменится и текст в элементе `span`; но если изменится текст в элементе `span`, значение свойства `scope.name` останется прежним.

Во втором элементе `div` производится связывание свойства `scope.name` текущего объекта контекста со значением элемента ввода. В данном случае образуется настоящая двунаправленная связь – как только изменится значение в элементе ввода, немедленно изменится значение свойства `scope.name`. Это изменение, в свою очередь, будет автоматически подхвачено связью между `scope.name` и элементом `span`.

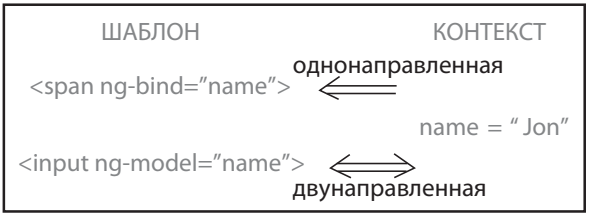


**Рис. 5.3.** Одно- и двунаправленные связи между моделью и элементами HTML

Почему была создана отдельная директива для определения связей с элементами ввода? Директива `ngBind` связывает (однонаправленным образом) значение своего выражения с текстом, содержащимся в элементе разметки. Директива `ngModel`, напротив, образует двунаправленную связь, чтобы изменения в элементе ввода отражались на содержимом модели.

Кроме того, AngularJS поддерживает директивы преобразования и проверки значений в директиве `ngModel` в моменты синхронизации содержимого между моделью и директивой ввода. Мы рассмотрим работу этого механизма в разделе с описанием контроллера `ngModelController`.

# Создание формы с информацией о пользователе

В этом разделе описывается простая форма для ввода информации о пользователе для нашего примера приложения SCRUM. На протяжении этой главы мы последовательно будем наращивать функциональность этой формы, чтобы продемонстрировать возможности форм в AngularJS. Ниже приводится базовая реализация действующей формы:

```
<h1>User Info</h1>
<label>E-mail</label>
<input type="email" ng-model="user.email">

<label>Last name</label>
<input type="text" ng-model="user.lastName">

<label>First name</label>
<input type="text" ng-model="user.firstName">

<label>Website</label>
<input type="url" ng-model="user.website">

<label>Description</label>
<textarea ng-model="user.description"></textarea>

<label>Password</label>
<input type="password" ng-model="user.password">

<label>Password (repeat)</label>
<input type="password" ng-model="repeatPassword">

<label>Roles</label>
```

```
<label class="checkbox">
    <input type="checkbox" ng-model="user.admin"> Is Administrator
</label>

<pre ng-bind="user | json"></pre>
```

Опробовать ее можно на странице http://bit.ly/10ZomqS.

На первый взгляд форма выглядит как простой список стандартных HTML-элементов ввода, но в действительности все они являются директивами ввода фреймворка AngularJS. Каждая директива ввода `input` дополняется директивой `ngModel`, связывающей текущий контекст со значением элемента `input`. В данном примере каждый элемент `input` связан с полем объекта `user`, который сам является частью текущего контекста. Значение поля модели можно было бы вывести в консоль, как показано ниже:

```
$log($scope.user.firstName);
```

Обратите внимание на отсутствие элемента `form`, а также атрибутов `name` и `id` в элементах ввода. Для простой формы без проверки введенной информации этого вполне достаточно. AngularJS гарантирует синхронизацию значений в элементах ввода со значениями в модели. Благодаря этому мы свободно можем манипулировать моделью `user` в контроллере, не беспокоясь о том, как эти изменения будут отображаться в представлении.

Мы также связали элемент `pre` с JSON-представлением модели `user`. Благодаря этому мы сможем видеть, как AngularJS выполняет синхронизацию модели.

# Директивы ввода

В этом разделе мы познакомимся с директивами ввода, поддерживаемыми фреймворком AngularJS по умолчанию. Использование директив ввода весьма естественно для тех, кто занимается созданием форм HTML, потому что они основаны на HTML.

В своих формах вы можете использовать все стандартные типы элементов ввода HTML. Директивы ввода работают совместно с директивой `ngModel`, обеспечивающей дополнительные функциональные возможности, такие как проверка ввода или связывание с моделью. Директива `input` в AngularJS проверяет значение атрибута `type`, чтобы определить, какую функциональность следует добавить в элемент ввода.

## Добавление проверки обязательного наличия значения

Все основные директивы ввода поддерживают атрибут `required` (или `ngRequired`). Добавляя этот атрибут в элемент ввода, вы сообщаете фреймворку AngularJS, что значения `null`, `undefined` или `""` (пустая строка) в `ngModel` являются недопустимым. Подробнее об этом рассказывается в разделе «Проверка полей», ниже.

## Текстовые элементы ввода

Простейшая директива ввода, `type="text"` или `textarea`, позволяет ввести любую строку. При изменении текста в элементе ввода, немедленно обновляется значение модели.

Другие директивы ввода текстовой информации (адресов электронной почты, URL или чисел) действуют аналогично, за исключением того, что им позволено обновлять модель, только если введенная строка соответствует заданному регулярному выражению. Если вводится адрес электронной почты, соответствующее поле в модели будет оставаться пустым, пока в элемент ввода не будет введена строка с допустимым адресом электронной почты. Это означает, что в модель никогда не попадет недопустимый адрес. В этом состоит одно из преимуществ отделения моделей от представлений.

В дополнение к этим проверкам все директивы ввода текста позволяют указать минимальную и максимальную длину текста, а также произвольное регулярное выражение. Для этой цели используются директивы `ngMinLength`, `ngMaxLength` и `ngPattern`:

```
<input type="password" ng-model="user.password"
       ng-minlength="3" ng-maxlength="10"
       ng-pattern="/^.*(?=.*\d)(?=.*[a-zA-Z]).*$/">
```

Опробовать этот пример можно по адресу: http://bit.ly/153L87Q.

Здесь поле `user.password` модели должно содержать от 3 до 10 символов включительно и соответствовать регулярному выражению, требующему наличие в строке хотя бы одной буквы и цифры.

Обратите внимание, что встроенные средства проверки не препятствуют пользователю вводить недопустимые строки. Директива ввода просто очищает поле модели, если строка оказывается недопустимой.

## Кнопки-флажки

Кнопка-флажок (checkbox) просто показывает состояние ввода логического значения. В нашей форме эта директива ввода присваивает `true` или `false` полю модели, определяемому директивой `ngModel`. Вы можете убедиться в этом, поэкспериментировав с полем «Is Administrator» в нашей форме.

```
<input type="checkbox" ng-model="user.admin">
```

Поле `user.admin` будет установлено в значение `true`, если флажок отмечен, и в значение `false` в противном случае. И наоборот, флажок будет отображаться как отмеченный, если присвоить полю `user.admin` значение `true`.

Существует также возможность определять строковые значения, соответствующие значениям `true` и `false`. Например, мы могли бы записывать в поле `role` строки `"admin"` и `"basic"`, как показано ниже:

```
<input type="checkbox" ng-model="user.role"
       ng-true-value="admin" ng-false-value="basic">
```

Опробовать этот пример можно по адресу: http://bit.ly/Yidt37.

В данном случае поле `user.role` будет содержать либо строку `"admin"`, либо строку `"basic"`, в зависимости от состояния флажка.

## Радиокнопки

Радиокнопки (radio buttons) действуют как группа взаимозависимых вариантов выбора. Фреймворк AngularJS значительно упрощает объединение радиокнопок в группу: достаточно связать все радиокнопки в группе с одним и тем же полем модели. После этого можно использовать стандартный HTML-атрибут `value`, чтобы определить, какое значение сохранить в модели при выборе той или иной радиокнопки:

```
<label>
    <input type="radio" ng-model="user.sex" value="male">
    Male
</label>
<label>
    <input type="radio" ng-model="user.sex" value="female">
    Female
</label>
```

Опробовать этот пример можно по адресу: http://bit.ly/14hYNsN.

## Элементы выбора из списка

Директива ввода `select` дает возможность создать визуальный элемент раскрывающегося списка (drop-down list), из которого пользователь сможет выбрать один или несколько пунктов. В AngularJS можно определять пункты таких списков не только статически, но и динамически, в виде массива в объекте контекста.

### Простые строковые пункты

Если имеется статический список пунктов для выбора, можно просто перечислить их в виде элементов `option` внутри элемента `select`:

```
<select ng-model="sex">
    <option value="m" ng-selected="sex=='m'">Male</option>
    <option value="f" ng-selected="sex=='f'">Female</option>
</select>
```

Имейте в виду, что значением атрибута `value` может быть только строка, поэтому связываемое с ним значение в модели может быть только строкой.

Если понадобится связать элемент с полем модели, не являющимся строкой, следует использовать прием динамического создания списка из имеющихся данных, а затем использовать директиву `ngOptions`, как показано ниже.

### Динамическое определение пунктов с помощью директивы ngOptions

Фреймворк AngularJS поддерживает для директивы `select` расширенный синтаксис динамического определения сложных списков. Если потребуется связать со значением директивы `select` объект, а не простую строку, используйте атрибут `ngOptions`. Он принимает **выражение-генератор**, возвращающее пункты для отображения. Это выражение имеет вид, как показано на рис. 5.4.

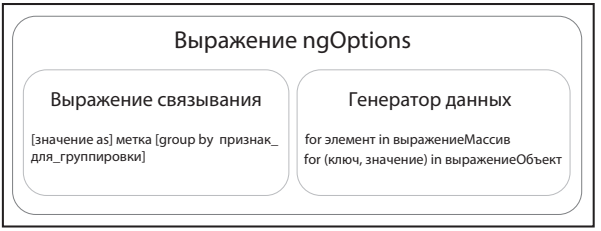


**Рис. 5.4.** Выражение-генератор `ngOptions`

Генератор данных описывает источник информации для отображаемых пунктов. Исходными данными могут служить элементы массива или свойства объекта. Для каждого значения, возвращаемого генератором, будет создан пункт в раскрывающемся списке.

Выражение связывания описывает, что должно извлекаться из каждого элемента исходных данных и как извлеченная информация должна быть связана с пунктом в элементе `select`.

### Типичные примеры использования ngOptions

Прежде чем перейти к изучению особенностей выражений-генераторов, рассмотрим несколько типичных примеров.

### Использование массивов в качестве источника данных

Выбор объекта `user` из списка, содержащего значения полей `user.email`:

```
ng-options="user.email for user in users"
```

Выбор объекта `user` из списка с вычисляемыми пунктами (функция должна быть определена в текущем контексте):

```
ng-options="getFullName(user) for user in users"
```

Выбор адреса электронной почты вместо всего объекта `user` из списка, содержащего полные имена пользователей:

```
ng-options="user.email as getFullName(user) for user in users
```

Выбор объекта `user` из списка, в котором имена пользователей разбиты на две группы по половому признаку:

```
ng-options="getFullName(user) group by user.sex for user in users"
```

Опробовать этот пример можно по адресу: http://bit.ly/1157jqa.

### Использование объектов в качестве источника данных

Допустим, что имеется два объекта, определяющие отношения между названиями стран и их кодами:

```
$scope.countriesByCode = {
    'AF' : 'AFGHANISTAN',
    'AX' : 'ELAND ISLANDS',
    ...
};

$scope.countriesByName = {
```

```
    'AFGHANISTAN' : 'AF',
    'ELAND ISLANDS' : 'AX',
    ...
};
```

Чтобы выбрать код страны из списка с названиями, упорядоченными по коду:

```
ng-options="code as name for (code, name) in countriesByCode"
```

Чтобы выбрать код страны из списка с названиями, упорядоченными по алфавиту:

```
ng-options="code as name for (name, code) in countriesByName"
```

Опробовать этот пример можно по адресу: http://bit.ly/153LKdE.

Теперь, после знакомства с некоторыми примерами, можно перейти к изучению полной спецификации этих выражений.

### Генераторы данных

Если источником данных является массив, тогда `выражениеМассив` (рис. 5.4) должно возвращать массив. Директива выполнит обход всех элементов массива и в каждой итерации присвоит текущий элемент массива переменной `элемент`.

Пункты в раскрывающемся списке будут отображаться в том же порядке, в каком следуют соответствующие им элементы массива.

Если источником данных является объект, тогда `выражениеОбъект` (рис. 5.4) должно возвращать объект. Директива выполнит обход всех свойств объекта и в каждой итерации присвоит значение текущего свойства переменной `значение`, а имя свойства – переменной `ключ`.

Пункты в раскрывающемся списке будут упорядочены по именам свойств в алфавитном порядке.

### Выражение связывания

Выражение связывания определяет, как получить отображаемую метку и значение для каждого пункта списка и как группировать их. Это выражение может использовать все, что доступно выражениям AngularJS, включая фильтры. В общем случае выражение связывания имеет следующий синтаксис:

```
значение as метка group by признак_для_группировки
```

Если выражение `значение` не указано, тогда в качестве значения для записи в модель будет использоваться сам элемент данных. Если указано выражение группировки, оно должно возвращать имя группы для данного пункта списка.

### Пустые пункты в директиве select

Что произойдет, если фактическое значение в модели, связанной с директивой `select`, не соответствует ни одному из значений в списке? В этом случае директива отобразит пустой пункт в начале списка.

Пустой пункт будет выбираться всякий раз, когда значение в модели не соответствует ни одному из пунктов в списке. Если пользователь вручную выберет пустой пункт, в модель будет записано значение `null`.

Имеется возможность явно определить пустой пункт, добавив соответствующий элемент `option` с пустой строкой в качестве значения:

```
<select ng-model="..." ng-options="...">
    <option value="">-- No Selection --</option>
</select>
```

Опробовать этот пример можно по адресу: http://bit.ly/ZeNpZX.

Здесь определяется пустой пункт с отображаемой меткой «-- No Selection --».

Когда пустой пункт определяется явно, он всегда будет отображаться в списке и может быть выбран пользователем.

Если не определить собственный пустой пункт в объявлении директивы `select`, он будет генерироваться автоматически.

Если пустой пункт генерируется директивой, он будет виден, только когда модель будет хранить значение, не соответствующее ни одному пункту в списке. Поэтому пользователь не сможет вручную выбрать его и установить значение `select` в `null` или `undefined`.

Пустой пункт можно скрыть, определив его вручную и настроив его стиль, как `display:none`.

```
<option style="display:none" value=""></option>
```

Опробовать этот пример можно по адресу: http://bit.ly/ZeNpZX.

В данном случае директива будет использовать пустой пункт, но браузер не будет отображать его. Если теперь значение в модели не будет совпадать ни с одним пунктом, в браузере отобразится пустое поле списка, но в самом списке пустой пункт будет отсутствовать.

### Директива select и эквивалентность объектов

Директива `select` сопоставляет значение в модели со значениями пунктов списка, используя оператор эквивалентности объектов (`===`). То есть, если значениями пунктов списка являются объекты, а не простые значения (такие как числа или строки) в модели необходимо сохранять ссылку на фактическое значение в пункте меню. В противном случае директива `select` будет считать объекты разными.

Мы можем определить в контроллере доступные для выбора пункты списка в виде массива объектов:

```
app.controller('MainCtrl', function($scope) {
    $scope.sourceList = [
        {'id': '10005', 'name': "Anne"},
        {'id': '10006', 'name': "Brian"},
        {'id': '10007', 'name': "Charlie"}
    ];
    $scope.selectedItemExact = $scope.sourceList[0];
    $scope.selectedItemSimilar = {'id': '10005', 'name': "Anne"};
});
```

Здесь `selectedItemExact` фактически хранит ссылку на первый элемент в массиве `sourceList`, а `selectedItemSimilar` – совершенно посторонний объект, даже при том, что он содержит аналогичную информацию:

```
<select
    ng-model="selectedItemExact"
    ng-options=" item.name for item in sourceList">
</select>
<select
    ng-model="selectedItemSimilar"
    ng-options="item.name for item in sourceList">
</select>
```

Опробовать этот пример можно по адресу: http://bit.ly/Zrachk.

Здесь с помощью директивы `select` создаются два раскрывающихся списка. Изначально тот, что связан с `selectedItemSimilar`, не будет отображать выбранное значение. Поэтому вы всегда должны связывать выбранное значение с элементом массива `ng-options`. Для этого, например, можно выполнить поиск в массиве, чтобы найти соответствующий элемент.

### Выбор сразу нескольких пунктов

Если потребуется организовать выбор сразу нескольких пунктов списка, можно просто добавить в директиву `select` атрибут `multiple`. Директива `ngModel` свяжет ее с массивом, содержащим ссылки на значения всех выбранных пунктов.

В AngularJS имеется директива `ngMultiple`, которая принимает выражение, позволяющее решить – когда предоставлять возможность множественного выбора. В настоящее время директива `select` не реагирует на изменение значения `ngMultiple`, поэтому директива `ngMultiple` имеет весьма ограниченную область применения.

## *Использование скрытых полей ввода*

В AngularJS все данные модели хранятся в объекте контекста, поэтому скрытые поля ввода редко бывают необходимы. По этой же причине в AngularJS отсутствует директива ввода `hidden`. Скрытые поля ввода могут потребоваться всего в двух случаях: для внедрения значений, получаемых от сервера, и для поддержки традиционного механизма отправки форм HTML.

### Внедрение значений, получаемых от сервера

Если на стороне сервера вы пользуетесь некоторым механизмом шаблонов для создания разметки HTML, и вам требуется передать данные на сторону клиента через шаблон, просто добавьте директиву `ng-init` в разметку HTML, генерируемую на сервере, которая добавит желаемое значение в контекст:

```
<form ng-init="user.hash='13513516'">
```

Здесь разметка HTML, отправляемая сервером, содержит элемент `form` с директивой `ng-init`, инициализирующей хеш-код `user.hash` в контексте формы.

### Отправка традиционной формы HTML

Иногда бывает желательно отправить на сервер некоторые значения, отсутствующие в представлении, то есть, не видимые пользователю. Желаемого можно достигнуть, добавив скрытые поля в форму. Однако в AngularJS мы работаем с моделью, отделенной от формы, поэтому нам не нужны скрытые поля. Можно просто добавить необходимые значения в контекст и затем имитировать отправку формы

с помощью службы `$http`. Подробнее о том, как это делается, рассказывается в главе 3.

# Устройство механизма связывания данных в ngModel

До настоящего момента мы не раз видели, что `ngModel` создает связь между моделью и значением в поле ввода. В этом разделе мы посмотрим, что еще может дать нам эта директива, и как она работает.

## *ngModelController*

Каждая директива `ngModel` создает экземпляр `ngModelController`. Этот контролер доступен всем директивам в элементе `input`.

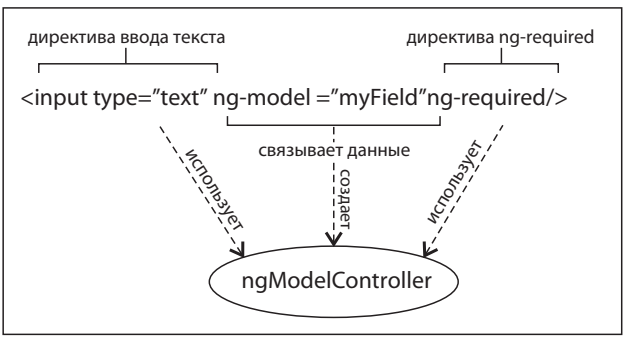


**Рис. 5.5.** Контроллер `ngModelController` доступен всем директивам в элементе ввода

Контроллер `ngModelController` реализует управление связью между значением, хранящимся в модели (определяется директивой `ngModel`), и значением, отображаемым в элементе ввода.

Кроме того, `ngModelController` определяет допустимость видимого значения и определяет факт изменения содержимого элемента ввода.

### Передача значения между моделью и представлением

Контроллер `ngModelController` имеет конвейер, который включается всякий раз, когда происходит обновление связанных данных. Он состоит из двух массивов: `$formatters`, используемого для

преобразования данных при передаче из модели в представление, и `$parsers`, используемого для преобразования данных при передаче из представления в модель. Каждая директива в элементе `input` может добавлять собственные форматеры и парсеры, чтобы участвовать в работе конвейера, как показано на рис. 5.6:

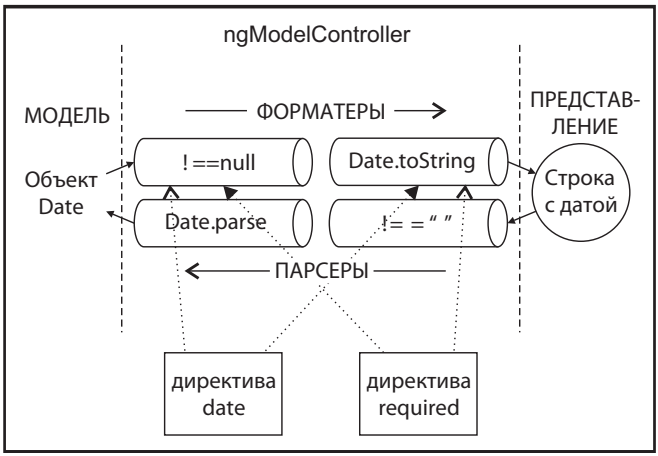


**Рис. 5.6.** Конвейер передачи данных в `ngModelController`

Здесь в конвейер передачи данных добавлены две директивы. Директива `date` выполняет преобразование дат, а директива контролирует, обязательное наличие значения.

### Слежение за изменением значения

Наряду с преобразованием данных при передаче между моделью и представлением, `ngModelController` определяет моменты изменения значения и его допустимость.

В момент инициализации контроллер `ngModelController` помечает значение как «нетронутое», то есть, неизмененное. В элементах ввода оно отмечается CSS-классом `ng-pristine`. Когда представление изменяется, например, в результате ввода символов в поле, значение помечается как «грязное» (измененное), и CSS-класс `ng-pristine` замещается классом `ng-dirty`.

Определив собственные стили CSS для этих классов, можно изменять внешний вид элементов ввода, в зависимости от наличия изменений в данных:

```
.ng-pristine { border: solid black 1px; }
.ng-dirty { border: solid black 3px; }
```

В данном случае при изменении данных в элементе ввода рамка вокруг него станет толще.

### Проверка допустимости значения в поле ввода

Директивы в элементе ввода могут также сообщать контроллеру `ngModelController` о допустимости или недопустимости значения. Обычно это реализуется путем включения инструмента проверки значения в конвейер. Контроллер `ngModelController` проверяет допустимость и применяет CSS-класс `ng-valid` или `ng-invalid`, в соответствии с результатом проверки. Мы можем определить собственные стили отображения элементов, опираясь на эти классы:

```
.ng-valid.ng-dirty { border: solid green 3px; }
.ng-invalid.ng-dirty { border: solid red 3px; }
```

Здесь используется комбинация признаков наличия изменений и допустимости, чтобы стили применялись только к полям, измененным пользователем: если введено недопустимое значение, элемент ввода окружается толстой красной рамкой, а если введено допустимое значение – толстой зеленой рамкой.

В следующем разделе, «Проверка форм», мы будет показано, как работать с понятиями «неизмененный», «грязный», «допустимый» и «недопустимый» программно.

# Проверка форм в AngularJS

В этом разделе рассказывается, как использовать директивы проверки и как они работают с контроллером `ngFormController` поддержки полноценного фреймворка проверки данных.

## *ngFormController*

Каждая директива `form` (или `ngForm`) создает экземпляр контроллера `ngFormController`. Объект `ngFormController` управляет признаками допустимости и наличия изменений в форме. Важно отметить, что он использует контроллер `ngModelController` для проверки каждого поля с директивой `ngModel` в форме.

В момент создания, контроллер `ngModelController` регистрирует себя в первом контроллере `ngFormController`, с которым сталкивается в списке своих родительских элементов. Благодаря этому `ngFormController` знает все свои директивы ввода, за которыми нужно следить. Он может проверить допустимость этих полей или

признак наличия изменений в них и установить соответствующие признаки для всей формы.

### Использование атрибута name для включения форм в контекст

Есть возможность сделать контроллер `ngFormController` доступным в локальном контексте, указав имя формы. Любые элементы ввода внутри формы, также имеющие имена, получат доступ к своим объектам `ngModelController` через свойство в своем объекте `ngFormController`.

В табл. 5.1 показано, какие контроллеры, связанные со всеми элементами формы, доступны в контексте:

**Таблица 5.1.** Контроллеры, связанные с элементами формы и доступные в контексте

| HTML | Контекст | Контроллер |
|---|---|---|
| <form name="form1"> | model1, model2, ...<br>form1 : {<br>$valid, $invalid,<br>$pristine, $dirty, ... | ngFormController |
| <input<br>name="field1"<br>ng-model="model1"<br>/> | field1: {<br>$valid, $invalid,<br>$pristine, $dirty, ...<br>}, | ngModelController |
| <input<br>name="field2"<br>ng-model="model2"<br>/> | field2: {<br>$valid, $invalid,<br>$pristine, $dirty, ...<br>}<br>}, | ngModelController |
| </form> | | |

# Добавление динамического поведения в форму с информацией о пользователе

Наша форма позволяет вводить значения в поля и мы можем изменять внешний вид элементов ввода, исходя из введенных значений. Но чтобы обеспечить у пользователя более благоприятные впечатления, можно было бы дополнительно показывать сообщения об ошиб-

ках и изменять состояние кнопок в форме, в зависимости от состояний полей.

Наличие объектов `ngFormController` и `ngModelController` в объекте контекста позволяет работать с информацией о состоянии формы программно. Мы можем использовать такие значения, как `$invalid` и `$dirty`, чтобы решать, что должно быть доступно или видимо пользователю.

## Вывод сообщений об ошибках

Если в каком-то поле ввода обнаружится недопустимое значение, мы можем вывести сообщение об ошибке как для всей формы целиком, так и для конкретного поля ввода. Эту возможность реализует следующий шаблон:

```
<form name="userInfoForm">
  <div class="control-group"
       ng-class="getCssClasses(userInfoForm.email)">

    <label>E-mail</label>
    <input type="email" ng-model="user.email"
           name="email" required>

    <span ng-show="showError(userInfoForm.email, 'email')" ...>
      You must enter a valid email
    </span>

    <span ng-show="showError(userInfoForm.email, 'required')" ...>
      This field is required
    </span>
  </div>
  ...
</form>
```

и контроллер:

```
app.controller('MainCtrl', function($scope) {
    $scope.getCssClasses = function(ngModelContoller) {
        return {
            error: ngModelContoller.$invalid &&
ngModelContoller.$dirty,
            success: ngModelContoller.$valid &&
ngModelContoller.$dirty
        };
    };
    $scope.showError = function(ngModelController, error) {
```

```
            return ngModelController.$error[error];
        };
    });
```

Опробовать этот пример можно по адресу: http://bit.ly/XwLUFZ.

Он демонстрирует обработку поля ввода адреса электронной почты в форме с информацией о пользователе. Для оформления формы мы использовали стили Twitter Bootstrap CSS, где определены классы CSS `control-group` и `inline-help`. Мы также создали в контроллере две вспомогательные функции.

Директива `ng-class` изменяет классы CSS в элементе `div`, содержащем текстовую метку, поле ввода и текст справки. Она вызывает метод `getCssClasses()`, передавая ему объект и имя ошибки.

В качестве аргумента-объекта методу фактически передается `ngModelController`, доступный через контроллер `ngFormController`, который в свою очередь доступен через значение `scope.userInfoForm.email` контекста.

Метод `getCssClasses()` возвращает объект, определяющий классы CSS для добавления. Именами свойств объекта служат имена классов CSS, а значениями – `true`, если класс с этим именем должен быть добавлен. В данном случае `getCssClasses()` вернет `true` в свойстве `error`, если модель изменилась и содержит недопустимое значение, или в свойстве `success`, если модель изменилась и содержит допустимое значение.

### Выключение кнопки сохранения

Мы можем сделать неактивной кнопку **Save** (Сохранить), когда форма находится в состоянии, непригодном для сохранения.

```
<form name="userInfoForm">
    ...
    <button ng-disabled="!canSave()">Save</button>
</form>
```

Здесь в представление добавляется кнопка **Save** (Сохранить) с директивой `ngDisabled`. Данная директива делает кнопку неактивной, когда выражение возвращает `true`. В этом примере используется инвертированное значение, возвращаемое методом `canSave()`. Метод `canSave()` должен быть доступен в текущем контексте, для чего мы определим его в главном контроллере:

```
app.controller('MainCtrl', function($scope) {
    $scope.canSave = function() {
```

```
        return $scope.userInfoForm.$dirty &&
            $scope.userInfoForm.$valid;
        };
    });
```

Опробовать этот пример можно по адресу: http://bit.ly/123zIhw.

Метод `canSave()` проверяет – установлены ли флаги `$dirty` и `$valid` в `userInfoForm`. Если они установлены, следовательно форма готова к сохранению.

## Отключение процедуры проверки, встроенной в браузер

Современные браузеры также пытаются проверять значения полей ввода. Обычно такая проверка выполняется в момент отправки формы. Например, если в поле ввода указан атрибут `required`, браузер немедленно сообщит об ошибке, как только вы попытаетесь отправить форму с незаполненными обязательными полями.

Так как все необходимые проверки мы реализуем посредством директив и контроллеров AngularJS, для нас обычно нежелательно, чтобы браузер выполнял свои проверки. Мы можем отключить их, применив атрибут `novalidate` в элементе `form`, появившийся в HTML5:

```
<form name="novalidateForm" novalidate>
```

Опробовать этот пример можно по адресу: http://bit.ly/1110hS4.

Данная форма, с именем `novalidateForm`, получила атрибут `novalidate`, сообщающий браузеру, что он не должен пытаться проверять поля ввода, имеющиеся в этой форме.

# Вложенные формы

В отличие от стандартных форм HTML, формы AngularJS могут вкладываться друг в друга. Но, так как теги `form` внутри других тегов `form` считаются недопустимыми с точки зрения языка разметки HTML, поддержка вложенных форм в AngularJS была реализована в виде директивы `ngForm`.

Каждая форма с именем будет добавлена в родительскую форму или непосредственно в контекст, если родительская форма отсутствует.

### Вложенные формы как компоненты многократного пользования

Вложенные формы действуют подобно составным полям со своим собственным механизмом проверки допустимости значений. Такие формы могут играть роль своеобразных компонентов других форм. В примере ниже мы сгруппировали два текстовых поля ввода для создания виджета, позволяющего вводить пароль и его подтверждение:

```
<script type="text/ng-template" id="password-form">
  <ng-form name="passwordForm">
    <div ng-show="user.password != user.password2">
      Passwords do not match
    </div>
    <label>Password</label>
    <input ng-model="user.password" type="password" required>
    <label>Confirm Password</label>
    <input ng-model="user.password2" type="password" required>
  </ng-form>
</script>

<form name="form1" novalidate>
  <legend>User Form</legend>
  <label>Name</label>
  <input ng-model="user.name" required>
  <ng-include src="'password-form'"></ng-include>
</form>
```

Опробовать этот пример можно по адресу: http://bit.ly/10QWwyu.

Мы определили вложенную форму как фрагмент шаблона. В данном случае – это встраиваемый блок `script`, но этот фрагмент можно также сохранить в отдельном файле. Ниже определяется родительская форма `form1`, которая подключает вложенную форму с помощью директивы `ngInclude`.

Вложенная форма включает собственную информацию о допустимости ее содержимого и соответствующие классы CSS. Обратите также внимание, что вложенная форма имеет атрибут `name`, значение которого превращается в имя свойства вмещающей формы.

## Повторение вложенных форм

Иногда бывает нужно повторить некоторые поля в форме несколько раз, исходя из данных в модели. Такая потребность часто возникает, когда необходимо использовать одну и ту же форму для отображения отношений «один-ко-многим» в данных.

В нашем приложении SCRUM мы могли бы позволить пользователям указывать ноль или более адресов веб-сайтов в форме ввода информации о пользователе. Для этого можно воспользоваться директивой ngRepeat:

```
<form ng-controller="MainCtrl">
  <h1>User Info</h1>
  <label>Websites</label>
  <div ng-repeat="website in user.websites">
    <input type="url" ng-model="website.url">
    <button ng-click="remove($index)">X</button>
  </div>
  <button ng-click="add()">Add Website</button>
</form>
```

Ниже приводится контроллер, инициализирующий модель и реализующий вспомогательные функции remove() и add():

```
app.controller('MainCtrl', function($scope) {
    $scope.user = {
        websites: [
            {url: 'http://www.bloggs.com'},
            {url: 'http://www.jo-b.com'}
        ]
    };
    $scope.remove = function(index) {
        $scope.user.websites.splice(index, 1);
    };
    $scope.add = function() {
        $scope.user.websites.push({ url: ''});
    };
});
```

Опробовать этот пример можно по адресу: http://bit.ly/XHLEWQ.

В шаблоне присутствует директива ngRepeat, которая выполняет итерации по веб-сайтам, сохраненным в профиле пользователя. Каждая директива ввода в повторяющемся блоке связывается с соответствующим свойством website.url в модели user.websites. Вспомогательные функции реализуют добавление и удаление элементов массива, а все остальное делает механизм связывания данных в AngularJS.

Весьма заманчиво определить каждый элемент массива websites как простую строку с адресом URL веб-сайта. Но этот способ неработоспособен, так как строки в JavaScript передаются по значению, из-за чего при попытке изменить адрес в поле ввода связь между строкой в блоке ngRepeat и строкой в массиве будет потеряна.

## Проверка повторяющихся полей ввода

Проблемы с такой реализацией начинают возникать, когда требуется организовать проверку повторяющихся полей. Чтобы получить доступ к свойствам `$valid`, `$invalid`, `$pristine`, `$dirty` и другим, нужно снабдить каждое поле ввода именем, уникальным в пределах формы. К сожалению в AngularJS не поддерживается возможность динамической генерации значений для атрибута `name` в директивах `input`. Имя должно быть фиксированной строкой.

Решить эту проблему можно с помощью вложенных форм. Каждая форма включает себя в текущий контекст, поэтому, если поместить вложенную форму в блок с повторяемыми директивами ввода, мы сможем выполнить проверку полей, используя этот контекст. Например, взгляните на следующий шаблон:

```
<form novalidate ng-controller="MainCtrl" name="userForm">
  <label>Websites</label>
  <div ng-show="userForm.$invalid">The User Form is invalid.</div>
  <div ng-repeat="website in user.websites" ng-form="websiteForm">
    <input type="url" name="website"
           ng-model="website.url" required>
    <button ng-click="remove($index)">X</button>
    <span ng-show="showError(websiteForm.website, 'url')">
      Pleae must enter a valid url</span>
    <span ng-show="showError(websiteForm.website, 'required')">
      This field is required</span>
  </div>
  <button ng-click="addWebsite()">Add Website</button>
</form>
```

и контроллер

```
app.controller('MainCtrl', function($scope) {
    $scope.showError = function(ngModelController, error) {
        return ngModelController.$error[error];
    };
    $scope.user = {
        websites: [
            {url: 'http://www.bloggs.com'},
            {url: 'http://www.jo-b.com'}
        ]
    };
});
```

Опробовать этот пример можно по адресу: http://bit.ly/14i1sTp.

Здесь, чтобы создать в текущем контексте вложенную форму для каждого веб-сайта в массиве `websites`, к элементу `div` применяется директива `ngForm`. Каждая вложенная форма получает имя `website-`

Form, а поле ввода в каждой форме – имя website. Это означает, что мы можем выполнить проверку ngModel для каждого веб-сайта изнутри контекста ngRepeat.

Мы пользуемся этой возможностью для вывода сообщений об ошибках. Две директивы ng-show отображают свои сообщения, когда функция showError возвращает true. Функция showError проверяет переданный ей объект ngModelController на наличие соответствующего элемента в поле $error. Мы можем также передать websiteForm.website в эту функцию, потому что он ссылается на объект ngModelController, связанный с полем ввода адреса веб-сайта.

За пределами ngForm, нельзя обратиться к объекту websiteForm (ngFormController) или websiteForm.website (ngModelController), потому что они отсутствуют в этом контексте. Однако можно обратиться к объекту userForm (ngFormController). Допустимость информации в этой форме определяется, исходя из допустимости всех вложенных в нее полей ввода и форм. Если одна из форм websiteForm будет признана недопустимой, недопустимой окажется и вмещающая ее форма userForm. Элемент div в начале формы отображает сообщение об ошибке, только если свойство userForms.$invalid имеет значение true.

# Отправка традиционной формы HTML

В этом разделе рассказывается, как фреймворк AngularJS обрабатывает отправку форм. Одностраничные приложения AJAX, для создания которых фреймворк AngularJS подходит лучше всего, обычно не используют процедуру непосредственной отправки форм на сервер, как это делают традиционные веб-приложения. Но иногда поддержка такой возможности бывает желательной. Ниже будут показаны разные варианты отправки формы, которые могут пригодиться для реализации отправки данных формы на сервер.

## *Непосредственная отправка форм на сервер*

Если в приложении на основе AngularJS указать в форме атрибут action, она будет отправляться по указанному адресу как обычно:

```
<form method="get" action="http://www.google.com/search">
  <input name="q">
</form>
```

Опробовать этот пример можно по адресу: http://bit.ly/115cQgq.

Имейте в виду, что пример на сайте Plnkr блокирует перенаправление на сайт Google.

## Обработка события отправки формы

Если атрибут `action` не указан, AngularJS будет полагать, что приложение самостоятельно осуществляет отправку формы вызовом функции контекста. В этом случае AngularJS будет препятствовать попыткам непосредственной отправки формы на сервер.

Упомянутую клиентскую функцию можно вызвать с помощью директивы `ngClick` в элементе `button` или директивы `ngSubmit` в элементе `form`.

Не следует использовать обе директивы, `ngSubmit` и `ngClick`, в одной форме, потому что браузер вызовет их обе и в результате форма будет отправлена дважды.

### Отправка формы с помощью ngSubmit

Чтобы выполнить отправку формы с помощью директивы `ngSubmit`, нужно определить выражение, которое произведет отправку. Вычисление выражения производится в ответ на нажатие клавиши **Enter** в одном из полей ввода или щелчок на одной из кнопок:

```
<form ng-submit="showAlert(q)">
  <input ng-model="q">
</form>
```

Опробовать этот пример можно по адресу: http://bit.ly/ZQBLYj.

Здесь нажатие клавиши **Enter** в поле ввода приведет к вызову метода `showAlert`.

Директиву `ngSubmit` следует использовать только в формах с одним полем ввода и содержащих не более одной кнопки, как например форма поиска в примере выше.

### Отправка формы с помощью ngClick

Чтобы выполнить отправку формы с помощью директивы `ngClick` в элементе `button` или `input [type=submit]`, нужно определить выражение, которое произведет отправку в ответ на щелчок на кнопке:

```
<form>
  <input ng-model="q">
  <button ng-click="showAlert(q)">Search</button>
</form>
```

Опробовать этот пример можно по адресу: http://bit.ly/153OvLS.

Здесь щелчок на кнопке или нажатие клавиши **Enter** в поле ввода приведет к вызову метода `showAlert`.

# Сброс формы в исходное состояние

В форме ввода информации о пользователе было бы неплохо предусмотреть возможность отменить ввод и сбросить форму в исходное состояние. Это легко реализовать, создав копию исходной модели и использовав ее для замены всех изменений, произведенных пользователем.

Шаблон:

```
<form name="userInfoForm">
  ...
  <button ng-click="revert()"
          ng-disabled="!canRevert()">Revert Changes</button>
</form>
```

Контроллер:

```
app.controller('MainCtrl', function($scope) {
    ...
    $scope.user = {
        ...
    };
    $scope.passwordRepeat = $scope.user.password;

    var original = angular.copy($scope.user);

    $scope.revert = function() {
        $scope.user = angular.copy(original);
        $scope.passwordRepeat = $scope.user.password;
        $scope.userInfoForm.$setPristine();
```

```
        };

        $scope.canRevert = function() {
            return !angular.equals($scope.user, original);
        };

        $scope.canSave = function() {
            return $scope.userInfoForm.$valid &&
                !angular.equals($scope.user, original);
        };
    });
```

Опробовать этот пример можно по адресу: http://bit.ly/17vHLWX.

Здесь создается кнопка, щелчок на которой возвращает модель в исходное состояние. В результате щелчка на кнопке вызывается функция `revert()` в текущем контексте. Кнопка становится неактивной, когда `canRevert()` возвращает `false`.

Вы можете видеть в реализации контроллера вызов метода `angular.copy()` с целью создания копии модели в локальной переменной. Метод `revert()` копирует оригинал обратно в действующую модель `user` и устанавливает форму в исходное состояние, чтобы удалить классы CSS в `ng-dirty`.

# В заключение

В этой главе было показано, как AngularJS расширяет стандартные элементы форм HTML с целью предоставить более гибкую и мощную систему ввода. Она позволяет отделить модель от представления с помощью `ngModel` и поддерживает механизмы слежения за изменениями и проверки допустимости введенных значений с помощью специализированных директив и объекта `ngFormController`.

В следующей главе мы расскажем, как лучше управлять навигацией по приложению. Мы покажем, насколько обширны возможности AngularJS в смысле отображения адресов URL непосредственно в различные аспекты приложения, и как с помощью `ngView` организовать автоматическое отображение той или иной информации, исходя из текущего адреса URL.

# ГЛАВА 6.
# Организация навигации

В предыдущих главах рассказывалось, как организовать получение данных со стороны сервера, их редактирование с использованием форм AngularJS и отображение на отдельных страницах с применением различных директив. В этой главе мы покажем как организовать отдельные экраны в полнофункциональное приложение с удобной системой навигации.

Хорошо продуманные и простые для запоминания **адреса URL (Uniform Resource Locators – унифицированные указатели ресурсов)** играют важную роль в структурировании приложений. Они позволяют быстро перемещаться между экранами приложения, используя широко известные возможности браузеров. Фреймворк AngularJS включает различные службы и директивы, несущие поддержку адресов URL для Web 1.0 в одностраничные веб-приложения, наиболее заметные черты которой перечислены ниже.

- Внешние URL, ссылающиеся на определенные функции внутри одностраничного веб-приложения. Такие адреса можно сохранять в закладках или обмениваться ими (например, по электронной почте или посредство систем обмена мгновенными сообщениями).
- Кнопки **Back** (Назад) и **Forward** (Вперед) браузера действуют в соответствии с ожиданиями, давая пользователям возможность перемещаться между разными экранами одностраничного веб-приложения.
- Адреса URL могут иметь хорошо читаемый, простой для запоминания формат в браузерах, поддерживающих программный интерфейс HTML5 для доступа к истории посещений.
- Поддержка адресов URL в AngularJS остается единообразной для всех версий браузеров – одно и то же приложение прекрасно будет работать и с браузерами, поддерживающими программный интерфейс HTML5 для доступа к истории посещений, и со старыми браузерами.

Фреймворк AngularJS имеет сложную систему обработки адресов URL. В этой главе вы познакомитесь со следующими темами:

- использование адресов URL в одностраничных веб-приложениях;
- подход к адресам URL, используемый в AngularJS и слой абстракции над адресами URL: службы `$location` и `$anchorScroll`;
- организация навигации в клиентских веб-приложениях с использованием службы `$route` (с ее провайдером – `$routeProvider`) и директивы `ngView`;
- Распространенные приемы использования, советы и рекомендации по использованию URL в одностраничных веб-приложениях на основе AngularJS.

# Адреса URL в одностраничных веб-приложениях

На раннем этапе развития Всемирной паутины навигация между страницами осуществлялась очень просто. Достаточно был ввести в адресной строке браузера определенный адрес, чтобы получить доступ к точно определенному ресурсу. В конце концов, адреса URL собственно и были предназначены, чтобы адресовать единственные, физические ресурсы (файлы) на серверах. После загрузки страницы можно мы могли перепрыгивать между ресурсами с помощью гиперссылок, а также пользоваться кнопками браузера **Back** (Назад) и **Forward** (Вперед) для перемещения между посещавшимися страницами.

Появление динамически отображаемых страниц сломало эту простую парадигму навигации. Ни с того, ни с сего, один и тот же адрес URL может привести браузер к разным страницам, в зависимости от внутреннего состояния приложения. Первыми жертвами повышения интерактивности Всемирной паутины стали кнопки браузеров **Back** (Назад) и **Forward** (Вперед). Их поведение стало просто непредсказуемым, и многие веб-сайты до сих пор препятствуют использованию этих двух кнопок для навигации (предлагая пользователям внутренние навигационные ссылки).

Одностраничные веб-приложения не особенно улучшают ситуацию! В современных приложениях, основанных на применении технологии AJAX, часто в адресной строке браузера сохраняется один

и тот же адрес URL (использовавшийся для начальной загрузки приложения). Все последующие взаимодействия с сервером обычно производятся с помощью объекта XHR и никак не отражаются в адресной строке браузера. При такой организации взаимодействий кнопки **Back** (Назад) и **Forward** (Вперед) оказываются полностью бесполезными, так как щелчок на них может привести к переходу на совершенно другой веб-сайт, а не в определенную точку внутри текущего веб-приложения. Создание закладок или копирование ссылки из адресной строки браузера также оказываются бессмысленными. Закладка всегда будет ссылаться на начальную страницу веб-приложения.

Однако кнопки **Back** (Назад) и **Forward** (Вперед), а также поддержка закладок в браузерах – очень удобный механизм. Пользователи не прочь использовать их и при работе с одностраничными веб-приложениями! К счастью AngularJS обладает полным комплектом инструментов, обеспечивающих ту же эффективность работы с адресами URL, какой обладала старая добрая Всемирная паутина со статическими ресурсами!

## Адреса URL с решеткой до появления HTML5

Как оказывается, есть один несложный прием, позволяющий вернуть привычную поддержку адресов URL в веб-приложения на основе технологии AJAX.

Этот прием опирается на тот факт, что изменение части адреса URL в адресной строке браузера, следующей за символом решетки (`#`), не вызывает перезагрузку отображаемой страницы. Эта часть адреса URL называется адресом фрагмента. Изменяя адрес фрагмента, можно добавлять новые элементы в стек истории посещений браузера (объект `window.history`). Этот стек используется кнопками **Back** (Назад) и **Forward** (Вперед). То есть, при правильной организации работы с историей посещений можно добиться корректной работы кнопок **Back** (Назад) и **Forward** (Вперед).

Рассмотрим множество типичных адресов URL, часто используемых CRUD-подобных приложениях. В идеале было бы желательно иметь адреса URL, ссылающиеся на список элементов, на форму редактирования единственного элемента, на форму создания нового элемента и так далее. Если взять в качестве примера функцию администрирования пользователей (из примера приложения SCRUM),

нам необходимы следующие адреса URL, отличающиеся лишь частично:

- `/admin/users/list` – выводит список существующих пользователей;
- `/admin/users/new` – выводит форму добавления нового пользователя;
- `/admin/users/[userId]` – выводит форму редактирования информации о существующем пользователе с идентификатором `[userId]`.

Эти адреса URL можно было бы преобразовать в адреса URL с фрагментами, применяя прием использования символа `#`, упомянутый выше:

- `http://myhost.com/#/admin/users/list`;
- `http://myhost.com/#/admin/users/new`;
- `http://myhost.com/#/admin/users/[userId]`.

Адреса URL, ссылающиеся на внутренние функции одностраничного веб-приложения подобным образом, часто называют «адресами URL с решеткой».

При наличии поддержки схемы адресов URL, описанной выше, появляется возможность изменить URL в адресной строке браузера и избежать при этом полной перезагрузки страницы. Браузер различает такие URL (с одинаковой основной частью адреса и с разными адресами фрагментов после символа `#`), сохраняет их в истории посещений и использует при обработке щелчков на кнопках **Back** (Назад) и **Forward** (Вперед). В то же время, если изменился лишь адрес фрагмента, браузер не выполняет никаких обращений к серверу.

Очевидно, что изменения одной только схемы адресов URL недостаточно. Необходимо написать некоторый код на JavaScript, который будет обслуживать изменение адреса фрагмента и соответственно модифицировать состояние приложения на стороне клиента.

## HTML5 и интерфейс истории посещений

Как рядовые пользователи, все мы любим простые, запоминающиеся и пригодные для сохранения в закладках адреса URL, но только что описанный трюк с символом решетки делает адреса URL чересчур длинными и, говоря начистоту, уродливыми. К счастью стандарт HTML5 предусматривает решение этой проблемы: прикладной интерфейс истории посещений (History API).

Прикладной интерфейс истории посещений (History API) поддерживается большинством современных браузеров. Однако в Internet Explorer он поддерживается, только начиная с версии 10. В более ранних версиях доступно лишь решение на основе адресов фрагментов.

Проще говоря, используя History API можно имитировать посещение внешних ресурсов без фактических обращений к серверу. Например, с помощью метода `history.pushState` можно добавлять в стек истории посещений полноценные адреса URL. Поддержка History API включает также встроенный механизм слежения за изменениями в стеке истории посещений. Мы можем написать собственный обработчик события `window.onpopstate` и изменять состояние приложения в ответ на это событие.

Используя History API, предусматриваемый стандартом HTML5, в одностраничных веб-приложениях можно реализовать ясную схему адресов URL (без символа решетки) и обеспечить благоприятные впечатления пользователей от возможности использовать механизм закладок и кнопок **Back** (Назад) и **Forward** (Вперед). Адреса URL из предыдущего примера можно привести к следующему виду:

- `http://myhost.com/admin/users/list;`
- `http://myhost.com/admin/users/new;`
- `http://myhost.com/admin/users/[userId].`

Эти адреса выглядят как самые обычные, «стандартные» адреса URL, ссылающиеся на действительные ресурсы. Это удобно, так как в результате мы получаем простые и легко запоминающиеся адреса URL. А если ввести один из этих адресов в адресную строку браузера, браузер не отличит его от любого другого адреса и отправит запрос серверу.

Чтобы режим, предусматриваемый стандартом HTML5, действовал корректно, сервер должен иметь соответствующие настройки и всегда возвращать начальную страницу приложения. Подробнее об этом рассказывается ниже в этой главе.

# Служба $location

Фреймворк AngularJS реализует слой абстракции над адресами URL в виде службы `$location`. Эта служба прячет различия между ад-

ресами с решеткой и адресами HTML5, позволяя разработчикам приложений взаимодействовать с непротиворечивым прикладным интерфейсом, независимым от типа браузера и поддерживаемого им режима работы с адресами URL, в том числе:

- предоставляет удобный и ясный прикладной интерфейс для доступа к различным элементам адресов (протокол, имя хоста, номер порта, путь, параметры запроса и так далее);
- позволят программно манипулировать различными элементами адресов и отражать изменения в адресной строке браузера;
- дает возможность следить за изменениями различных элементов адреса в адресной строке браузера и обрабатывать их;
- обрабатывать действия пользователя со ссылками на странице (например, щелчки на тегах `<a>`), отражать эти действия в истории посещений браузера.

## Знакомство с интерфейсом службы $location и адресами URL

Прежде чем перейти к практическим примерам использования службы `$location`, необходимо познакомиться с ее прикладным интерфейсом, а так как основной целью службы является работа с адресами URL, в первую очередь следует рассмотреть взаимосвязь между различными элементами URL и методами службы.

Возьмем в качестве примера адрес URL, ссылающийся на список пользователей. Чтобы усложнить пример, будем использовать URL, содержащий все возможные элементы: путь, строку запроса и адрес фрагмента. Допустим, что основная часть URL имеет следующий вид:

```
/admin/users/list?active=true#bottom
```

Этот URL можно расшифровать так: в административном разделе перечислить всех активных пользователей и перейти в конец списка. Здесь указана лишь часть URL, представляющая интерес с точки зрения приложения, в действительности же URL включает протокол, имя хоста и так далее. Этот адрес URL, в полной его форме, будет представлен по-разному, в зависимости от используемого режима (с решеткой или HTML5). В следующем URL можно заметить элементы, повторяющиеся в обоих режимах. В HTML5 он мог бы выглядеть так:

```
http://myhost.com/myapp/admin/users/list?active=true#bottom
```

В режиме с решеткой он мог бы быть более длинным:

```
http://myhost.com/myapp#/admin/users/list?active=true#bottom
```

Независимо от используемого режима, служба `$location` прячет различия между ними и предоставляет единообразный прикладной интерфейс. В табл. 6.1 перечислены некоторые из его методов:

**Таблица 6.1.** Методы службы $location

| Метод | Для примера выше вернет |
|---|---|
| `$location.url()` | `/admin/users/list?active=true#bottom` |
| `$location.path()` | `/admin/users/list` |
| `$location.search()` | `{active: true}` |
| `$location.hash()` | `bottom` |

Все эти методы являются методами чтения/записи в стиле jQuery. Иными словами, они могут использоваться и для чтения и для записи значения заданного элемента адреса URL. Например, прочитать адрес фрагмента URL можно вызовом `$location.hash()`, а изменить – вызовом того же метода с аргументом, например: `$location.hash('top')`.

Служба `$location` предлагает массу других методов (не перечисленных в табл. 6.1) для доступа к разным элементам URL: протоколу (`protocol()`), имени хоста (`host()`), номеру порту (`port()`) и абсолютному адресу URL (`absUrl()`). Эти методы действуют только как методы чтения – они не могут использоваться для изменения URL.

## Адреса фрагментов, навигация внутри страницы и $anchorScroll

Одна из особенностей адресов URL с символом решетки состоит в том, что адреса фрагментов после знака # обычно используются для навигации внутри загруженного документа, а в одностраничных приложениях рассматриваются как полноценная часть URL. Однако иногда все же бывает желательно иметь возможность прокручивать загруженный документ до указанной позиции. Проблема в том, что в режиме использования адресов URL с решеткой, они могут содержать два символа #, например:

```
http://myhost.com/myapp#/admin/users/list?active=true#bottom
```

Браузеры не в состоянии определить, что второй символ решетки (#bottom) должен использоваться для навигации внутри документа, и нам нужна небольшая помощь со стороны AngularJS. Эту помощь нам окажет служба $anchorScroll.

По умолчанию служба $anchorScroll просматривает адреса фрагментов в URL, и при обнаружении элемента, предназначенного для навигации внутри документа, прокрутит этот документ до указанной позиции. Данная служба корректно работает в обоих режимах использования адресов URL, HTML5 (когда в URL может присутствовать только один символ #) и с решеткой (URL может содержать два символа #). Проще говоря, служба $anchorScroll берет на себя работу, которую обычно выполняет браузер, но учитывает возможность работы в режиме адресов URL с решеткой.

Если необходим более полный контроль над функцией прокрутки документа в службе $anchorScroll, можно отключить автоматический просмотр адресов URL. Для этого нужно вызвать метод disableAutoScrolling() службы $anchorScrollProvider в блоке настройки модуля, как показано ниже:

```
angular.module('myModule', [])
    .config(function ($anchorScrollProvider) {
        $anchorScrollProvider.disableAutoScrolling();
    });
```

После этого приложение переходит полностью на ручное управление прокруткой документа. Выполнить прокрутку документа можно в любой момент, вызовом функции $anchorScroll().

## Настройка режима HTML5 интерпретации адресов URL

По умолчанию фреймворк AngularJS настроен на использование режима интерпретации адресов URL с решеткой. Чтобы включить поддержку более удобочитаемых адресов URL, поддерживаемых в режиме HTML5, следует изменить настройки фреймворка AngularJS по умолчанию и настроить сервер.

### На стороне клиента

Чтобы включить режим HTML5 интерпретации адресов URL в AngularJS, достаточно вызвать метод html5Mode() службы $locationProvider с соответствующим аргументом:

```
angular.module('location', [])
    .config(function ($locationProvider) {
```

```
        $locationProvider.html5Mode(true);
    })
```

### На стороне сервера

Чтобы обеспечить корректную интерпретацию адресов в режиме HTML5, необходима некоторая поддержка со стороны сервера. В двух словах: на веб-сервере требуется настроить перенаправление, чтобы в ответ на запросы с адресами URL любой длины, принадлежащими приложению, возвращалась начальная страница приложения (содержащая директиву `ng-app`).

Чтобы понять, зачем это необходимо, рассмотрим ситуацию, когда пользователь использует URL, сохраненный в закладках (в режиме HTML5), ссылающийся на список заданий некоторого конкретного проекта, например:

```
http://host.com/projects/50547faee4b023b611d2dbe9/productbacklog
```

Для браузера такой адрес URL выглядит как обычный адрес URL и, соответственно, браузер пошлет запрос серверу. Однако вполне очевидно, что данный URL имеет смысл только в контексте выполняющегося на стороне клиента одностраничного приложения – ресурс с адресом `/projects/50547faee4b023b611d2dbe9/productbacklog` физически не существует и не может быть сгенерирован сервером динамически. Единственное, что может сделать сервер с таким адресом – преобразовать его в адрес начальной страницы приложения. Это приведет к загрузке приложения в браузер. Когда приложение запустится, служба `$location` подхватит URL (все еще хранящийся в адресной строке браузера) и приложение получит возможность выполнить его интерпретацию.

Обсуждение дополнительных подробностей, касающихся настройки различных серверов, далеко выходит за рамки данной книги, однако существует ряд общих правил, применимых ко всем серверам. Во-первых, существует примерно три типа адресов URL, которые обычно приходится обрабатывать веб-серверу:

- ссылающиеся на статические ресурсы (изображения, файлы CSS, компоненты фреймворка AngularJS и так далее);
- предназначенные для извлечения или изменения данных, хранящихся на сервере (например, интерфейс RESTful);
- ссылающиеся на функции приложения в режиме HTML5, в ответ на которые (обычно хранящиеся в закладках или введенные вручную в адресной строке браузера) сервер должен возвращать начальную страницу приложения.

Из-за невозможности перечислить все адреса URL, которые могут быть посланы серверу в режиме HTML5 интерпретации URL, лучше, пожалуй, будет использовать отдельные префиксы для доступа к статическим ресурсам и выполнения операций с данными. Именно эта стратегия используется в приложении SCRUM, где доступ ко всем статическим ресурсам осуществляется по адресам URL с префиксом `/static`, а доступ к данным – по адресам URL с префиксом `/databases`. Для всех остальных адресов настраивается перенаправление на начальную страницу приложения SCRUM (`index.html`).

## Навигация вручную с помощью службы $location

Теперь, после знакомства с некоторыми методами службы `$location` и особенностями ее настройки, можно попробовать применить новые знания на практике и реализовать очень простую схему навигации с использованием директивы `ng-include` и службы `$location`.

Даже самая простая схема навигации в одностраничном веб-приложении способна предложить некоторые фундаментальные возможности, упрощающее. Как минимум мы должны иметь возможность:

- определять единообразные маршруты, простые в поддержке;
- обновлять состояние приложения в ответ на изменение URL;
- изменять URL в ответ на действия пользователя, связанные с навигацией в пределах приложения (щелчки на ссылках или на кнопках **Back** (Назад) и **Forward** (Вперед), и так далее).

В оставшейся части книги под термином «маршрутизация» будут подразумеваться средства, помогающие синхронизировать состояние приложения (представления и соответствующие модели) с изменениями адреса URL. Единственный маршрут – это коллекция метаданных, описывающих переход приложения в состояние, соответствующее заданному адресу URL.

### Структурирование страниц на основе маршрутов

Прежде чем погрузиться в примеры кода, следует отметить, что в типичном веб-приложении имеются «статические элементы» страниц (заголовок, подвал и так далее) и «динамические элементы», изменяющиеся в ответ на действия пользователя. Учитывая это, следует так организовать разметку и контроллеры, чтобы обеспечить четкое разделение статических и динамических элементов. Опираясь на приме-

ры адресов URL выше, мы можем структурировать разметку HTML приложения, как показано ниже:

```
<body ng-controller="NavigationCtrl">
<div class="navbar">
  <div class="navbar-inner">
    <ul class="nav">
      <li><a href="#/admin/users/list">List users</a></li>
      <li><a href="#/admin/users/new">New user</a></li>
    </ul>
  </div>
</div>
<div class="container-fluid" ng-include="selectedRoute.templateUrl">
  <!-- Здесь находится содержимое, зависящее от маршрута -->
</div>
</body>
```

Динамическая часть приложения в этом примере представлена тегом `</div>` с директивой `ng-include`, ссылающейся на динамический URL: `selectedRoute.templateUrl`. Но как это выражение обеспечивает изменение динамической части в ответ на изменение URL?

Рассмотрим код на JavaScript, точнее реализацию контроллера `NavigationCtrl`. Во-первых, переменную `routes` можно определить, как показано ниже:

```
.controller('NavigationCtrl', function ($scope, $location) {

    var routes = {
        '/admin/users/list': {templateUrl: 'tpls/users/list.html'},
        '/admin/users/new': {templateUrl: 'tpls/users/new.html'},
        '/admin/users/edit': {templateUrl: 'tpls/users/edit.html'}
    };
    var defaultRoute = routes['/admin/users/list'];
    ...
});
```

Объект `routes` определяет базовую структуру приложения. Он отображает все возможные адреса фрагментов в шаблоны разметки. Взглянув на эти определения, можно сразу заметить, какие экраны составляют приложение и какие шаблоны разметки используются в каждом маршруте.

### Отображение маршрутов в адреса URL

Наличия простой структуры маршрутизации недостаточно. Нам нужно обеспечить синхронизацию активного маршрута с текущим адресом URL. Для этого достаточно организовать слежение за компонентом `path()` текущего URL:

```
$scope.$watch(function () {
    return $location.path();
}, function (newPath) {
    $scope.selectedRoute = routes[newPath] || defaultRoute;
});
```

Здесь каждое изменение в компоненте `$location.path()` будет инициировать поиск среди предопределенных маршрутов. Если URL удастся распознать, будет выбран соответствующий ему маршрут. В противном случае произойдет возврат к маршруту по умолчанию.

### Определение контроллеров шаблонов

После обнаружения маршрута выполняется загрузка соответствующего шаблона разметки и включение его в страницу с помощью директивы `ng-include`. Как вы наверняка помните, директива `ng-include` создает новый контекст, который требуется наполнить данными, будь то список пользователей, информация о пользователе для редактирования и так далее.

Подготовка данных и реализация поведения в AngularJS возлагаются на контроллеры. Нам нужен способ, который позволил бы определить контроллер для каждого шаблона разметки. Очевидно, что самый простой путь – воспользоваться директивой `ng-controller` в корневом элементе каждого шаблона. Например, шаблон разметки, определяющий форму редактирования информации о пользователе, мог бы выглядеть примерно так:

```
<div ng-controller="EditUserCtrl">
  <h1>Edit user</h1>
    ...
</div>
```

Недостаток такого решения в том, что оно не позволяет повторно использовать тот же шаблон с разными контроллерами. Иногда бывает удобно повторно использовать одну ту же разметку HTML и менять только поведение и данные, стоящие позади нее. Одним из примеров таких ситуаций как раз и является форма редактирования, которую можно было бы использовать и для изменения информации о существующем пользователе, и для добавления нового пользователя. В данной ситуации разметка формы остается одной и той же, отличаются только данные и поведение.

### Недостающие мелочи в реализации навигации вручную

Самодельное решение, представленное выше, недостаточно надежно, и его не следует использовать в действующих приложениях. Однако, несмотря на свою неполноту, оно иллюстрирует некоторые интересные особенности службы `$location`.

Во-первых, служба `$location` реализует великолепную обертку вокруг низкоуровневого API, экспортируемого браузерами. Эта обертка не только обеспечивает единообразный стиль работы в разных версиях браузеров, но и с разными режимами интерпретации URL (с решеткой и с поддержкой истории посещений HTML5).

Во-вторых, этот пример позволяет по-настоящему оценить возможности, предлагаемые фреймворком AngularJS и его службами. Любой, кто хотя бы раз пытался реализовать подобную навигационную систему на чистом JavaScript, быстро заметит, как много возможностей предлагается фреймворком. Однако, этот пример далек от идеала и его можно улучшать и улучшать! Но мы не будем продолжать развивать дальше этот код, опирающийся на службу `$location`, а рассмотрим встроенное в AngularJS решение проблем с навигацией: службу `$route`.

# Служба $route

Фреймворк AngularJS имеет встроенную службу `$route`, которую можно настроить для обработки изменения маршрутов в одностраничных веб-приложениях. Она реализует все операции, которые мы пытались выполнять вручную, с применением службы `$location`, и дополнительно предлагает еще массу интересных возможностей. Далее мы приступим к постепенному знакомству с ней.

Начиная с версии 1.2, система маршрутизации AngularJS будет вынесена в отдельный файл (`angular-route.js`) и в собственный модуль (`ngRoute`). Если вы пользуетесь последней версией AngularJS, не забывайте подключать файл `angular-route.js` и объявлять зависимость от модуля `ngRoute`.

## *Определение основных маршрутов*

Прежде чем углубиться в изучение более сложных приемов использования, попробуем сначала реализовать элементарные маршруты, используя синтаксис службы `$route`.

Маршруты в AngularJS можно определять на этапе настройки приложения, используя для этого службу `$routeProvider`. Синтаксис службы `$routeProvider` напоминает синтаксис, использовавшийся выше, когда мы пытались реализовать маршрутизацию с применением службы `$location`:

```
angular.module('routing_basics', [])
    .config(function($routeProvider) {
        $routeProvider
            .when('/admin/users/list',
                  {templateUrl: 'tpls/users/list.html'})
            .when('/admin/users/new', {templateUrl: 'tpls/users/
new.html'})
            .when('admin/users/:id', {templateUrl: 'tpls/users/
edit.html'})

            .otherwise({redirectTo: '/admin/users/list'});
    })
```

Прикладной интерфейс службы `$routeProvider` позволяет определять новые маршруты, составляя цепочки из вызовов методов (`when`) и настраивать маршрут по умолчанию (`otherwise`).

После инициализации в приложение нельзя будет добавить новые маршруты (или удалить существующие). Это обусловлено тем, что провайдеры AngularJS могут внедряться и использоваться только в блоках настройки, выполняемых на этапе инициализации приложения.

В предыдущих примерах можно было заметить, что для настройки маршрутов использовалось единственное свойство `templateUrl`, но служба `$routeProvider` обладает намного более богатым синтаксисом.

Существует возможность с помощью свойства `template` определить содержимое маршрута непосредственно в его объявлении. Но такое решение не обладает достаточной гибкостью (и сложнее в сопровождении), поэтому оно редко используется на практике.

## Отображение содержимого маршрута

Обнаружив совпадение адреса URL с одним из маршрутов, его содержимое (определяется свойством `templateUrl` или `template`) можно отобразить с помощью директивы `ng-view`. В версии на основе службы `$location` разметка имела следующий вид:

```
<div class="container-fluid" ng-include="selectedRoute.templateUrl">
    <!-- Здесь находится содержимое, зависящее от маршрута -->
</div>
```

При использовании `ng-view` ее можно переписать иначе:

```
<div class="container-fluid" ng-view>
    <!-- Здесь находится содержимое, зависящее от маршрута -->
</div>
```

Как видите, мы просто заменили директиву `ng-include` директивой `ng-view`. На этот раз нам не потребовалось указывать значение атрибута, потому что директива `ng-view` «знает», что она должна отобразить содержимое текущего маршрута.

## Гибкое сопоставление маршрутов

В простейшей реализации мы полагались на очень простой алгоритм сопоставления с маршрутами, не поддерживающий переменную часть в адресах URL. Фактически это можно назвать алгоритмом только с большой натяжкой – мы просто пытались найти свойство в объекте, соответствующее пути в адресе URL! Из-за такой простоты алгоритма мы были вынуждены использовать параметр запроса в URL, чтобы передать идентификатор пользователя, как показано ниже:

```
/admin/users/edit?user={{user.id}}
```

Было бы намного удобнее пользоваться адресами URL, где идентификатор пользователя является составной частью URL, например:

```
/admin/users/edit/{{user.id}}
```

Механизм маршрутизации в AngularJS поддерживает такую возможность, позволяя определять переменную часть URL в виде произвольной строки, начинающейся с символа двоеточия (`:`). Например, определить схему маршрутизации, где идентификатор пользователя является составной частью URL, можно так:

```
.when('/admin/users/:userid', {templateUrl: 'tpls/users/edit.html'})
```

Такому шаблону адреса будет соответствовать любой адрес URL с произвольной строкой на месте `:userid`, например:

```
/users/edit/1234
/users/edit/dcc9ef31db5fc
```

Напротив, адреса с отсутствующей строкой на месте `:userid` или со строкой, содержащей символы слеша (`/`), не будут соответствовать этому шаблону адреса.

Обеспечить совпадение адресов, содержащих слеши в параметрах, возможно, но для этого придется использовать несколько иной синтаксис: `*id`. Символ звездочки можно также использовать, чтобы обеспечить совпадение путей, содержащих слеши: `/wiki/pages/*page`. В версии AngularJS 1.2 синтаксис сопоставления маршрутов будет расширен еще больше.

### Определение маршрутов по умолчанию

Маршрут по умолчанию можно настроить вызовом метода `otherwise`, передав ему определение маршрута, который должен использоваться для обслуживания любых адресов, не соответствующих другим маршрутам. Обратите внимание, что метод `otherwise` не требует передачи шаблона URL, так как маршрут по умолчанию может быть только один.

Обычно маршрут по умолчанию выполняет переадресацию на один из уже определенных маршрутов, используя свойство `redirectTo` объекта определяющего этот маршрут.

Маршрут по умолчанию будет использоваться в обоих случаях: и когда в URL отсутствует путь, и когда получен недопустимый URL (не совпадающий ни с одним маршрутом).

### Доступ к значениям параметров маршрутов

Выше было показано, что определение маршрута может содержать переменную часть, выступающую в роли параметра. Если URL соответствует некоторому маршруту с параметрами, вы легко сможете получить доступ к значениям этих параметров с помощью службы `$routeParams`. В действительности служба `$routeParams` – это простой объект JavaScript (хеш), ключами которого являются имена параметров, а значениями – строки, извлеченные из адреса URL.

Так как `$routeParams` представляет собой обычную службу, ее можно внедрять в любые объекты, управляемые системой внедрения зависимостей фреймворка AngularJS. Подобное внедрение можно увидеть на примере контроллера (`EditUserCtrl`), используемого для изменения информации о пользователе (`/admin/users/:userid`):

```
.controller('EditUserCtrl', function($scope, $routeParams, Users){
    $scope.user = Users.get({id: $routeParams.userid});
    ...
})
```

Служба `$routeParams` объединяет в один объект и значения параметров в пути URL, и значения параметров в строке запроса. Этот код с таким же успехом смог бы обслужить маршрут, объявленный как `/admin/users/edit`, и соответствующий ему адрес URL: `/admin/users/edit?userid=1234`.

## Повторное использование шаблонов разметки с разными контроллерами

До настоящего момента мы использовали подход, согласно которому внутри каждого шаблона разметки определяется контроллер, с помощью директивы `ng-controller`, ответственный за инициализацию его контекста. Но система маршрутизации в AngularJS дает возможность определять контроллеры на уровне маршрутов. Изъяв контроллеры из шаблонов, мы фактически устраняем тесную связь между шаблоном и контроллером, выполняющим инициализацию контекста этого шаблона.

Рассмотрим шаблон, реализующий форму для редактирования информации о пользователе:

```
<div ng-controller="EditUserCtrl">
  <h1>Edit user</h1>
  ...
</div>
```

Мы можем изменить его, убрав директиву `ng-controller`:

```
<div>
  <h1>Edit user</h1>
  ...
</div>
```

Теперь можно определить контроллер на уровне маршрута:

```
.when('/admin/users/:userid', {
    templateUrl: 'tpls/users/edit.html'
    controller: 'EditUserCtrl'})
```

Перемещение контроллера на уровень определения маршрута дает возможность многократно использовать один и тот же контроллер с разными шаблонами и, что более важно, многократно использовать один и тот же шаблон с разными контроллерами. Эта дополнительная гибкость может пригодиться во множестве разных ситуаций. Типичным примером может служить шаблон с формой редактирования элемента списка. Обычно для создания нового элемента и редактирования существующего можно с успехом использовать одну и ту же

разметку, но логика добавления нового элемента может несколько отличаться от логики редактирования существующего (например, использовать функцию создания нового вместо обновления существующего элемента).

## Предотвращение «мерцания» пользовательского интерфейса при изменении маршрута

Обычно переход приложения от одного экрана к другому сопряжен с получением и отображением новой разметки, а также с загрузкой соответствующих данных (модели). Однако, как оказывается, существует две немного отличающиеся стратегии, которые можно использовать для отображения нового экрана:

- отобразить разметку максимально быстро (даже если данные еще не готовы) и затем обновить пользовательский интерфейс по прибытии данных с сервера;
- выполнить все необходимые запросы к серверу, получить все необходимые данные и только потом отображать разметку для нового экрана.

По умолчанию используется первый подход. Для маршрута, в котором определены оба свойства, `templateUrl` и `controller`, AngularJS приступит к отображению разметки шаблона, даже если данные, запрошенные контроллером, еще не поступили со стороны сервера. Разумеется, AngularJS автоматически обновит изображение на экране, когда эти данные поступят (и будут связаны с контекстом), но при этом пользователи могут заметить неприятный эффект мерцания. Мерцание пользовательского интерфейса происходит из-за необходимости повторного отображения одного и того же шаблона через короткий промежуток времени, первый раз без данных, и второй – с данными.

Система маршрутизации в AngularJS обладает превосходной встроенной поддержкой второго подхода, когда фактическое изменение маршрута (и отображение нового пользовательского интерфейса) откладывается до готовности всех запрошенных данных. В свойстве `resolve` объекта, определяющего маршрут, можно перечислить все асинхронные зависимости контроллера этого маршрута, а AngularJS позаботится о разрешении всех этих зависимостей до фактического изменения маршрута (и инициализации контроллера).

Чтобы показать принцип использования свойства `resolve`, перепишем маршрут формы редактирования информации о пользователе, как показано ниже:

```
.when('/admin/users/:userid', {
    templateUrl: 'tpls/users/edit.html'
    controller: 'EditUserCtrl',
    resolve: {
        user: function($route, Users) {
            return Users.getById($route.current.params.userid);
        }
    }
})
```

Свойство `resolve` – это объект, ключи которого определяют переменные для внедрения в контроллер маршрута. Значения переменных определяют специализированные функции, которые также могут иметь зависимости, внедряемые системой AngularJS DI. В примере выше внедряются службы `$route` и `Users`, с помощью которых производится извлечение возвращаемых данных.

Такие функции могут возвращать простые значения, объекты или отложенные результаты. Если возвращается отложенный результат, AngularJS задержит изменение маршрута до момента завершения асинхронного задания. Аналогично, если определено несколько функций, возвращающих отложенные результаты, AngularJS дождется момента, когда завершится последнее асинхронное задание и только потом произведет изменение маршрута.

Функции в разделе `resolve` определения маршрута могут возвращать отложенные результаты. Фактическое изменение маршрута произойдет только после успешного завершения всех асинхронных заданий.

Как только все переменные маршрута (объявленные в разделе `resolve`) получат фактические значения, они будут внедрены в контроллер маршрута, как показано ниже:

```
.controller('EditUserCtrl', function($scope, user){
    $scope.user = user;
    ...
})
```

Это очень мощный прием, так как позволяет определять переменные, локальные для данного маршрута, и внедрять их в контролер маршрута. Существует достаточно много действующих приложений,

использующих этот прием. Он также применяется и в примере приложения SCRUM, где один и тот же контроллер используется дважды, с разными значениями переменной `user` (значение может определяться на месте или запрашиваться с сервера). Ниже приводится выдержка из реализации приложения SCRUM:

```
$routeProvider.when('/admin/users/new', {
    templateUrl:'admin/users/users-edit.tpl.html',
    controller:'UsersEditCtrl',
    resolve:{
        user: function (Users) {
            return new Users();
        }
    }
});

$routeProvider.when('/admin/users/:userId', {
    templateUrl:'admin/users/users-edit.tpl.html',
    controller:'UsersEditCtrl',
    resolve:{
        user: function ($route, Users) {
            return Users.getById($route.current.params.userId);
        }
    }
});
```

Определение локальных переменных на уровне маршрута (в разделе `resolve`) означает, что контроллеры, объявленные как часть маршрута, также могут внедряться вместе с этими локальными переменными. Это серьезно упрощает тестирование логики контроллеров.

## Предотвращение изменения маршрута

Иногда, в зависимости от сложившихся условий, бывает желательно исключить возможность изменения маршрута. Например, представьте, что доступ к форме редактирования информации о пользователе определяется следующим маршрутом:

```
/users/edit/:userid
```

Нам нужно решить, что делать, если указан идентификатор несуществующего пользователя. От приложения вполне естественно было бы ожидать, что оно не выполнит переход по маршруту, ссылающемуся на несуществующий элемент.

Как оказывается, свойство `resolve` определения маршрута имеет встроенную поддержку блокировки изменения маршрута. Если значение одного из ключей свойства `resolve` является отложенным

результатом, в процессе получения которого возникла ошибка, AngularJS отменит переход и не позволит сменить представление пользовательского интерфейса.

Если при вычислении какого-либо отложенного результата, возвращаемого в разделе `resolve` маршрута, возникнет ошибка, попытка изменить маршрут будет отменена и пользовательский интерфейс не изменится.

Важно также отметить, что в случае отмены попытки перехода, содержимое адресной строки браузера не будет восстановлено в прежнее состояние. Чтобы понять, что это означает на практике, предположим, что список пользователей отображается по адресу URL `/users/list`. Пункты списка могут быть оформлены в виде гиперссылок на соответствующие формы редактирования (`/users/edit/:userid`). Щелчок на любой из них изменит содержимое адресной строки браузера (то есть, в ней появится адрес, например такой как: `/users/edit/1234`). Но это не гарантирует возможность перехода к редактированию информации о выбранном пользователе (учетная запись пользователя могла быть удалена непосредственно перед этим; у пользователя, выполнившего щелчок, может быть недостаточно прав на выполнение операции; и так далее). Если попытка изменить маршрут будет отменена, адресная строка браузера не будет возвращена в предыдущее состояние и в ней по-прежнему будет красоваться адрес `/users/edit/1234`, даже при том, что пользовательский интерфейс отражает содержимое маршрута `/users/list`.

Адресная строка браузера и отображаемый пользовательский интерфейс могут рассинхронизироваться, если попытка смены маршрута будет отменена из-за ошибки получения отложенного результата.

# Ограничения службы $route

Встроенная служба `$route` реализована с большой тщательностью и с успехом может использоваться в подавляющем большинстве приложений, но некоторые ситуации все же оказываются ей не по зубам. В этом разделе будут перечислены такие ситуации, чтобы вы знали о них и были готовы изменить проект приложения или реализовать собственную службу маршрутизации!

Усилиями сообщества развивается более мощная система маршрутизации для приложений на основе AngularJS: `ui-router`. Ее цель – обеспечить поддержку вложенных маршрутов и маршрутов для фрагментов экрана. На момент написания этих строк работа еще не была завершена, но вы можете загрузить версию для разработчиков: https://github.com/angular-ui/ui-router.

## Один маршрут соответствует одной области на экране

Теперь вы знаете, что директива `ng-view` используется для обозначения элементов DOM, чье содержимое должно быть замещено содержимым маршрута (определяется свойством `template` или `templateUrl`). Это говорит о том, что один маршрут в службе `$route` может описать только одну область внутри пользовательском интерфейсе.

Однако на практике иногда бывает желательно заполнить несколько областей на экране в ответ на изменение маршрута. Типичным примером может служить меню, общее для нескольких маршрутов и изменяющееся при переходе из одной области приложении в другую, как показано на рис. 6.1.

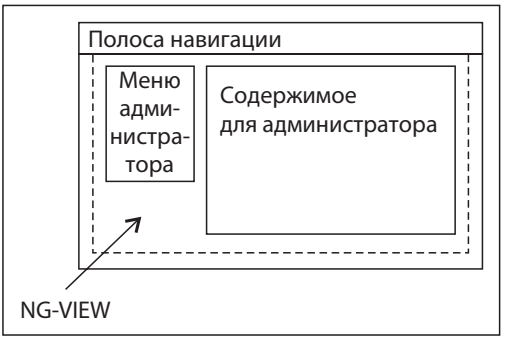


**Рис. 6.1.** Меню, общее для нескольких маршрутов

При такой организации приложения необходимо, чтобы меню администратора оставалось доступным, если при навигации пользователь продолжает оставаться в пределах административного раздела. Единственный способ реализовать такую схему навигации в текущей версии AngularJS – использовать комбинацию из директивы `ng-include` и приема, описанного в следующем разделе.

### Обработка нескольких областей на экране с помощью ng-include

Объекты с определениями маршрутов – это обычные объекты JavaScript, и они могут содержать любые свойства, в дополнение к тем, что поддерживаются фреймворком AngularJS. Определение собственных свойств в этом объекте никак не повлияет на поведение службы `$route`, но система маршрутизации в AngularJS сохранит эти свойства, благодаря чему мы сможем обратиться к ним позднее. Учитывая это, мы могли бы определить собственные свойства `menuUrl` и `contentUrl` на уровне маршрута:

```
$routeProvider.when('/admin/users/new', {
    templateUrl:'admin/admin.tpl.html',
    contentUrl:'admin/users/users-edit.tpl.html',
    menuUrl:'admin/menu.tpl.html',
    controller:'UsersEditCtrl',
    ...
});
```

и сохранить в свойстве `templateUrl` ссылку на новый шаблон, реализующий поддержку меню и вложенных подшаблонов, как показано ниже:

```
<div>
  <div ng-include='$route.current.contentUrl'>
    <!-- здесь находится меню -->
  </div>
  <div ng-include='$route.current.menuUrl'>
    <!-- здесь располагается содержимое -->
  </div>
</div>
```

Данный прием обеспечивает желаемый визуальный эффект, но при этом элемент DOM с меню будет повторно перерисовываться при каждом изменении маршрута, даже при навигации в пределах административного раздела, когда в этом нет никакой необходимости. Это не очень большая проблема, если шаблон меню прост и не нуждается в получении дополнительных данных со стороны сервера. Но если повторное отображение влечет за собой дорогостоящие вычисления, придется искать обходные пути, чтобы избежать их при каждом изменении маршрута.

### Отсутствует поддержка вложенных маршрутов

Другое ограничение существующей системы маршрутизации – отсутствие поддержки вложенных маршрутов. С практической точки

зрения это означает, что мы можем иметь одну, и только одну директиву ng-view. Иначе говоря, мы не можем использовать директиву ng-view внутри каждого шаблона, упоминаемого в объекте с определением маршрута. Это может стать серьезной проблемой для больших приложений, где маршруты образуют естественную иерархию. В примере приложения SCRUM также имеются такие маршруты:

- /projects: список всех проектов;
- /projects/[project id]/sprints: список спринтов указанного проекта;
- /projects/[project id]/sprints/[sprint id]/tasks: список заданий для указанного спринта в указанном проекте.

Такая схема навигации образует визуальную иерархию, которую можно было бы изобразить, как показано на рис. 6.2.

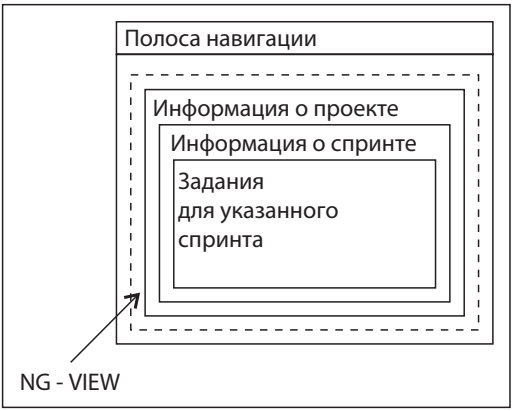


**Рис. 6.2.** Иерархическая схема навигации в приложении SCRUM

При переключении между списками заданий в разных спринтах внутри одного и того же проекта, было бы желательно сохранить данные (и пользовательский интерфейс) для данного проекта, потому что они не изменяются. К сожалению, принцип «один маршрут – одна область на экране» вынуждает выполнить повторную загрузку всей области экрана, включая шаблон отображения проекта с соответствующими ему данными. Это в свою очередь означает необходимость получать модель проекта при каждом изменении маршрута, даже если при этом она не изменяется!

Мы можем поиграть с директивой ng-include и добиться необходимого вложения визуальных элементов пользовательского ин-

терфейса, но у нас не так много возможностей, чтобы уменьшить количество попыток получить данные со стороны сервера. Мы можем лишь постараться максимально ускорить передачу данных или кешировать последующие запросы к серверу.

# Распространенные приемы использования, советы и рекомендации

Предыдущие разделы в этой главе представляют собой неплохой обзор прикладного интерфейса AngularJS, используемого для навигации в одностраничных веб-приложениях. В этом разделе мы разберем практические примеры использования этого интерфейса и обсудим наиболее эффективные приемы.

## *Обработка ссылок*

Тег ссылки (`<a>`) является в языке HTML основным инструментом создания навигационных ссылок. Фреймворк AngularJS позволяет интерпретировать эти ссылки по-особенному и в следующих подразделах рассматриваются приемы, опирающиеся на эту особенность AngularJS.

### Ссылки, реагирующие на щелчок мышью

Фреймворк AngularJS включает поддержку директивы, предотвращающей выполнение действий по умолчанию для ссылок без атрибута `href`. Это дает возможность создавать элементы, реагирующие на щелчок мышью, используя тег и директиву `ng-click`. Например, можно вызвать функцию, определенную в контексте, щелкнув на элементе DOM, определяемом, как показано ниже:

```
<a ng-click='showFAQ()'>Frequently Asked Questions</a>
```

Теги ссылок, не поддерживающие операцию навигации, довольно удобны на практике – некоторые фреймворки CSS используют их для отображения различных визуальных элементов, навигационные возможности которых не имеют никакого значения. Например, фреймворк Twitter Bootstrap CSS использует теги ссылок для отображения заголовков во вкладках.

Учитывая особенности поведения тега ссылки без атрибута `href`, у кого-то может возникнуть вопрос – какой метод использовать для

создания настоящих навигационных ссылок? В ответ можно сказать, что для этой цели годятся следующие эквивалентные методы:

```
<a href="/admin/users/list">List users</a>
```

или:

```
<a ng-click="listUsers()">List users</a>
```

где метод `listUsers` должен быть определен в контексте, как:

```
$scope.listUsers = function() {
    $location.path("/admin/users/list");
};
```

Между этими двумя методами есть очень тонкие отличия. Во-первых, ссылки с атрибутом `href` являются более дружественными, поскольку пользователь может щелкнуть на такой ссылке правой кнопкой мышки и выбрать в контекстном меню операцию открытия ссылки в новой вкладке (или окне) браузера. Вообще говоря, предпочтение следует отдавать ссылкам с атрибутам `href` или, еще лучше, использовать эквивалентный способ на основе директивы `ng-href`, упрощающей динамическое определение ссылок:

```
<a ng-href="/admin/users/{{user.$id()}}">Edit user</a>
```

## Единообразная обработка адресов URL в ссылках в стиле HTML5 и с решеткой

Служба `$location` в приложениях на основе AngularJS может быть настроена на режим интерпретации адресов URL с решеткой или в стиле HTML5. Этот выбор должен быть сделан заранее, в блоке настройки, и ссылки должны создаваться в соответствии с избранной стратегией.

Если был выбран режим интерпретации с решеткой, ссылки должны создаваться, как показано ниже (обратите внимание на символ `#`):

```
<a ng-href="#/admin/users/{{user.$id()}}">Edit user</a>
```

В режиме HTML5 ссылки будут выглядеть несколько проще:

```
<a ng-href="/admin/users/{{user.$id()}}">Edit user</a>
```

Как видите, все ссылки должны создаваться в соответствии с режимом интерпретации адресов, настроенным в службе `$locationProvider`. Проблема здесь в том, что в типичных приложениях существует достаточно много ссылок, поэтому, если впоследст-

вии будет принято решение изменить настройки `$location`, придется отыскать все ссылки в приложении и добавить (или удалить) символ решетки.

Принимайте решение о режиме интерпретации URL на как можно более раннем этапе создания приложения, иначе вам придется отыскивать и изменять все ссылки в приложении.

### Ссылки на внешние страницы

Фреймворк AngularJS предполагает, что ссылки, созданные с помощью тега `<a>`, являются внутренними для приложения, и потому должны изменять внутреннее состояние приложения, а не инициировать загрузку страницы браузером. Чаще это предположение совпадает с нашими намерениями, но иногда требуется вставить ссылку на ресурс, который должен быть загружен браузером. В режиме HTML5 ссылки на внешние ресурсы неотличимы от внутренних ссылок. Решить эту проблему можно с помощью атрибута `target`:

```
<a href="/static/form.pdf" target="_self">Download</a>
```

## Организация определений маршрутов

В крупномасштабных веб-приложениях обычное дело иметь несколько десятков маршрутов. Несмотря на очень удобный синтаксис, предоставляемый службой `$routeProvider`, определения маршрутов могут разбухать до невероятных размеров (особенно когда используется свойство `resolve`). Если в приложении определяется большое количество маршрутов и каждое определение получается достаточно объемным, мы легко можем получить огромный файл JavaScript с несколькими сотнями строк кода! Хуже того, хранение всех маршрутов в одном файле означает, что он часто будет изменяться всеми разработчиками, участвующими в проекте, что чревато большими проблемами при объединении изменений в репозитории.

### Распределение определений маршрутов по нескольким файлам

Фреймворк AngularJS не вынуждает определять все маршруты в единственном файле! Если избрать подход, предложенный в главе 2, где каждая функциональная часть определяется в собственном модуле, можно вынести маршруты, связанные с определенными разделами приложения, в соответствующие модули.

Система поддержки модулей в AngularJS позволяет определять в каждом модуле свою функцию `config`, куда можно внедрить службу `$routeProvider` и определить маршруты. Например, в примере приложения SCRUM административный модуль делится на два подмодуля: один для управления пользователями и другой для управления проектами. Каждый из этих подмодулей определяет собственные маршруты:

```
angular.module('admin-users', [])
    .config(function ($routeProvider) {
        $routeProvider.when('/admin/users', {...});
        $routeProvider.when('/admin/users/new', {...});
        $routeProvider.when('/admin/users/:userId', {...});
    });

angular.module('admin-projects', [])
    .config(function ($routeProvider) {
        $routeProvider.when('/admin/projects', {...});
        $routeProvider.when('/admin/projects/new', {...});
        $routeProvider.when('/admin/projects/:userId', {...});
    });

angular.module('admin', ['admin-projects', 'admin-users']);
```

Избрав такой путь, мы можем распределить определения маршрутов по разным модулям и избежать проблем, связанных с сопровождением одного гигантского файла с определениями всех маршрутов.

## Борьба с повторяющимся кодом в определениях маршрутов

Как упоминалось в предыдущем разделе, определения маршрутов могут разбухать до огромных размеров, когда используется свойство `resolve`. Но, если взглянуть пристальнее на код, который пишется для определения маршрутов в каждом функциональном разделе, легко можно обнаружить повторяющиеся фрагменты кода и настроек. Это особенно верно для CRUD-подобных приложений, где маршруты из разных функциональных блоков часто конструируются по похожим шаблонам.

Один из способов борьбы с повторяющимся кодом в определениях маршрутов заключается в создании собственных провайдеров, обертывающих службу `$routeProvider`. С помощью собственных провайдеров можно определить высокоуровневый API, определяющий маршруты и реализующий функциональные шаблоны, характерные для конкретного приложения.

Провайдеры, основанные на этой идее, наверняка будут отличаться в разных приложениях, поэтому мы не будем рассматривать здесь подробные инструкции по их созданию. Однако пример приложения SCRUM содержит определение такого провайдера, позволяющего существенно уменьшить объем кода, необходимого для определения маршрутов. Исходный код приложения можно получить на GitHub, тем не менее, приведем ниже пример использования собственного провайдера:

```
angular.module('admin-users', ['services.crud'])
    .config(function (crudRouteProvider) {
        crudRouteProvider.routesFor('Users')
            .whenList({
                users: function(Users) { return Users.all(); }
            })
            .whenNew({
                user: function(Users) { return new Users(); }
            })
            .whenEdit({
                user: function ($route, Users) {
                    return Users.getById($route.current.params.
itemId);
                }
            });
    })
```

Это все, что необходимо, чтобы определить полный комплект маршрутов CRUD в одном функциональном разделе!

# В заключение

Эффективная организация навигационных связей внутри приложения играет первостепенную роль, так как образует каркас, на который «навешиваются» все остальные функциональные возможности приложения. Наличие исчерпывающей структуры связей очень важно для наших пользователей, так как позволяет им с легкостью перемещаться внутри приложения. Но это важно и для нас, разработчиков, так как позволяет и помогает структурировать код приложения.

В этой главе мы узнали, что приложения на основе AngularJS способны вызывать у пользователей самые благоприятные впечатления за счет предоставления средств навигации, по своим удобствам сопоставимым со средствами, имевшимися в Web 1.0. С практической точки зрения это означает, что мы снова можем дать пользователям возможность использовать кнопки **Back** (Назад) и **Forward** (Вперед)

браузера для навигации внутри приложения. Более того, мы можем копировать в адресную строку браузера адреса URL, пригодные для создания закладок.

Служба `$route` (и ее провайдер `$routeProvider`) позволяют структурировать средства навигации так, чтобы иметь возможность обновлять только один элемент пользовательского интерфейса (ограниченную область на экране) в ответ на изменение маршрута. Встроенная служба `$route` покрывает почти все потребности большинства приложений, но если ее характеристика «один маршрут – одна область на экране» окажется слишком ограничивающей, вы сможете воспользоваться альтернативными решениями, созданными сообществом. А достаточно уверенные в себе могут даже создать собственную систему маршрутизации. Теперь, после знакомства со службой `$location` в этой главе, которая лежит в основе службы `$route`, вы готовы к этому.

Ближе к концу этой главы мы рассмотрели несколько распространенных приемов, советов и рекомендаций решения проблем, с которыми часто приходится сталкиваться при разработке крупных приложений с разветвленной системой навигации. Надеемся, что примеры, представленные в этом разделе, найдут применение в ваших приложениях! Мы настоятельно рекомендуем выносить определения маршрутов в соответствующие функциональные модули, чтобы избежать необходимости сопровождать огромные файлы, содержащие определения всех поддерживаемых маршрутов.

Темы, касающиеся маршрутизации и навигации, тесно связаны с темой обеспечения безопасности приложения. В приложениях, где не вся информация должна быть доступна широкому кругу пользователей, необходимо предусматривать меры по ограничению некоторых пользователей определенным подмножеством доступных им маршрутов. В следующей главе мы познакомимся с приемами обеспечения безопасности маршрутов, а также со множеством других аспектов поддержания безопасности приложений.

# ГЛАВА 7. Безопасность приложений

В любом веб-приложении необходимо предусматривать меры защиты конфиденциальной информации от несанкционированного доступа. Единственным, по-настоящему безопасным местом в таких приложениях является сервер. За его пределами необходимо учитывать возможность взлома программного кода и вследствие этого предусматривать проверки в точке, где данные поступают или покидают сервер. В первой части главы рассматриваются необходимые меры безопасности, которые следует предпринять и на стороне сервера, и на стороне клиента, перечисленные ниже:

- предотвращение на стороне сервера несанкционированного доступа к данным и разметке HTML;
- шифрование трафика с целью исключить перехват пакетов;
- предотвращение атак типа «межсайтовый скриптинг» (Cross-Site scripting, XSS) и «подделка межсайтовых запросов» (Cross-Site Request Forgery, XSRF);
- блокирование попыток внедрения кода JSON.

На стороне сервера меры по обеспечению безопасности должны осуществляться *всегда*, но это не значит, что они не нужны на стороне клиента – мы обязаны предусматривать подобные меры и в пользовательском интерфейсе приложения, чтобы явно показать, что пользователь может иметь доступ только к определенному кругу функциональных возможностей, соответствующих его привилегиям. Кроме того, мы обязаны реализовать ясную процедуру аутентификации, не мешающую пользователю взаимодействовать с приложением. Во второй части этой главы обсуждаются вопросы обеспечения безопасности с применением средств фреймворка AngularJS, в том числе:

- различия в обеспечении безопасности полнофункциональных клиентских приложений и более традиционных приложений, опирающихся на хранение информации на стороне сервера;

- обработка ошибок авторизации на сервере посредством перехвата HTTP-ответов;
- ограничение доступа к разделам путем обеспечения безопасности маршрутов;

# Аутентификация и авторизация на стороне сервера

Все приложения типа клиент/сервер обладают общей характерной чертой – единственным безопасным местом хранения данных является сервер. Нельзя полагаться, что клиентский код заблокирует доступ к конфиденциальной информации. Аналогично, сервер никогда не должен полагаться, что клиент проверит допустимость отправляемых им данных.

Это особенно верно для приложений на JavaScript, исходный код которых легко прочитать и даже изменить, с целью осуществить злонамеренные действия.

В настоящих веб-приложениях серверы должны обеспечивать соответствующий уровень безопасности. В нашем демонстрационном примере сервер предпринимает самые простые меры. Мы реализуем аутентификацию и авторизацию пользователей с помощью Passport, расширения для фреймворка ExpressJS. Идентификатор аутентифицированного пользователя будет храниться в шифрованном блоке данных cookie с информацией о сеансе, который будет передаваться браузеру при входе пользователя. Этот блок cookie будет отправляться серверу с каждым запросом, чтобы сервер мог аутентифицировать их.

## *Обработка неавторизованного доступа*

Когда клиент отправляет запрос по некоторому адресу URL, сервер определяет, требуется ли доступа к этому URL аутентифицировать пользователя и обладает для аутентифицированный пользователь необходимыми привилегиями. В нашем приложении мы приняли меры, чтобы сервер возвращал ошибку HTTP 401 Unauthorized (попытка неавторизованного доступа) в ответ на попытку пользователя выполнить:

- любой HTTP-запрос к базе данных коллекций, кроме запросов GET (например, запросы POST, PUT или DELETE), если пользователь не прошел процедуру аутентификации;
- любой HTTP-запрос к базе данных пользователей или проектов, кроме запросов GET, если пользователь не обладает привилегиями администратора.

При такой организации мы можем защитить данные (запросы в формате JSON) от неавторизованного доступа. При необходимости аналогичные меры можно было бы применить и к другим ресурсам, таким как разметка HTML или изображения.

### Реализация прикладного интерфейса аутентификации на стороне сервера

Для поддержки аутентификации пользователей приложения, на стороне сервера поддерживается следующий HTTP-интерфейс:

- `POST /login`: этот запрос аутентифицирует пользователя по указанным в теле POST-запроса имени пользователя и паролю;
- `POST /logout`: этот запрос закрывает сеанс работы с пользователем и удаляет cookie аутентификации;
- `GET /current-user`: в ответ на этот запрос сервер возвращает информацию о текущем пользователе.

Этого интерфейса вполне достаточно для демонстрации приемов поддержки аутентификации и авторизации в приложениях.

В коммерческих приложения требования к безопасности могут быть неизмеримо выше и сложнее, и возможно вы отдадите предпочтение инструментам аутентификации сторонних разработчиков, таким как OAuth2 (http://oauth.net/).

## Безопасность шаблонов разметки

Иногда возникают ситуации, когда нежелательно, чтобы пользователи имели доступ к некоторым шаблонам (HTML), если они не имеют соответствующих привилегий. Такие шаблоны могут содержать косвенные сведения, которые можно было бы использовать для несанкционированного доступа к конфиденциальной информации.

Простейшим решением проблемы в подобных ситуациях могла бы стать проверка авторизации запросов к таким шаблонам на стороне сервера. В первую очередь следует исключить предварительную загрузку уязвимых шаблонов при запуске, а затем – настроить на сервере проверку текущего пользователя при обращении к любому из таких шаблонов, и возврат ошибки HTTP 401 Unauthorized, если пользователь не авторизован.

Если вы собираетесь полагаться на проверку привилегий доступа запросов к шаблонам с разметкой, необходимо гарантировать невозможность их кеширования в браузере (или прокси-сервере). Для этого разметка должна возвращаться сервером со следующими HTTP-заголовками:

```
Cache-Control: no-cache, no-store, must-revalidate
Pragma       : no-cache
Expires      : 0
```

Фреймворк AngularJS кеширует все шаблоны, загруженные службой `$templateCache`. Если потребуется обеспечить безопасность шаблонов, необходимо также предотвратить кеширование шаблонов, требующих авторизованного доступа. Эти шаблоны должны удаляться из службы `$templateCache` перед аутентификацией нового пользователя, чтобы он не получил доступ к шаблонам по случайности.

Удалять шаблоны из службы `$templateCache` можно выборочно или все сразу и в любой момент времени. Например, уязвимые шаблоны можно удалять при выходе из маршрута с ограниченным доступом или по завершении сеанса. Однако подобная организация чистки кеша может оказаться сложной в реализации, поэтому самый, пожалуй, надежный способ – полностью перезагрузить страницу, то есть обновить все содержимое в окне браузера в момент завершения сеанса.

Перезагрузка приложения, путем обновления страницы в браузере, имеет дополнительное преимущество – уничтожение всех данных, которые могли хранить самые разные службы AngularJS.

# Противостояние нападениям

Чтобы обеспечить безопасный доступ для добропорядочных пользователей, между сервером и браузером должен быть определенный элемент доверия. К сожалению, существует множество разновиднос-

тей нападений, когда злоумышленник может использовать это доверие в своих интересах. При соответствующих настройках на сервере, фреймворк AngularJS в состоянии защитить от таких нападений.

## Предотвращение перехвата cookie («атака через посредника»)

Всякий раз, когда между клиентом и сервером осуществляется передача данных по протоколу HTTP, существует вероятность вмешательства третьей стороны с целью прочитать конфиденциальную информацию или, хуже того, cookies авторизации, чтобы перехватить управление сеансом и получить возможность обращаться к серверу от вашего имени. Нападения такого вида часто называют «атака через посредника» («man-in-the-middle»), http://ru.wikipedia.org/wiki/Человек_посередине. Самый простой способ предотвращения таких нападений – использование протокола HTTPS вместо HTTP.

Любое приложение, выполняющее передачу конфиденциальной информации между приложением и сервером, должно использовать протокол HTTPS, чтобы обеспечить шифрование данных.

Шифруя соединение посредством протокола HTTPS, мы исключаем возможность чтения конфиденциальных данных посторонними лицами, оказавшимися между сервером и клиентом, а также не позволяем неавторизованным пользователям получать блоки cookies аутентификации, с помощью которых они могли бы перехватить управление сеансом.

В нашем демонстрационном приложении запросы от нашего сервера к серверу MongoLab DB уже отправляются по протоколу HTTPS. Чтобы обеспечить полную защиту от перехвата конфиденциальной информации, необходимо также обеспечить использование протокола HTTPS для взаимодействий между клиентом и сервером. Часто для этого достаточно организовать на сервере прием, а на клиенте – отправку запросов по протоколу HTTPS.

На стороне сервера данная возможность зависит от используемой серверной платформы, что далеко выходит за рамки этой книги. Но в `Node.js` для этой цели можно использовать модуль `https`, как показано ниже:

```
var https = require('https');
var privateKey =
```

```
        fs.readFileSync('cert/privatekey.pem').toString();
    var certificate =
        fs.readFileSync('cert/certificate.pem').toString();
    var credentials = {key: privateKey, cert: certificate};
    var secureServer = https.createServer(credentials, app);
    secureServer.listen(config.server.securePort);
```

На стороне клиента достаточно обеспечить независимость от протокола HTTP адресов URL, используемых для связи с сервером. Проще всего сделать это – исключить упоминание протокола из всех адресов URL.

```
angular.module('app').constant('MONGOLAB_CONFIG', {
    baseUrl: '/databases/',
    dbName: 'ascrum'
});
```

Дополнительно необходимо гарантировать передачу cookie аутентификации только по протоколу HTTPS. Добиться этого можно, установив параметры `httpOnly` и `secure` в значение `true` при создании блока cookie на сервере.

## Предотвращение нападений вида «межсайтовый скриптинг»

Нападения вида «межсайтовый скриптинг» (Cross-Site Scripting, XSS) заключаются во внедрении злонамеренного сценария в вебстраницу при просмотре ее другим пользователем. Наибольший вред от таких нападений может быть нанесен, если внедренный сценарий сможет обращаться к серверу, потому что сервер будет считать эти обращения аутентифицированными и выполнять их.

Существует большое разнообразие нападений XSS. Чаще всего используется разновидность, основанная на отображении получаемого из сети содержимого без надлежащего экранирования, предотвращающего интерпретацию специально подготовленной разметки HTML. В следующем разделе рассказывается, как решить эту проблему на стороне клиента, но вы так же должны гарантировать полную проверку данных, поставляемых пользователем, на стороне сервера, прежде чем сохранять их или возвращать обратно клиенту.

### Защита содержимого HTML в выражениях AngularJS

AngularJS экранирует все теги HTML в тексте, отображаемом с помощью директивы `ng-bind` или механизма интерполяции шаблона (то есть текста в `{{`фигурных скобках`}}`). Например, для модели:

```
$scope.msg = 'Hello, <b>World</b>!';
```

и шаблона с разметкой:

```
<p ng-bind="msg"></p>
```

механизм отображения выполнит экранирование тегов <b> так, что они будут отображаться как обычный текст, а не как разметка:

```
<p>Hello, &lt;b&gt;World&lt;/b&gt;!</p>
```

Такой подход обеспечивает неплохую защиту от нападений XSS. Если же вам действительно понадобится отображать текст, содержащий разметку HTML, тогда придется выказать полное доверие и использовать директиву ng-bind-html-unsafe, либо организовать чистку текста, загрузив модуль ngSanitize и задействовав директиву ng-bind-html.

### Небезопасное отображение разметки HTML

Следующая привязка обеспечит интерпретацию браузером тегов <b>:

```
<p ng-bind-html-unsafe="msg"></p>
```

### Чистка разметки HTML

В AngularJS имеется еще одна директива, обеспечивающая выборочную чистку одних тегов HTML и интерпретацию других, – директива ng-bind-html. Она используется так же, как ее небезопасный эквивалент:

```
<p ng-bind-html="msg"></p>
```

С точки зрения экранирования, директива ng-bind-html являет собой компромисс между директивой ng-bind-html-unsafe (позволяющей интерпретировать все теги HTML) и директивой ng-bind (запрещающей интерпретацию любых тегов HTML). Она может пригодиться в ситуациях, когда желательно разрешить пользователям вводить некоторые теги HTML.

Для чистки используется достаточно обширный «белый список» безопасных тегов HTML. В число основных экранируемых входят теги <script> и <style>, а также атрибуты, принимающие адреса URL, такие как href, src и usemap.

Директива ng-bind-html располагается в отдельном модуле (ngSanitize) и требует подключения дополнительного файла

angular-sanitize.js. Если планируется использовать директиву ng-bind-html, необходимо объявить зависимость от модуля ngSanitize, как показано в следующем фрагменте:

```
angular.module('expressionsEscaping', ['ngSanitize'])
    .controller('ExpressionsEscapingCtrl', function ($scope) {
        $scope.msg = 'Hello, <b>World</b>!';
    });
```

Директива ng-bind-html использует службу $sanitize, реализация которой так же находится в модуле ngSanitize. Эта служба является функцией, принимающей строку и возвращающей экранированную ее версию:

```
var safeDescription = $sanitize(description);
```

Если только вы не используете имеющиеся унаследованные системы (например, системы управления содержимым, серверные компоненты, отправляющие клиенту фрагменты разметки HTML и другие), избегайте использования разметки в моделях. Такая разметка не может содержать директивы AngularJS и требует использования директивы ng-bind-html-unsafe или ng-bind-html для достижения желаемого результата.

## Предотвращение внедрения данных в формате JSON

Существует разновидность нападений, выражающаяся во внедрении данных в формате JSON, позволяющая сторонним злонамеренным веб-сайтам получить доступ к конфиденциальным ресурсам в формате JSON, если они возвращаются в виде массивов JSON. Это достигается за счет загрузки в веб-страницу данных в формате JSON в виде сценария с последующим его выполнением. Дополнительную информацию по этой теме можно найти по адресу: http://haacked.com/archive/2008/11/20/anatomy-of-a-subtle-json-vulnerability.aspx.

Служба $http имеет встроенную реализацию защиты от нападений этого вида. Чтобы предотвратить выполнение браузером данных в формате JSON, возвращаемых небезопасным ресурсом, можно настроить на сервере добавление ко всем JSON-запросам префикса ")]}',\n", который является недопустимым кодом на языке JavaScript и не может быть выполнен. Служба $http автоматически отбрасывает этот префикс, если он присутствует в JSON-ответе. Например, если ресурс возвращает следующий массив:

```
['a,'b','c']
```

он оказывается уязвим для нападения внедрением JSON. Вместо этого данные должны возвращаться в виде:

```
)]}',
['a','b','c']
```

Этот фрагмент является недопустимым кодом на JavaScript. Он не может быть выполнен браузером и поэтому оказывается неуязвимым для описанных выше нападений. Служба `$http` автоматически отбрасывает недопустимый префикс, если он присутствует, и возвращает допустимые данные в формате JSON в объекте ответа.

## Предотвращение подделки межсайтовых запросов

В любом приложении, где сервер оказывает доверие зарегистрированному пользователю и позволяет выполнять некоторые операции на сервере, полагаясь на это доверие, существует вероятность, что другие сайты смогут получить доступ к этим операциям и выполнять их от вашего имени. Эта разновидность нападений называется «подделка межсайтовых запросов» (Cross-Site Request Forgery, XSRF). Если пользователь посетит злонамеренный сайт, когда он уже прошел процедуру аутентификации на защищенном сайте, веб-страница со злонамеренного сайта сможет послать защищенному сайту запрос, поскольку в данный момент вы считаетесь аутентифицированным пользователем.

Такие нападения часто реализуются в виде мошеннического атрибута `src` в теге `<img>`, который пользователь может загрузить по неосторожности при переходе на злонамеренную страницу, не закрывая сеанс работы с защищенным сайтом. Когда браузер попытается загрузить изображение, он фактически выполнит запрос к защищенному сайту.

```
<img src="http://my.securesite.com/createAdminUser?username=badguy">
```

Чтобы устранить эту проблему, сервер может передавать браузеру секретный ключ, доступный только сценарию на JavaScript, выполняющемуся в браузере, и недоступный в атрибуте `src`. При обращении к серверу этот ключ должен включаться в заголовки запросов, чтобы подтвердить аутентичность пользователя.

Служба `$http` уже реализует это решение для защиты от нападений подобного рода. Чтобы его активизировать, необходимо на сто-

роне сервера обеспечить передачу в сеансовом блоке cookie параметра с именем `XSRF-TOKEN`, в ответ на первый GET-запрос приложения. Значение этого параметра должно быть уникальным для данного сеанса.

На стороне клиента служба `$http` будет извлекать этот ключ из cookie и добавлять его в каждый HTTP-запрос в виде заголовка `X-XSRF-TOKEN`. Сервер должен проверять ключ в каждом запросе и блокировать доступ, если он окажется недействительным. Разработчики AngularJS рекомендуют использовать этот ключ в дополнение к cookie аутентификации, чтобы повысить защищенность приложения.

# Обеспечение безопасности на стороне клиента

В оставшейся части этой главы мы посмотрим, что следует предпринять в нашем приложении на основе AngularJS, выполняющемся под управлением браузера, чтобы обеспечить его безопасность и дать пользователям чувство защищенности с помощью процедур аутентификации и авторизации.

По умолчанию фреймворк AngularJS не содержит реализацию процедур аутентификации и авторизации. В примере приложения нам пришлось реализовать дополнительные службы и директивы, которые можно было бы использовать в шаблонах и контроллерах для отображения информации, имеющей отношение к обеспечению безопасности, обрабатывать ошибки авторизации и контролировать открытие и закрытие сеанса.

На рис. 7.1 изображена диаграмма с двумя элементами пользовательского интерфейса, `login-toolbar` и `login-form`, каждый из которых основан на использовании службы `security`. Служба `security`, в свою очередь, использует службу `$dialog` для создания модального диалога в элементе `login-form`.

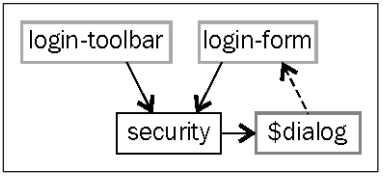


**Рис. 7.1.** Функциональная диаграмма поддержки безопасности в примере приложения SCRUM

## Служба security

Служба `security` – это созданный нами компонент, реализующий основной прикладной интерфейс для управления открытием и закрытием сеанса, и для получения информации о текущем пользователе. Эту службу можно внедрять в контроллеры и директивы, которые в свою очередь могут добавлять перечисленные ниже свойства и методы в объект контекста, чтобы обеспечить доступ к ним из шаблонов.

- `currentUser`: свойство с информацией о текущем, аутентифицированном пользователе;
- `getLoginReason()`: этот метод возвращает локализованное сообщение, объясняющее необходимость аутентификации, например: «Текущий пользователь не авторизован».
- `showLogin()`: этот метод отображает форму аутентификации. Вызывается в ответ на щелчок на кнопке **Log in** (Войти), а также при получении HTTP-ответа HTTP 401 Unauthorized;
- `login(email, password)`: этот метод отправляет указанные параметры на сервер для аутентификации. Вызывается, когда пользователь отправляет форму аутентификации. В случае успеха форма закрывается и повторяется попытка выполнить запрос, вызвавший появление этой формы.
- `logout(redirectTo)`: этот метод закрывает текущий сеанс работы с пользователем и выполняет переход по указанному маршруту. Вызывается в ответ на щелчок на кнопке **Log out** (Выйти).
- `cancelLogin(redirectTo)`: этот метод прерывает попытку открыть сеанс, аннулирует все неавторизованные запросы, приведшие к появлению формы аутентификации, и выполняет переход по указанному маршруту. Вызывается, когда пользователь закрывает форму аутентификации или щелкает на кнопке **Cancel** (Отмена).

## Отображение формы аутентификации

Нам необходимо дать пользователю возможность открыть сеанс работы с приложением. Эту возможность реализует форма аутентификации (рис. 7.2). Она состоит из шаблона (`security/login/form.tpl.html`) и контроллера (`LoginFormController`). Когда возникает потребность в аутентификации, служба `security` открывает (`showLogin()`) и закрывает (`login()` или `cancelLogin()`) форму автоматически.

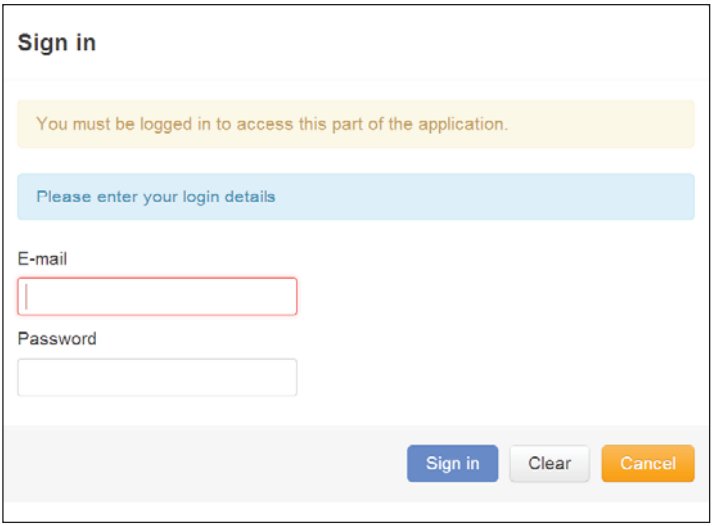


**Рис. 7.2.** Форма аутентификации

Для этой цели мы использовали службу `$dialog` из проекта AngularUI bootstrap. Эта служба позволяет отобразить форму в виде модального диалога, достаточно лишь передать ей шаблон и контролер формы.

Получить более подробную информацию о службе `$dialog` можно на веб-сайте: http://angularui.github.io/bootstrap/#/dialog.

В нашей службе `security` имеется два вспомогательных метода: `openLoginDialog()` и `closeLoginDialog()`.

```
var loginDialog = null;
function openLoginDialog() {
    if ( !loginDialog ) {
        loginDialog = $dialog.dialog();
        loginDialog.open(
            'security/login/form.tpl.html',
            'LoginFormController')
        .then(onLoginDialogClose);
    }
}

function closeLoginDialog(success) {
    if (loginDialog) {
        loginDialog.close(success);
        loginDialog = null;
    }
}
```

Чтобы открыть диалог, мы вызываем метод `openLoginDialog()` и передаем ему URL шаблона формы аутентификации (`security/login/form.tpl.html`) и имя контроллера (`LoginFormController`). Этот метод возвращает объект отложенного результата, который получит фактическое значение при закрытии диалога вызовом метода `closeLoginDialog()`. Когда диалог закрывается, мы вызываем метод `onLoginDialogClose()`, выполняющий заключительные операции, и затем выполняем тот или иной код, в зависимости от успеха процедуры аутентификации.

В шаблоне и контроллере нет ничего необычного – они реализуют простейшую форму ввода адреса электронной почты и пароля, и передают их службе `security`. Щелчок на кнопке **Sign in** (Зарегистрироваться) вызывает метод `security.login()`, а щелчок на кнопке **Cancel** (Отмена) – метод `security.cancelLogin()`.

## Создание меню и панелей инструментов с поддержкой системы безопасности

Чтобы не вызывать у пользователя отрицательных эмоций, мы не должны отображать элементы управления, которыми пользователь не сможет воспользоваться без соответствующих привилегий. Такое сокрытие элементов не является серьезным препятствием для наиболее решительных пользователей и предназначено, только чтобы не смущать своим присутствием пользователей, не обладающих достаточными привилегиями. Прием выборочного отображения элементов управления в пользовательском интерфейсе является весьма распространенной практикой. Чтобы упростить создание шаблонов, реагирующих на текущее состояние аутентификации и авторизации, можно создать службу `currentUser`.

### Сокрытие элементов меню

Мы должны отображать только те элементы меню, которые соответствуют привилегиям текущего пользователя.


**Рис. 7.3.** Навигационное меню приложения SCRUM

Навигационное меню приложения (рис. 7.3) включает множество директив `ng-show` в элементах, требующих проверки привилегий текущего пользователя для их отображения.

```
<ul class="nav" ng-show="isAuthenticated()">
  <li ...><a href="/projects">My Projects</a></li>
  <li ... ng-show="isAdmin()">
    <a ... >Admin ...</a>
    <ul ...>
      <li><a ... >Projects</a></li>
      <li><a ... >Users</a></li>
    </ul>
  </li>
</ul>
```

Как видите, здесь реализовано сокрытие целого каскада меню, если пользователь не аутентифицирован. Кроме того, некоторые элементы меню скрываются, если пользователь не обладает привилегиями администратора.

## Создание панели инструментов управления сеансом

Мы можем создать простую директиву `login-toolbar` многократного пользования, которая просто отображает кнопку **Log in** (Войти), если пользователь еще не аутентифицирован, или имя текущего пользователя и кнопку **Log out** (Выйти) в противном случае (рис. 7.4).


**Рис. 7.4.** Панель инструментов управления сеансом

Ниже представлен шаблон с разметкой для этой директивы. Он внедряет службы `currentUser` и `security` в контекст директивы, благодаря чему обеспечивается возможность отображения информации о пользователе, а также отображения/сокрытия кнопок **Log in** (Войти) и **Log out** (Выйти).

```
<ul class="nav pull-right">

<li class="divider-vertical"></li>

<li ng-show="isAuthenticated()">
  <a href="#">{{currentUser.firstName}} {{currentUser.lastName}}</a>
</li>

<li ng-show="isAuthenticated()">
```

```
    <form class="navbar-form">
      <button class="btn" ng-click="logout()">Log out</button>
    </form>
  </li>

  <li ng-hide="isAuthenticated()">
    <form class="navbar-form">
      <button class="btn" ng-click="login()">Log in</button>
    </form>
  </li>

  </ul>
```

# Поддержка аутентификации и авторизации на стороне клиента

Обеспечение безопасности полнофункциональных веб-приложений на основе AngularJS существенно отличается от обеспечения безопасности традиционных веб-приложений, опирающихся на хранение информации на стороне сервера. Это отличие определяет, когда и как должны выполняться аутентификация и авторизация пользователей.

Традиционные веб-приложения обычно не хранят информацию о своем состоянии в браузере. Чтобы выполнить некоторую операцию, часто приходится производить полный цикл-запроса и отображения новой страницы. Благодаря этому, получив очередной запрос, сервер сможет определить уровень привилегий пользователя и при необходимости переадресовать его на страницу аутентификации.

В традиционном веб-приложении мы могли бы просто отправить браузеру некоторую форму аутентификации и в случае успеха вернуть первоначально запрошенную страницу.

В полнофункциональных веб-приложениях не принято отправлять полную страницу в ответ на каждую операцию. Такие приложения хранят всю необходимую информацию о своем состоянии на стороне клиента и обмениваются с сервером только данными. Сервер ничего не знает о текущем состоянии клиента, что усложняет традиционную реализацию переадресации на первоначально запрошенную страницу после отправки формы аутентификации. Но есть другое решение: клиент может отправлять на сервер информацию о своем состоянии,

а после благополучной аутентификации сервер может возвращать эту информацию обратно клиенту, чтобы тот продолжил работу с того места, где она была прервана.

## Обработка ошибок авторизации

Когда сервер отвергает обработку неавторизованного запроса, он возвращает код ошибки HTTP 401 Unauthorized. В этом случае мы должны дать пользователю возможность пройти процедуру аутентификации и повторить запрос. Запрос может быть частью какого-то сложного процесса, протекающего на стороне, не имеющего адреса URL, по которому его можно было бы идентифицировать. При переадресации на страницу аутентификации мы должны были бы приостановить выполнение текущей операции, дать пользователю возможность подтвердить свои полномочия и позволить ему продолжить работу с того же места, где она была прервана.

Вместо этого мы организуем перехват ответов сервера с кодом ошибки HTTP 401 Unauthorized до того, как они будут возвращены вызывающему коду. Перехватив такой ответ, мы приостановим выполнение приложения, запустим процедуру аутентификации и в случае успешного ее завершения повторим запрос, потерпевший неудачу. Этот порядок действий изображен на рис. 7.5.

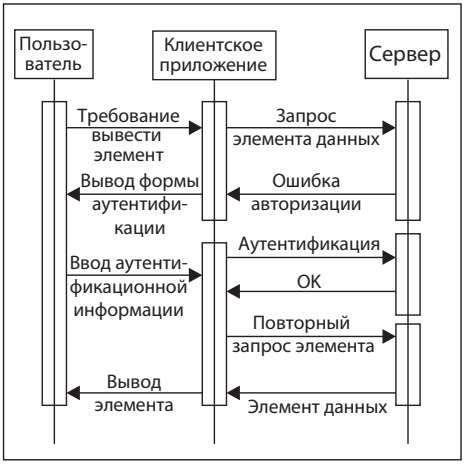


**Рис. 7.5.** Схема аутентификации пользователя в процессе выполнения некоторой операции

## Перехват ответов

Не забывайте, что служба `$http` возвращает отложенный результат, который получает фактическое значение после получения ответа от сервера. Огромное достоинство отложенных результатов в том, что их можно объединять в цепочки, преобразовывать фактические значения, возвращаемые ими, и даже возвращать совершенно другие отложенные результаты. Как описывается в главе 3, AngularJS позволяет определять свои функции-обработчики (перехватчики) для обработки ответов от сервера перед передачей их вызывающей программе.

### Перехватчики HTTP-ответов

Перехватчик ответов – это обычная функция, принимающая объект отложенного результата, ожидающий получения ответа от сервера, и возвращающая такой же объект отложенного результата. Для каждого HTTP-запроса служба `$http` передает объект отложенного результата каждому перехватчику, позволяя им изменить этот объект, прежде чем он будет возвращен вызывающей программе.

В общем случае функция-перехватчик вызывает метод `then()`, для присоединения к цепочке обработчиков объекта отложенного результата, после чего создает новый объект отложенного результата и возвращает его. Внутри обработчиков можно читать и изменять объект ответа, например – заголовки и данные, как показано в следующем примере:

```
function myInterceptor(promise) {
    return promise.then(function(response) {
        if ( response.headers()['content-type'] == "text/plain") {
            response.data = $sanitize(response.data);
        };
        return response;
    });
}
```

Этот перехватчик проверяет заголовок `content-type` ответа. Если он имеет значение `text/plain`, выполняется экранирование данных в теле ответа и вызывающей программе возвращается объект отложенного результата.

## Создание службы securityInterceptor

Мы создадим службу `securityInterceptor` для работы с объектами отложенного результата, содержащими объект ответа сервера. В нашем перехватчике мы проверим, не является ли ответ ошибкой

авторизации с кодом 401. И, если сервер пришлет ответ с этим кодом ошибки, мы создадим новый объект отложенного результата для повторной попытки выполнить первоначальный запрос, и вернем его вызывающей программе вместо оригинала.

Суть этой идеи раскрыта в отличной статье Витольда Щербы (Witold Szczerba): http://www.espeo.pl/2012/02/26/authentication-in-angularjs-application.

Мы реализуем `securityInterceptor` в виде службы, а затем добавим ее в массив `responseInterceptors` службы `$http`.

```
.config(['$httpProvider', function($httpProvider) {
    $httpProvider.responseInterceptors.push(
        'securityInterceptor');
}]);
```

Добавляться в массив должно имя службы `securityInterceptor` в виде строки, а не сам объект, потому что он зависит от служб, недоступных в блоке `config`.

Так как перехватчик `securityInterceptor` реализован как служба, это дает возможность внедрять в нее другие службы.

Мы не можем внедрить службу `$http` непосредственно в перехватчик запросов, потому что в результате возникнет циклическая зависимость. Вместо этого мы будем внедрять службу `$injector` и использовать ее для доступа к службе `$http` в момент вызова.

```
.factory('securityInterceptor',
    ['$injector', 'securityRetryQueue',
     function($injector, queue) {
        return function(promise) {
            var $http = $injector.get('$http');
            return promise.then(null, function(response) {
                if(response.status === 401) {
                    promise = queue.pushRetryFn('unauthorized-server',
                                function() {return $http(response.
config); }
                                );
                }
                return promise;
            });
        };
    }])
```

Наш перехватчик просматривает ответы с кодом 401. Достигается это за счет передачи обработчика методу `then()` во втором аргументе. Передавая `null` в первом аргументе, мы показываем, что не заинтересованы в обработке результатов успешного завершения отложенного задания.

Когда запрос потерпит неудачу с кодом ошибки 401, перехватчик создаст элемент в очереди службы `securityRetryQueue`, о которой рассказывается ниже. Эта служба повторит запрос, когда наступит время обработать очередь после успешной аутентификации.

Важно отметить, что обработчики отложенных результатов могут возвращать либо фактическое значение, либо объект отложенного результата:

- если обработчик вернет фактическое значение, оно будет передано непосредственно следующему обработчику в цепочке;
- если обработчик вернет объект отложенного результата, следующий обработчик в цепочке не будет вызван, пока новое отложенное задание не будет выполнено (успехом или неудачей).

В нашем случае, когда перехватчик получит ответ с кодом ошибки 401, он фактически вернет новый объект отложенного результата, извлекающий из очереди `securityRetryQueue` элемент, который представляет повторный запрос. Этот новый объект отложенного результата получит фактическое значение, если `securityRetryQueue` выполнит повторный запрос и он увенчается успехом, или завершится неудачей, если `securityRetryQueue` отменит повторную попытку, или будет получена ошибка с каким-либо другим кодом.

Пока вызывающая программа терпеливо ждет ответа от сервера, мы можем отобразить диалог аутентификации, позволить пользователю аутентифицировать себя и выполнить элементы, хранящиеся в очереди. Когда вызывающая программа получит успешный ответ, она сможет продолжить работу, как если бы пользователь был аутентифицирован давным-давно.

## Создание службы securityRetryQueue

Служба `securityRetryQueue` предоставляет место для хранения информации обо всех запросах, которые необходимо будет повторить, когда аутентификация пользователя завершится успехом. Фактически она является списком, куда добавляются функции (реализующие повторные попытки) вызовом `pushRetryFn()`. Обработка элементов

списка осуществляется вызовом `retryAll()` или `cancelAll()`. Ниже представлен метод `retryAll()`.

```
retryAll: function() {
    while(retryQueue.length) {
        retryQueue.shift().retry();
    }
}
```

Каждый элемент в очереди должен обладать двумя методами, `retry()` и `cancel()`. Метод `pushRetryFn()` упрощает настройку этих объектов, как показано ниже:

```
pushRetryFn: function(reason, retryFn) {
    var deferredRetry = $q.defer();
    var retryItem = {
        reason: reason,
        retry: function() {
            $q.when(retryFn()).then(function(value) {
                deferredRetry.resolve(value);
            },function(value) {
                deferredRetry.reject(value);
            });
        },
        cancel: function() {
            deferredRetry.reject();
        }
    };
    service.push(retryItem);
    return deferredRetry.promise;
}
```

Эта функция возвращает объект отложенного результата для выполнения повторной попытки, реализуемой функцией `retryFn`. Объект отложенного результата будет вычислен при извлечении элемента из очереди.

Нам потребовалось создать новый собственный объект отложенного задания для данного объекта отложенного результата, потому что его вычисление происходит не в результате получения ответа от сервера, а как следствие вызова метода `retry()` или `cancel()`.

## Передача извещения службе security

Последним фрагментом мозаики, которую нам требуется сложить, является передача извещения службе `security` о добавлении нового элемента. В нашей реализации мы просто предусматриваем в служ-

бе `securityRetryQueue` метод `onItemAdded()`, который вызывается при добавлении каждого нового элемента в очередь:

```
push: function(retryItem) {
    retryQueue.push(retryItem);
    service.onItemAdded();
}
```

Служба `security` должна переопределить его собственной реализацией, чтобы иметь возможность реагировать на ошибки авторизации, как показано в следующем фрагменте:

```
securityRetryQueue.onItemAdded = function() {
    if (securityRetryQueue.hasMore() ) {
        service.showLogin();
    }
};
```

Этот код находится в службе `security`, а переменная `service` – это сама служба `security`.

# Предотвращение переходов по защищенным маршрутам

Предотвращение доступа к защищенным маршрутам с использованием кода на стороне клиента не обеспечивает необходимый уровень безопасности. Единственный надежный способ, гарантирующий невозможность попадания неавторизованных пользователей в защищенные разделы приложения, требует перезагрузки страницы, чтобы дать серверу возможность отказать в доступе. Но перезагрузка страницы не является идеальным вариантом, потому что сводит на нет все достоинства полнофункциональных клиентских приложений.

Перезагрузка страницы для обеспечения безопасности обычно является не самым лучшим решением в полнофункциональных клиентских приложениях. Его можно использовать, если имеется четкая грань между разделами приложения. Например, если приложение в действительности состоит из двух приложений, каждое из которых предъявляет свои требования к аутентификации, их можно связать с разными адресами URL и проверять на сервере привилегии пользователя, прежде чем разрешать загружать то или иное приложение.

На деле, так как мы можем защитить данные, отображаемые перед пользователем, не так важно блокировать доступ к маршрутам с пере-

адресацией на сервер. Вместо этого можно просто запретить переход неавторизованного пользователя к защищенному маршруту на стороне клиента, в момент изменения маршрута.

Это не является средством защиты. В действительности это лишь способ помочь пользователю корректно пройти аутентификацию, если он попытается перейти по адресу, требующему авторизованного доступа.

## Использование функций в свойстве resolve маршрутов

Каждый маршрут, определенный с помощью провайдера `$routeProvider` службы `$route`, может содержать множество функций в своем свойстве `resolve`. Каждая из этих функций возвращает объект отложенного результата, который должен быть успешно вычислен, чтобы попытка перехода по данному маршруту также увенчалась успехом. Если какое-либо отложенное задание завершится неудачей, попытка выполнить переход по указанному маршруту будет отвергнута.

Простейший подход к проверке авторизации заключается в определении маршрута с функцией, которая завершается успехом, только если текущий пользователь обладает необходимыми привилегиями.

```
$routeProvider.when('/admin/users', {
    resolve: [security, function requireAdminUser(security) {
        var promise = security.requestCurrentUser();
        return promise.then(function(currentUser) {
            if ( !currentUser.isAdmin() ) {
                return $q.reject();
            }
            return currentUser;
        });
    }]
});
```

В этом примере создается отложенное задание, определяющее привилегии текущего пользователя обращением к службе `security`, которое завершается неудачей, если пользователь не обладает привилегиями администратора. Недостаток такого решения заключается в отсутствии для пользователя возможности выполнить аутентификацию. Оно просто блокирует доступ к маршруту.

Так же, как при обработке ошибки с кодом 401, мы можем реализовать повторные попытки перехода в случае ошибки. Для этого достаточно всего лишь добавить задание, реализующее повторную попытку, в очередь службы `securityRetryQueue`, которое будет инициировано после успешной процедуры аутентификации.

```
function requireAdminUser(security, securityRetryQueue) {
    var promise = security.requestCurrentUser();
    return promise.then(function(currentUser) {
        if ( !currentUser.isAdmin() ) {
            return securityRetryQueue.pushRetryFn(
                'unauthorized-client',
                requireAdminUser);
        }
    });
}
```

Если теперь пользователь, не обладающий привилегиями администратора, попытается выполнить переход по маршруту, имеющему этот метод в свойстве `resolve`, в очередь службы `securityRetryQueue` будет добавлен новый элемент. Добавление элемента вынудит службу `security` отобразить форму аутентификации, с помощью которой пользователь сможет получить привилегии администратора. Когда процедура аутентификации завершится успехом, произойдет повторный вызов метода `requireAdminUser` и в случае успешного его завершения, будет выполнен переход по данному маршруту.

## Создание службы authorization

Для поддержки методов в свойстве `resolve` объектов-определений маршрутов, мы создали службу `authorization` с методами для проверки привилегий текущего пользователя.

В более сложном приложении можно создать службу, настраиваемую списком ролей и привилегий, необходимых для организации системы безопасности приложения.

В нашем приложении эта служба имеет очень простую реализацию и включает всего два метода: `requireAuthenticatedUser()` и `requireAdminUser()`, которые можно описать так:

- `requireAuthenticatedUser()`: возвращает отложенный результат, который будет благополучно вычислен, только когда пользователь успешно пройдет процедуру аутентификации;

- `requireAdminUser()`:возвращает отложенный результат, который будет благополучно вычислен, только когда пользователь успешно подтвердит свои привилегии администратора.

Так как эти методы определены в службе, недоступной непосредственно при настройке `$routeProvider`, их обычно приходится вызывать внутри функции, заключенной в массив:

```
['securityAuthorization', function(securityAuthorization) {
    return securityAuthorization.requireAdminUser();
}]
```

Чтобы избежать повторений, мы можем просто поместить этот массив в реализацию провайдера службы `authorization`, как показано ниже:

```
.provider('securityAuthorization', {
    requireAdminUser: [
        'securityAuthorization',
        function(securityAuthorization) {
            return securityAuthorization.requireAdminUser();
        }
    ],
    $get: [
        'security',
        'securityRetryQueue',
        function(security, queue) {
            var service = {
                requireAdminUser: function() {
                },
            };
            return service;
        }
    ]
});
```

Фактическая реализация службы `authorization` находится в свойстве `$get`. Вспомогательные методы проверки привилегий для доступа к маршруту, такие как `requireAdminUser()`, мы реализовали как методы провайдера. При настройке маршрутов мы можем просто внедрить провайдера и затем использовать его методы:

```
config([
    'securityAuthorizationProvider',
    function (securityAuthorizationProvider) {
        $routeProvider.when('/admin/users', {
            resolve: securityAuthorizationProvider.
requireAdminUser
        });
    }
])
```

Теперь при попытке изменить маршрут, будет выполнена проверка привилегий пользователя и при необходимости ему будет предоставлена возможность аутентифицировать себя.

# В заключение

В этой главе мы рассмотрели некоторые общие проблемы безопасности, которые приходится решать в полнофункциональных клиентских приложениях, и провели параллели с традиционными веб-приложениями, опирающимися на хранение информации на стороне сервера. В частности, несмотря на то, что на стороне сервера проверка данных и привилегий должны выполняться обязательно, клиент и сервер все же могут сотрудничать друг с другом, чтобы эффективно противостоять нападениям. Мы реализовали несколько служб и директив для поддержки безопасности нашего приложения. Узнали, как служба `$http`, опирающаяся на использование объектов отложенных результатов, позволяет перехватывать HTTP-ответы сервера с кодом ошибки 401, благодаря чему можно организовать аутентификацию пользователя, не прерывая и не перезапуская логику выполнения приложения. В заключение мы добавили в реализацию поддержки маршрутов дополнительные функции, выполняющие проверку привилегий пользователя перед попыткой выполнить переход в защищенный раздел приложения.

Далее мы посмотрим, как научить браузер некоторым трюкам, создавая собственные директивы, позволяющие разрабатывать пользовательские интерфейсы в более декларативном стиле.

# ГЛАВА 8. Создание собственных директив

Вы можете далеко уйти, используя только контроллеры и директивы, уже имеющиеся в составе AngularJS, но рано или поздно наступит момент, когда вам потребуется научить свой браузер некоторым новым трюкам, создавая собственные директивы. Ниже перечислены основные причины, объясняющие необходимость создания собственных директив:

- необходимость в выполнении операций непосредственно с деревом DOM, например, с помощью JQuery;
- необходимость провести рефакторинг кода, чтобы избавиться от повторяющихся фрагментов;
- желание определить новые элементы для использования в разметке HTML нетехническими специалистами, например, дизайнерами.

В этой главе вы познакомитесь с особенностями разработки собственных директив AngularJS. Директивы могут появляться в самых разных местах и выполнять самые разные функции. Далее вы увидите:

- как определять директивы;
- примеры наиболее типичных директив и особенности их реализации;
- директивы, опирающиеся друг на друга;
- как тестировать директивы.

## Что такое директива AngularJS?

Директивы являются, пожалуй, самой мощной особенностью AngularJS. Они – своеобразный «клей», связывающий логику приложения с объектной моделью документа HTML (DOM). На рис. 8.1 изобра-

жена диаграмма, иллюстрирующая место директив в архитектуре приложений на основе AngularJS:

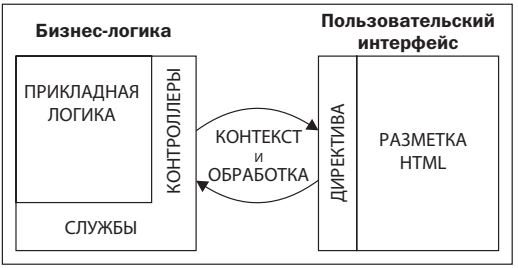


**Рис. 8.1.** Директивы в архитектуре приложений на основе AngularJS

Расширяя и изменяя интерпретацию разметки HTML браузером, директивы позволяют разработчикам или дизайнерам сконцентрировать свое внимание на том, что должно делать приложение, и выражать свои намерения максимально декларативным способом, а не низкоуровневым кодом, реализующим взаимодействия с деревом DOM. Такой подход ускоряет разработку, делает приложение более простым в сопровождении и, что самое главное, превращает программирование в удовольствие!

Директивы AngularJS привносят новый смысл и элементы поведения в разметку HTML приложения, и скрывают в себе все низкоуровневые операции с деревом DOM, обычно выполняемые с помощью библиотеки jQuery или jqLite, входящей в состав AngularJS.

Если загрузить библиотеку jQuery до загрузки фреймворка AngularJS, для выполнения операций с деревом DOM он будет использовать jQuery. В противном случае будет использоваться внутренняя, минимальная версия библиотеки jQuery, которую часто называют как jqLite.

Основная задача директив – изменение структуры дерева DOM и связывание контекста выполнения с деревом DOM. То есть операции с узлами DOM выполняются с использованием данных в контексте выполнения, а события DOM вызывают методы, доступные в контексте.

## Встроенные директивы

В составе фреймворка AngularJS имеется множество встроенных директив. Среди них имеются директивы, реализующие элементы

и атрибуты HTML, которые имеют явно нестандартный вид, такие как `ng-include`, `ng-controller` и `ng-click`. Но имеются также директивы, неотличимые от стандартных элементов HTML, такие как `script`, `a`, `select` и `input`. Все эти директивы образуют ядро фреймворка AngularJS.

Самое замечательное, что встроенные директивы реализованы с применением того же прикладного интерфейса, который можно использовать и для создания собственных директив. Они ничем не отличаются от директив, которые может написать любой разработчик. Изучение исходных текстов с реализацией директив в AngularJS – это отличный способ научиться писать собственные директивы.

Реализацию встроенных директив AngularJS можно найти в папке src/ng/directive проекта AngularJS (https://github.com/angular/angular.js/tree/master/src/ng/directive/).

## Использование директив в разметке HTML

Директивы могут выглядеть как **элементы** и **атрибуты** HTML, как **комментарии** или **классы CSS**. Кроме того, любая директива может иметь множество **форматов использования**.

Ниже приводится несколько примеров использования директив в разметке HTML (имейте в виду, что формат использования директив зависит от их предназначения):

```
<my-directive></my-directive>
<input my-directive>
<!-- directive: my-directive-->
<input class="my-directive">
```

Обычно при определении и использовании директив в JavaScript используются имена в «верблюжьей» нотации, например, `myDirective`.

### Порядок компиляции директив

Когда фреймворк AngularJS выполняет компиляцию шаблонов с разметкой HTML, он производит обход дерева DOM, созданного браузером, и пытается сопоставить каждый элемент, атрибут, комментарий и класс CSS со своим списком зарегистрированных директив. Обнаружив директиву, AngularJS вызывает ее функцию компиляции, которая возвращает связывающую функцию. Фреймворк сохраняет эти связывающие функции.

Компиляция выполняется до того, как будет подготовлен объект контекста, поэтому в функции компиляции контекст недоступен.

Когда все директивы будут скомпилированы, AngularJS создает контекст и связывает каждую директиву с контекстом вызовом связывающих функций.

На этапе связывания производится подключение контекста к директиве, и связывающая функция получает возможность связать контекст с деревом DOM.

На этапе компиляции в основном выполняется оптимизация. Практически всю работу можно выполнить в функции связывания (кроме отдельных операций, таких как вызов функции включения). Представьте повторяющуюся директиву (внутри `ng-repeat`) – функция компиляции этой директивы вызывается только один раз, но ее функция связывания вызывается в каждой итерации, и каждый раз ей передаются разные данные.

Подробно функции включения рассматриваются в главе 9.

В табл. 8.1 показано, какие функции компиляции вызываются, когда AngularJS обнаруживает директивы. Как видно из этой таблицы, каждая функция компиляции вызывается только один раз, при каждом обращении к директиве в шаблоне.

**Таблица 8.1.** Порядок вызова функций компиляции

| Шаблон | Этап компиляции |
|---|---|
| `<ul my-dir>`<br>`  <li ng-repeat=”obj in objs” my-dir>`<br>`  </li>`<br>`</ul>` | функция компиляции директивы `myDir`<br>функция компиляции директивы `ngRepeat`<br>функция компиляции директивы `myDir` |

В табл. 8.2 показано, какие функции связывания вызываются, когда шаблон преобразуется в окончательную разметку HTML. Как видно из этой таблицы, функция связывания директивы `myDir` вызывается в каждой итерации, выполняемой директивой повторения.

**Таблица 8.2.** Порядок вызова функций связывания

| Шаблон | Этап связывания |
|---|---|
| `<ul my-dir>`<br>`<!-- ng-repeat="obj in objs" ->`<br>`  <li my-dir></li>`<br>`  <li my-dir></li>`<br>`  <li my-dir></li>`<br>`</ul>` | функция связывания директивы `myDir`<br>функция связывания директивы `ngRepeat`<br>функция связывания директивы `myDir`<br>функция связывания директивы `myDir`<br>функция связывания директивы `myDir` |

Если требуется выполнить какие-то дорогостоящие операции, не использующие данные в контексте, их следует вынести в функцию компиляции, чтобы они выполнялись только один раз.

# Тестирование директив

Директивы выполняют низкоуровневые операции с деревом DOM и могут иметь весьма сложную реализацию. Эти обстоятельства увеличивают вероятность появления ошибок при создании директив и усложняют их отладку. Поэтому так важно максимально полно охватить директивы тестированием.

Создание модульных тестов для директив может сначала показаться чрезвычайно сложной задачей, но AngularJS предоставляет несколько замечательных инструментов, назначение которых состоит в том, чтобы сделать эту работу как можно менее трудоемкой и дать вам возможность получать удовольствие от надежных и простых в сопровождении директив.

В AngularJS имеется исчерпывающий комплект тестов для встроенных директив. Они находятся в папке test/ng/directive проекта AngularJS (https://github.com/angular/angular.js/tree/master/test/ng/directive).

В общем случае стратегия тестирования директив выглядит следующим образом:

- загрузить модуль, содержащий директиву;
- скомпилировать строку разметки с директивой, чтобы получить функцию связывания;
- вызвать функцию связывания, чтобы связать ее с объектом `$rootScope`;
- проверить в испытуемом элементе наличие ожидаемых свойств.

Ниже представлена обобщенная заготовка модульного теста для директивы:

```
describe('myDir directive', function () {
    var element, scope;

    beforeEach(module('myDirModule'));

    beforeEach(inject(function ($compile, $rootScope) {
        var linkingFn = $compile('<my-dir></my-dir>');
        scope = $rootScope;
        element = linkingFn(scope);
    }));

    it('has some properties', function() {
        expect(element.someMethod()).toBe(XXX);
    });

    it('does something to the scope', function() {
        expect(scope.someField).toBe(XXX);
    });
    ...
});
```

Загрузите модуль с реализацией директивы, создайте элемент с директивой, используя для этого функцию `$compile` и объект контекста `$rootScope`. Сохраните ссылки на элемент и контекст, чтобы они были доступны всем последующим тестам.

В зависимости от тестируемых особенностей, в вызовах функции `it` может потребоваться компилировать вспомогательные элементы. В этом случае необходимо также сохранить ссылку на функцию `$compile`.

Наконец, проверьте – действует ли директива в соответствии с ожиданиями, вызывая функции jQuery/jqLite и изменяя данные в контексте.

Если директива использует службу `$watch`, `$observe` или `$q`, вам понадобится также запустить цикл обработки вызовом `$digest` и только потом проверять результаты. Например:

```
it("updates the scope via a $watch", function() {
    scope.someField = 'something';
    scope.$digest();
    expect(scope.someOtherField).toBe('something');
});
```

В оставшейся части главы мы продолжим знакомство с приемами создания собственных директив через их тестирование, следуя при-

нципам **разработки через тестирование** (Test Driven Development, TDD).

# Определение директивы

Каждая директива должна быть зарегистрирована в некотором **модуле**. Для этого нужно вызвать метод `directive()` **модуля** и передать ему каноническое имя директивы, а так же фабричную функцию, возвращающую определение директивы.

```
angular.module('app', []).directive('myDir', function() {
    return myDirectiveDefinition;
});
```

Фабричная функция может внедряться со службами, используемыми директивой.

Определение директивы – это объект, поля которого сообщают компилятору, что делает эта директива. Некоторые поля носят декларативный характер (например, поле `replace: true` сообщает компилятору, что он должен заменить оригинальный элемент содержимым шаблона). Некоторые – императивный (например, поле `link: function(...)` определяет функцию связывания для использования компилятором).

В табл. 8.3 перечислены все поля в объекте определения директивы, которые могут использоваться.

**Таблица 8.3.** Поля объекта определения директивы

| Поле | Описание |
|---|---|
| `name` | Имя директивы. |
| `restrict` | В разметке какого типа может появляться данная директива. |
| `priority` | Помогает компилятору определять порядок компиляции директив. |
| `terminal` | Определяет возможность компиляции других директив, следующих ниже этой директивы. |
| `link` | Функция связывания, которая свяжет директиву с контекстом. |
| `template` | Строка, которая будет использоваться этой директивой для создания разметки. |
| `templateUrl` | Адрес URL, где можно найти шаблон для данной директивы. |
| `replace` | Определяет необходимость замены оригинального элемента шаблоном этой директивы. |
| `transclude` | Определяет необходимость передачи содержимого элемента этой директивы для использования в шаблоне и функции компиляции. |

| Поле | Описание |
|---|---|
| `scope` | Определяет необходимость создания нового дочернего или изолированного контекста для этой директивы. |
| `controller` | Функция, которая будет действовать как контроллер этой директивы. |
| `require` | Определяет необходимость внедрения контроллера из другой директивы в функцию связывания этой директивы. |
| `compile` | Функция компиляции, где можно выполнить операции с исходным деревом DOM, и создающая функцию связывания, если она не была указана в поле `link`. |

Чаще всего при создании собственных директив вам понадобятся лишь некоторые из этих полей. В оставшейся части этой главы мы покажем вам различные директивы, написанные для приложения SCRUM. Для каждой директивы будут описываться соответствующие элементы их определения.

# Оформление кнопок с помощью директив

В нашем приложении мы используем стили Bootstrap CSS. Эта библиотека стилей содержит разметку и классы CSS для оформления кнопок. Например, кнопку можно создать разметкой:

```
<button type="submit"
        class="btn btn-primary btn-large">Click Me!</button>
```

Необходимость помнить все эти классы раздражает и требует времени. Чтобы избавиться от этого, можно создать директиву, которая упростит задачу.

## *Создание директивы button*

Все кнопки в нашем приложении должны быть оформлены с применением стилей из библиотеки Bootstrap CSS. Вместо того, чтобы добавлять в разметку всех кнопок атрибут `class="btn"`, мы можем реализовать директиву `button`, которая будет делать все это автоматически. Ниже приводится модульный тест для этой директивы:

```
describe('button directive', function () {
    var $compile, $rootScope;
    beforeEach(module('directives.button'));
    beforeEach(inject(function(_$compile_, _$rootScope_) {
```

```
        $compile = _$compile_;
        $rootScope = _$rootScope_;
    }));

    it('adds a "btn" class to the button element', function() {
        var element = $compile('<button></button>')($rootScope);
        expect(element.hasClass('btn')).toBe(true);
    });
});
```

Он загружает модуль, создает кнопку и проверяет наличие в ней ожидаемого класса CSS.

Имейте в виду, что механизм внедрения зависимостей игнорирует парные символы подчеркивания (как, например, в идентификаторе `_$compile_`), окружающие имя параметра. Это позволяет копировать внедряемые службы в переменные с правильными именами для последующего использования (например, `$compile = _$compile_`).

Далее, любая кнопка с атрибутом `type="submit"` должна автоматически оформляться как кнопка по умолчанию, а кроме того, было бы неплохо иметь возможность определять размеры кнопок через атрибут `size`. Одним словом, нам нужна возможность создавать кнопки, как показано ниже:

```
<button type="submit" size="large">Submit</button>
```

Для проверки дополнительных особенностей можно добавить следующие модульные тесты:

```
it('adds size classes correctly', function() {
    var element = $compile('<button size="large"></button>')
    ($rootScope);
    expect(element.hasClass('btn-large')).toBe(true);
});

it('adds primary class to submit buttons', function() {
    var element = $compile('<button type="submit"></button>')
    ($rootScope);
    expect(element.hasClass('btn-primary')).toBe(true);
});
```

Теперь посмотрим, как можно реализовать директиву:

```
myModule.directive('button', function() {
   return {
      restrict: 'E',
      compile: function(element, attributes) {
```

```
            element.addClass('btn');
            if ( attributes.type === 'submit' ) {
               element.addClass('btn-primary');
            }
            if ( attributes.size ) {
               element.addClass('btn-' + attributes.size);
            }
         }
      };
   });
```

Мы исходим из предположения, что модуль `myModule` уже определен.

Мы дали нашей директиве имя `'button'` и ограничили круг ее применения, указав, что она может использоваться только как элемент (`restrict: 'E'`). Это означает, что данная директива будет применяться везде, где компилятор AngularJS обнаружит элемент `button`. Фактически, этой директивой мы расширили поведение стандартного HTML-элемента `button`.

Кроме этого в определении директивы присутствует только функция `compile`. Данная функция будет вызываться всякий раз, когда компилятор обнаружит совпадение имени элемента с именем директивы. Функции `compile` будет передан параметр с именем `element`. Это – объект jQuery (или jqLite), ссылающийся на элемент DOM, где встречена директива, в данном случае – сам элемент `button`.

В функции компиляции мы просто добавляем классы CSS в элемент, исходя из значений атрибутов элемента. Доступ к значениям атрибутов элемента осуществляется через внедренный параметр `attributes`.

Все эти изменения можно выполнить в функции компиляции, а не в функции связывания только потому, что при этом не требуются данные из контекста, который будет связан с элементом. Эти операции можно было бы также выполнить и в функции связывания, но если бы директива оказалась внутри цикла `ng-repeat`, тогда метод `addClass()` вызывался бы в каждой итерации.

Подробнее об этом рассказывается в разделе «Порядок компиляции директив».

Функция `compile` вызывается только один раз, а директива `ng-repeat` просто будет создавать копии созданного ею шаблона.

Если при создании элемента потребуется выполнить дорогостоящие операции с деревом DOM, тогда эта оптимизация позволит сэкономит массу времени, особенно при выполнении итераций через большие коллекции.

# Директивы-виджеты

Одной из самых сильных сторон директив является возможность создавать собственные теги. Иными словами, вы можете создавать собственные элементы и атрибуты, придающие новый смысл и поведение разметке HTML, характерные для предметной области, в которой будет использоваться ваше приложение.

Например, можно создать элемент `<user>`, отображающий информацию о пользователе, или элемент `<g-map>`, обеспечивающий взаимодействие со службой Google Map. Возможности бесконечны и самое большое преимущество в том, что разметка будет соответствовать предметной области приложения.

## *Создание директивы постраничного просмотра*

В нашем приложении SCRUM нам часто придется иметь дело с длинными списками заданий, не умещающимися на одном экране. Мы решили организовать постраничный просмотр таких списков. Обычно в таких ситуациях на экран выводится блок выбора со ссылками на отдельные страницы в списке.

Фреймворк Bootstrap CSS предоставляет отличный набор стилей для оформления таких виджетов (рис. 8.2).


**Рис. 8.2.** Виджет выбора страницы в списке

Мы реализуем такой блок выбора страницы в виде директивы-виджета, чтобы ее можно было использовать в разметке, не задумываясь о том, как она действует. В разметке эта директива будет иметь следующий вид:

```
<pagination num-pages="tasks.pageCount"
    current-page="tasks.currentPage">
</pagination>
```

## Тест для директивы постраничного просмотра списков

Тесты для этого виджета должны проверять все изменения, которые могут выполняться и в функции `$scope` и возникать в результате щелчка мышью на ссылках. Ниже выборочно приводится реализация наиболее важных тестов:

```
describe('pagination directive', function () {
    var $scope, element, lis;
    beforeEach(module('directives'));
    beforeEach(inject(function($compile, $rootScope) {
        $scope = $rootScope;
        $scope.numPages = 5;
        $scope.currentPage = 3;
        element = $compile('<pagination num-pages="numPages"
                  currentpage="currentPage"></pagination>')($scope);
        $scope.$digest();
        lis = function() { return element.find('li'); };
    }));

    it('has the number of the page as text in each page item',
       function() {
            for(var i=1; i<=$scope.numPages;i++) {
                expect(lis().eq(i).text()).toEqual(''+i);
       }
    });

    it('sets the current-page to be active', function() {
        var currentPageItem = lis().eq($scope.currentPage);
        expect(currentPageItem.hasClass('active')).toBe(true);
    });

    ...

    it('disables "next" if current-page is num-pages', function() {
        $scope.currentPage = 5;
        $scope.$digest();
        var nextPageItem = lis().eq(-1);
        expect(nextPageItem.hasClass('disabled')).toBe(true);
    });

    it('changes currentPage if a page link is clicked', function() {
        var page2 = lis().eq(2).find('a').eq(0);
        page2.click();
        $scope.$digest();
        expect($scope.currentPage).toBe(2);
    });

    ...

    it('does not change the current page on "next" click if already at
```

```
last page', function() {
        var next = lis().eq(-1).find('a');
        $scope.currentPage = 5;
        $scope.$digest();
        next.click();
        $scope.$digest();
        expect($scope.currentPage).toBe(5);
    });

    it('changes the number of items when numPages changes',
function() {
        $scope.numPages = 8;
        $scope.$digest();
        expect(lis().length).toBe(10);
        expect(lis().eq(0).text()).toBe('Previous');
        expect(lis().eq(-1).text()).toBe('Next');
    });
});
```

## Использование шаблонов с разметкой HTML в директивах

Для этого виджета требуется сгенерировать некоторую разметку HTML, которая заменит директиву. Проще всего это реализовать с помощью шаблона. Ниже приводится содержимое шаблона разметки для данной директивы:

```
<div class="pagination"><ul>
  <li ng-class="{disabled: noPrevious()}">
    <a ng-click="selectPrevious()">Previous</a>
  </li>
  <li ng-repeat="page in pages"
      ng-class="{active: isActive(page)}">
    <a ng-click="selectPage(page)">{{page}}</a>
  </li>
  <li ng-class="{disabled: noNext()}">
    <a ng-click="selectNext()">Next</a>
  </li>
</ul></div>
```

В шаблоне используется массив с именем `pages` и несколько вспомогательных функций, таких как `selectPage()` и `noNext()`. Эти функции являются внутренней реализацией виджета. Их требуется поместить в объект контекста, чтобы обеспечить их доступность внутри шаблона, но они не должны быть доступны в контексте, где используется сам виджет. С этой целью мы потребовали от компилятора создать новый, изолированный контекст для шаблона.

## Изолирование директивы от родительского контекста

В данный момент мы ничего не знаем о контексте, в каком будет использоваться директива. Поэтому считается хорошей практикой экспортировать из директивы только четко определенный общедоступный интерфейс. Это гарантирует отсутствие конфликтов имен с родительским контекстом.

У нас на выбор имеется три варианта использования контекста в директиве и ее шаблоне, которые определяются с помощью свойства `scope` в объекте определения директивы.

- Использовать контекст, в котором находится виджет. Этот вариант применяется по умолчанию и ему соответствует значение `scope: false`.
- Создать дочерний контекст, наследующий прототип контекста, в котором находится виджет. Этот вариант выбирается значением `scope: true`.
- Создать изолированный контекст, не наследующий родительский, благодаря чему он оказывается полностью изолированным от родительского контекста. Этот вариант выбирается присваиванием свойству `scope` объекта: `scope: { ... }`.

Нам требуется полностью отделить шаблон разметки виджета от остального приложения, чтобы предотвратить опасную утечку данных между ними. Мы будем использовать изолированный контекст, который действует, как показано на рис. 8.3.

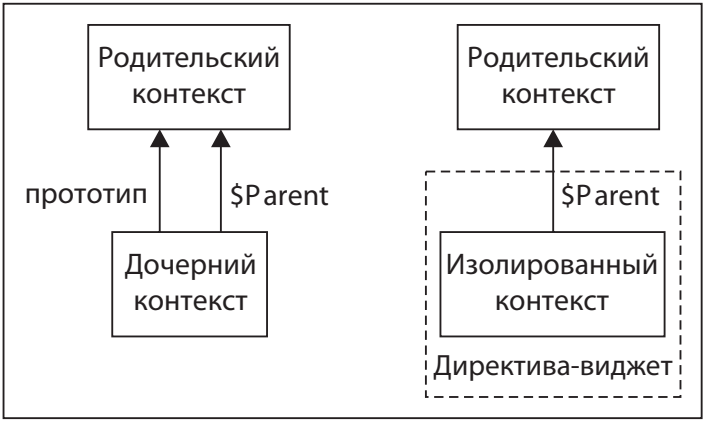


**Рис. 8.3.** Изолированный контекст директивы

Несмотря на то, что изолированный контекст не наследует прототип родителя, он все еще имеет доступ к родительскому контексту через свойство `$parent`. Но использовать эту возможность считается плохой практикой, так как нарушает изоляцию директивы от окружающей ее среды.

Так как теперь наш контекст изолирован от родительского контекста, необходимо организовать явную передачу значений между родительским и изолированным контекстами. Это достигается с помощью выражений AngularJS, ссылающихся на атрибуты элемента, где, где находится директива. В директиве постраничного просмотра списков эту роль играют атрибуты `num-pages` и `current-page`.

Синхронизировать выражения в этих атрибутах со свойствами в контексте шаблона можно посредством функций-наблюдателей за контекстом (watches). Функции-наблюдатели можно создавать вручную или предложить фреймворку AngularJS сделать это за нас. Существует три типа интерфейсов между атрибутами элементов и изолированным контекстом: интерполяция (`@`), связывание данных (`=`) и выражения (`&`). Эти интерфейсы определяются как свойства объекта контекста в виде пар ключ/значение.

Ключ определяет имя поля изолированного контекста, а значение определяет имя атрибута элемента с префиксом `@`, `=` или `&`:

```
scope: {
    isolated1: '@attribute1',
    isolated2: '=attribute2',
    isolated3: '&attribute3'
}
```

В этом примере определяется три поля изолированного контекста, которые будут отображаться фреймворком AngularJS на указанные атрибуты элемента, в котором находится директива.

Если имя атрибута отсутствует в описании значения, предполагается, что он имеет то же имя, что и поле изолированного контекста:

```
scope: { isolated1: '@' }
```

Здесь предполагается, что атрибут имеет имя `isolated1`.

## Интерполяция атрибутов с префиксом @

Префикс `@` сообщает фреймворку, что он должен интерполировать значение указанного атрибута и обновлять свойство изолированного контекста при изменении этого атрибута. Интерполяция выполняет-

ся с помощью фигурных скобок {{}}, результатом которой является строка, включающая значения из родительского контекста.

Программисты часто допускают ошибку, ожидая, что в результате интерполяции объекта получится сам объект. Результатом интерполяции всегда является строка. Например, если имеется объект `user` с полем `userName`, результатом интерполяции `{{user}}` будет объект `user` в форме строки, и вы не сможете обратиться к свойству `userName` этой строки.

Интерфейс интерполяции атрибутов сродни явному использованию функции-наблюдателя за атрибутами `$observe`:

```
attrs.$observe('attribute1', function(value) {
    isolatedScope.isolated1 = value;
});
attrs.$$observers['attribute1'].$$scope = parentScope;
```

### Связывание данных в атрибутах с префиксом =

Префикс = сообщает фреймворку, что он должен синхронизировать выражение в атрибуте со значением в изолированном контексте. Это связывание является двунаправленным, что позволяет непосредственно отображать объекты и значения между внешними и внутренним виджетами.

Так как этот интерфейс поддерживает двунаправленное связывание данных, выражение в атрибуте должно допускать возможность присваивания (то есть, ссылаться на поле контекста или другого объекта) – оно не может быть произвольным вычисляемым выражением.

Интерфейс связывания значений сродни явному использованию двух функций-наблюдателей за контекстом `$watch`:

```
var parentGet = $parse(attrs['attribute2']);
var parentSet = parentGet.assign;
parentScope.$watch(parentGet, function(value) {
    isolatedScope.isolated2 = value;
});
isolatedScope.$watch('isolated2', function(value) {
    parentSet(parentScope, value);
});
```

Фактическая реализация в действительности намного сложнее, так как должна гарантировать стабильность передачи данных между двумя контекстами.

### Определение функций обратного вызова в атрибутах с префиксом &

Префикс & сообщает, что выражение в атрибуте элемента будет доступно в контексте в виде функции, вызов которой будет инициировать вычисление выражения. Эту особенность удобно использовать для создания функций обратного вызова для виджетов.

Данный интерфейс сродни вызову функции `$parse` для обработки выражения в атрибуте и передаче обратно функции, получившейся в изолированном контексте:

```
parentGet = $parse(attrs['attribute3']);
scope.isolated3 = function(locals) {
    return parentGet(parentScope, locals);
};
```

## *Реализация виджета*

Ниже приводится объект определения директивы `pagination`:

```
myModule.directive('pagination', function() {
return {
    restrict: 'E',
    scope: {
        numPages: '=',
        currentPage: '='
    },
    template: ...,
    replace: true,
```

Данная директива может применяться только как элемент. Она создает изолированный контекст с данными `numPages` и `currentPage`, связанными с атрибутами `num-pages` и `current-page`, соответственно. Элементы директивы будут замещаться шаблоном, представленным выше:

```
    link: function(scope) {
        scope.$watch('numPages', function(value) {
            scope.pages = [];
             for(var i=1;i<=value;i++) { scope.pages.push(i); }
             if ( scope.currentPage > value ) {
                 scope.selectPage(value);
             }
         });

         ...

         scope.isActive = function(page) {
```

```
            return scope.currentPage === page;
        };

        scope.selectPage = function(page) {
            if ( ! scope.isActive(page) ) {
                scope.currentPage = page;
            }
        };

        ...

        scope.selectNext = function() {
            if ( !scope.noNext() ) {
                scope.selectPage(scope.currentPage+1);
            }
        };
    }
```

Функция `link` реализует в свойстве `$watch` создание массива страниц, исходя из значения свойства `numPages`. Она также добавляет в изолированный контекст несколько вспомогательных функций, которые будут использоваться в шаблоне директивы.

## Добавление в директиву функции обратного вызова selectPage

Было бы удобно иметь функцию или выражение, которые вызывались бы при изменении номера текущей страницы. Добиться этого можно, определив в директиве новый атрибут и отобразив его в изолированный контекст с использованием префикса `&`.

```
<pagination
    num-pages="tasks.pageCount"
    current-page="tasks.currentPage"
    on-select-page="selectPage(page)">
</pagination>
```

Здесь сообщается, что при выборе некоторой страницы директива должна вызвать функцию `selectPage(page)` и передать ей новый номер страницы. Далее приводится реализация теста для этой особенности:

```
    it('executes the onSelectPage expression when the current
page changes',
       inject(function($compile, $rootScope) {
        $rootScope.selectPageHandler =
            jasmine.createSpy('selectPageHandler');
```

```
        element = $compile(
            '<pagination num-pages="numPages" ' +
            ' current-page="currentPage" ' +
            ' on-select-page="selectPageHandler(page)">' +
            '</pagination>')($rootScope);
        $rootScope.$digest();
        var page2 = element.find('li').eq(2).find('a').eq(0);
        page2.click();
        $rootScope.$digest();
        expect($rootScope.selectPageHandler).toHaveBeenCalledWith(2);
    }));
```

Сначала функция `it` создает эмулятор для обработки обратного вызова, а затем имитирует выбор новой страницы щелчком мыши.

Для реализации этой возможности необходимо добавить в определение изолированного контекста еще одно поле:

```
scope: {
    ...,
    onSelectPage: '&'
},
```

Теперь функция `onSelectPage()` будет доступна в изолированном контексте. При ее вызове, она выполнит выражение, переданное в атрибуте `on-select-page`. Далее нам нужно изменить функцию `selectPage()` в изолированном контексте так, чтобы она вызывала `onSelectPage()`:

```
scope.selectPage = function(page) {
    if ( ! scope.isActive(page) ) {
        scope.currentPage = page;
        scope.onSelectPage({ page: page });
    }
};
```

Обратите внимание, что переменная `page` передается в выражение в виде хеша переменных. Эти переменные будут переданы в связанное выражение в момент его выполнения, как если бы они присутствовали в его контексте.

# Создание директивы проверки

В нашем приложении SCRUM имеется форма редактирования информации о пользователе. В этой форме мы требуем, чтобы пользователь указал пароль. Так как символы в поле ввода пароля замещаются

звездочками и пользователь не может видеть, что он вводит, было бы полезно иметь поле подтверждения пароля.

Нам необходимо убедиться в совпадении содержимого полей ввода пароля и подтверждения. Для этого мы создадим собственную директиву проверки, которую будем применять к элементу ввода, содержимое которого требуется проверить на совпадение с содержимым другого элемента ввода. В разметке это будет выглядеть примерно так:

```
<form name="passwordForm">
  <input type="password" name="password" ng-model="user.password">
  <input type="password" name="confirmPassword"
         ng-model="confirmPassword" validate-equals="user.password">
</form>
```

Эта директива проверки должна интегрироваться с контроллером `ngModelController`, чтобы обеспечить обратную связь с пользователем.

Мы можем экспортировать `ngModelController` в контекст, указав имя формы и имя элемента ввода. Это позволит обращаться к функции проверки модели из контроллера. Наша директива проверки будет устанавливать признак допустимости для поля ввода `confirmPassword`, если его значение совпадает со значением модели `user.password`.

## Внедрение контроллера другой директивы

Директивы проверки требуют наличия доступа к контроллеру `ngModelController` директивы `ng-model`. Реализовать внедрение этого контроллера можно с помощью поля `require` объекта определения директивы. Это поле принимает строку или массив строк. Каждая строка должна содержать каноническое имя директивы, где определен внедряемый контроллер.

Когда компилятор обнаружит требуемую директиву, ее контроллер будет внедрен в функцию связывания в четвертый параметр. Например:

```
require: 'ngModel',
link: function(scope, element, attrs, ngModelController) { ... }
```

Если потребуется внедрить несколько контроллеров, в четвертом параметре будет передан массив с контроллерами, следующими в том же порядке, как указано в поле `require`.

Если текущий элемент не содержит указанную директиву, компилятор сообщит об ошибке. Это отличный способ гарантировать обязательное использование другой директивы.

### Необязательное требование к наличию контроллера

Требование наличия контроллера, указанного в поле `require`, можно сделать необязательным, добавив префикс `'?'` перед именем директивы, например, `require: '?ngModel'`. Если требуемая директива не будет указана в элементе, в четвертом параметре функция связывания получит `null`. Если в поле `require` указывается более одного контроллера, значение `null` получат соответствующие элементы массива.

### Поиск родительского контроллера

Если директива, определяющая внедряемый контроллер, может находиться в родительских элементах, перед именем директивы в поле `require` следует поместить префикс `'^'`, например, `require: '?ngModel'`. В этом случае компилятор произведет поиск требуемой директивы в родительских элементах, снизу вверх, и вернет первый найденный контроллер.

Чтобы определить необязательный контроллер, реализованный в родительском элементе, допускается объединять префиксы `'?'` и `'^'`. Например, определение `require: '^?form'` позволит найти контроллер директивы `form`, как это делает директива `ng-model`, чтобы зарегистрировать себя в форме.

## *Взаимодействие с контроллером ngModelController*

После внедрения контроллера `ngModelController` мы получаем возможность использовать его для проверки содержимого элементов ввода. Это – типичная и достаточно простая ситуация для директив подобного рода. В табл. 8.4 перечислены функции и свойства, экспортируемые контроллером `ngModelController`.

**Таблица 8.4.** Функции и свойства, экспортируемые контроллером `ngModelController`

| Имя | Описание |
|---|---|
| `$parsers` | Конвейер из функций, которые будут вызываться поочередно при изменении значения в элементе ввода. |
| `$formatters` | Конвейер из функций, которые будут вызываться поочередно при изменении значения в модели. |
| `$setValidity(validationErrorKey, isValid)` | Функция, устанавливающая признак допустимости для данного типа ошибок. |
| `$valid` | Содержит истинное значение, если ошибки не обнаружены. |
| `$error` | Хранит информацию обо всех ошибках, обнаруженных в модели. |

Функции, указанные в полях `$parsers` и `$formatters`, принимают и возвращают значение, например, `function(value) { return value; }`. Каждая функция получит значение, которая вернет предыдущая функция в конвейере. Именно в эти функции помещается логика проверки и вызов `$setValidity()`.

## Тестирование директивы проверки

Суть тестирования директив проверки заключается в компиляции формы, содержащей поле ввода с директивой `ng-model` и директивой проверки. Например:

```
<form name="testForm">
  <input name="testInput"
         ng-model="model.testValue"
         validate-equals="model.compareTo">
</form>
```

В данном случае директива используется в качестве атрибута элемента ввода. Значением атрибута является выражение, возвращающее значение в модели. Наша директива будет сравнивать указанное значение со значением поля ввода.

С помощью директивы `ng-model` мы связали модель с полем ввода. В результате будет создан контролер `ngModelController`, экспортируемый в контекст как `$scope.testForm.testInput`, а значение модели будет экспортироваться в контекст как `$scope.model.testValue`.

Далее изменяются значения поля ввода и модели, и проверяется, изменились ли значения `$valid` и `$error` в контроллере `ngModelController`.

При подготовке теста мы сохраняем ссылки на модель и контроллер `ngModelController`.

```
describe('validateEquals directive', function() {
    var $scope, modelCtrl, modelValue;

    beforeEach(inject(function($compile, $rootScope) {
        ...
        modelValue = $scope.model = {};
        modelCtrl = $scope.testForm.testInput;
        ...
    }));

    ...

    describe('model value changes', function() {
        it('should be invalid if the model changes', function() {
            modelValue.testValue = 'different';
            $scope.$digest();
            expect(modelCtrl.$valid).toBeFalsy();
            expect(modelCtrl.$viewValue).toBe(undefined);
        });
        it('should be invalid if the reference model changes',
function() {
            modelValue.compareTo = 'different';
            $scope.$digest();
            expect(modelCtrl.$valid).toBeFalsy();
            expect(modelCtrl.$viewValue).toBe(undefined);
        });
        it('should be valid if the modelValue changes to be
the same as the reference', function() {
            modelValue.compareTo = 'different';
            $scope.$digest();
            expect(modelCtrl.$valid).toBeFalsy();

            modelValue.testValue = 'different';
            $scope.$digest();
            expect(modelCtrl.$valid).toBeTruthy();
            expect(modelCtrl.$viewValue).toBe('different');
        });
    });
```

Здесь изменяется контекст, значение модели поля ввода (`modelValue.testValue`) и значение модели поля ввода, с которым должно выполняться сравнение (`modelValue.compareTo`). Далее проверяется допустимость введенного значения (`modelCtrl`). Функ-

ция `$digest()` вызывается, чтобы обеспечить обновление значения поля ввода в соответствии со значением модели.

```
    describe('input value changes', function() {
        it('should be invalid if the input value changes',
function() {
            modelCtrl.$setViewValue('different');
            expect(modelCtrl.$valid).toBeFalsy();
            expect(modelValue.testValue).toBe(undefined);
        });
        it('should be valid if the input value changes to be
the same as the reference', function() {
            modelValue.compareTo = 'different';
            $scope.$digest();
            expect(modelCtrl.$valid).toBeFalsy();
            modelCtrl.$setViewValue('different');
            expect(modelCtrl.$viewValue).toBe('different');
            expect(modelCtrl.$valid).toBeTruthy();
        });
    });
});
```

Здесь, вызовом `$setViewValue()`, имитируется изменение поля ввода пользователем.

## Реализация директивы проверки

Теперь, после создания теста, можно приступить к реализации самой директивы:

```
myModule.directive('validateEquals', function() {
    return {
        require: 'ngModel',
        link: function(scope, elm, attrs, ngModelCtrl) {
            function validateEqual(myValue) {
                var valid = (myValue === scope.$eval(attrs.
validateEquals));
                ngModelCtrl.$setValidity('equal', valid);
                return valid ? myValue : undefined;
            }

            ngModelCtrl.$parsers.push(validateEqual);
            ngModelCtrl.$formatters.push(validateEqual);

            scope.$watch(attrs.validateEquals, function() {
                ngModelCtrl.$setViewValue(ngModelCtrl.$viewValue);
            });
        }
    };
});
```

Мы создали функцию `validateEqual(value)`, сравнивающую переданное ей значение `value` со значением выражения, и добавили ее в конвейеры `$parsers` и `$formatters`, чтобы она вызывалась при каждом изменении модели или представления.

В этой директиве необходимо также учитывать возможность изменения модели, с которой выполняется сравнение. Для этого мы организовали наблюдение за выражением, которое извлекается из параметра `attrs` функции связывания. Обнаружив изменение, мы искусственно запускаем конвейер `$parsers` вызовом `$setViewValue()`. Это гарантирует вызов всех функций в конвейере `$parsers`, если какая-то изменит значение модели до того, как она попадет в нашу реализацию проверки.

# Асинхронная проверка модели

Иногда бывает необходимо проверить результаты взаимодействий с удаленной службой, например, с базой данных. В таких ситуациях ответ от службы попадает в приложение асинхронно. Это привносит дополнительные проблемы не только с точки зрения проверки модели, но и тестирования таких взаимодействий.

В форме редактирования информации о пользователе было бы желательно проверить, не был ли раньше зарегистрирован пользователь, имеющий такой же адрес электронной почты. Для этого мы создадим директиву `uniqueEmail`, обращающуюся к серверу для проверки присутствия введенного адреса электронной почты в базе данных:

```
<input ng-model="user.email" unique-email>
```

## *Имитация службы Users*

Для поиска адреса электронной почты в базе данных приложение использует уже имеющуюся службу `Users`. Поэтому для нужд тестирования нам требуется создать фиктивную службу `Users` с методом `query()`.

В данном случае проще создать тестовый модуль и определить в нем фиктивный объект службы `Users`, чем внедрять действующую службу и пытаться следить за ее методом `query()`, потому что сама служба `Users` опирается на множество других служб и констант.

```
angular.module('mock.Users', []).factory('Users', function() {
    var Users = { };
    Users.query = function(query, response) {
        Users.respondWith = function(emails) {
            response(emails);
            Users.respondWith = undefined;
        };
    };
    return Users;
});
```

Функция `query()` создает метод `Users.respondWith()`, вызывающий функцию обратного вызова `response`, переданную в вызов `query()`. Это дает возможность имитировать в тестах получение ответа на запрос.

Перед отправкой запроса вызовом `Users.query()` и после его обработки вызовом `Users.respondWith()` мы присваиваем свойству `Users.respondWith` значение `undefined`.

Затем необходимо загрузить этот модуль вместе с тестируемым модулем:

```
beforeEach(module('mock.users'));
```

В результате произойдет подмена оригинальной службы `Users` ее имитацией.

## Тестирование директивы асинхронной проверки

Реализация теста этой директивы напоминает тест предыдущей директивы проверки:

```
beforeEach(inject(function($compile, $rootScope, _Users_){
    Users = _Users_;
    spyOn(Users, 'query').andCallThrough();
    ...
}));
```

Мы устанавливаем наблюдение за функцией `Users.query()`, но при этом гарантируем использование фиктивной ее версии, чтобы обеспечить имитацию возврата ответа вызовом `Users.respondWith()`.

Ниже приводятся наиболее важные модульные тесты:

```
it('should call Users.query when the view changes', function() {
    testInput.$setViewValue('different');
    expect(Users.query).toHaveBeenCalled();
});

it('should set model to invalid if the Users.query response
contains users', function() {
    testInput.$setViewValue('different');
    Users.respondWith(['someUser']);
    expect(testInput.$valid).toBe(false);
});

it('should set model to valid if the Users.query response
contains no users', function() {
    testInput.$setViewValue('different');
    Users.respondWith([]);
    expect(testInput.$valid).toBe(true);
});
```

Здесь проверяется – был ли вызван метод `Users.query()`. Кроме того, поскольку `Users.query()` использует функцию обратного вызова для возврата ответа, мы можем имитировать ответ сервера с помощью `Users.respondWith()`.

Нам также требуется убедиться, что приложение не выполнит повторное обращение к серверу, если пользователь просто повторно введет адрес электронной почты, уже хранящийся в модели. Например, при редактировании информации о существующем пользователе его адрес электронной почты уже имеется в базе данных на сервере, и этот адрес является допустимым.

```
it('should not call Users.query if the view changes to be the
same as the original model', function() {
    $scope.model.testValue = 'admin@abc.com';
    $scope.$digest();
    testInput.$setViewValue('admin@abc.com');
    expect(Users.query).not.toHaveBeenCalled();
    testInput.$setViewValue('other@abc.com');
    expect(Users.query).toHaveBeenCalled();

    querySpy.reset();
    testInput.$setViewValue('admin@abc.com');
    expect(Users.query).not.toHaveBeenCalled();
    $scope.model.testValue = 'other@abc.com';
    $scope.$digest();
    testInput.$setViewValue('admin@abc.com');
    expect(Users.query).toHaveBeenCalled();
});
```

Здесь в модель записывается адрес электронной почты, а затем проверяется факт вызова метода `Users.query()`, который должен вызываться, только если значение поля ввода отличается от оригинального значения в модели. Чтобы убедиться, что метод, находящийся под наблюдением, не использовался с момента последней проверки, мы вызываем `Users.query.reset()`.

## Реализация директивы асинхронной проверки

Реализация этой директивы по своей структуре напоминает реализацию предыдущей директивы проверки. Мы требуем внедрение контроллера директивы `ngModel` и добавляем свои функции в его конвейеры `$parsers` и `$formatters` внутри функции связывания:

```
myModule.directive('uniqueEmail', ["Users", function (Users) {
    return {
        require:'ngModel',
        link:function (scope, element, attrs, ngModelCtrl) {
            var original;
            ngModelCtrl.$formatters.unshift(function(modelValue) {
                original = modelValue;
                return modelValue;
            });

            ngModelCtrl.$parsers.push(function (viewValue) {
                if (viewValue && viewValue !== original ) {
                    Users.query({email:viewValue}, function (users) {
                        if (users.length === 0) {
                            ngModelCtrl.$setValidity('uniqueEmail',
true);
                        } else {
                            ngModelCtrl.$setValidity('uniqueEmail',
false);
                        }
                    });
                    return viewValue;
                }
            });
        }
    };
}]);
```

Обращение к серверу внутри функции, помещаемой в конвейер `$parser`, выполняется, только если пользователь изменил адрес электронной почты в поле ввода. Если значение изменяется программно,

посредством модели, предполагается, что действительность адреса гарантируется логикой приложения. Например, если приложение загружает информацию о существующем пользователе для последующего редактирования, адрес электронной почты является действительным, даже при том, что он присутствует в базе данных на сервере и может расцениваться как уже занятый.

Обычно функция, выполняющая проверку, возвращает `undefined`, если проверяемое значение оказывается недопустимым. Это препятствует сохранению недопустимого значения в модели. В данном случае, к моменту возврата из функции, еще не известно, является ли значение допустимым или недопустимым. Поэтому мы всегда возвращаем само значение, а установку признака допустимости оставляем за функцией обработки ответа.

В конвейер `$formatters` добавляется функция, сохраняющая значение модели в переменной `original`. С ее помощью устраняется необходимость обращения к серверу из функции проверки, если пользователь повторно ввел оригинальный адрес электронной почты, который в противном случае мог бы быть признан недействительным.

# Директива-обертка для виджета выбора даты из библиотеки jQueryUI

Иногда в веб-приложениях используются виджеты сторонних разработчиков, достаточно сложные, чтобы сделать неоправданной разработку версии исключительно на основе AngularJS. Вы можете повысить скорость разработки, обертывая такие виджеты директивами AngularJS, но при этом вам придется особое внимание уделить взаимодействию двух библиотек.

В этом разделе мы создадим директиву `datepicker`, обертывающую виджет `datepicker` из библиотеки jQueryUI. В табл. 8.5 перечислены функции, экспортируемые виджетом и используемые для интеграции с AngularJS, где `element` – это обертка jQuery вокруг элемента, к которому подключается виджет.

**Таблица 8.5.** Функции, экспортируемые виджетом datepicker

| Функция | Описание |
|---|---|
| `element.datepicker(options)` | Создает новый виджет на основе параметров `options` и подключает его к элементу `element`. |
| `element.datepicker("setDate", date)` | Устанавливает дату в виджете. |
| `element.datepicker("getDate")` | Возвращает дату из виджета. |
| `element.datepicker("destroy")` | Уничтожает виджет и удаляет его из элемента `element`. |

Нам необходимо, чтобы виджет информировал приложение, когда пользователь выберет новую дату в нем. С этой целью в аргументе `options`, на основе которого создается новый виджет, можно указать функцию обратного вызова `onSelect`:

```
element.datepicker({onSelect: function(value, picker) { ... });
```

Чтобы упростить реализацию мы решили, что директива `datepicker` может быть связана только с JavaScript-объектом `Date` в модели.

В обобщенном виде (рис. 8.4) шаблон обертывания виджетов ввода из библиотеки jQuery напоминает директивы проверки данных. Здесь также можно внедрить контроллер директивы `ngModel` и добавить свои функции в конвейеры `$parsers` и `$formatters` для преобразования значений при передаче между моделью и представлением.

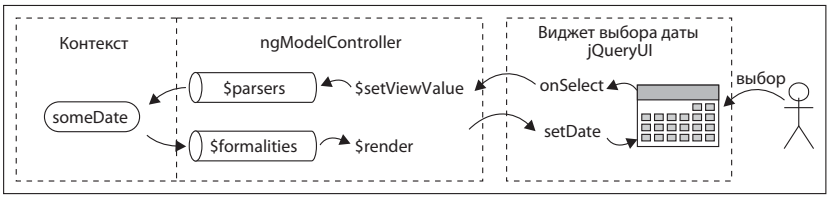


**Рис. 8.4.** Структура директивы-обертки вокруг виджета `datepicker` из библиотеки jQueryUI

Кроме того, нам потребуется вводить данные в виджет при изменении модели и сохранять их в модели, при изменении виджета. Мы переопределим метод `ngModel.$render()` обновления виджета. Эта функция вызывается после успешного выполнения всех функций в

конвейере `$formatters`. Для вывода данных будет использоваться функция обратного вызова `onSelect`, в которой будет вызываться метод `ngModel.$setViewValue()`, обновляющий значение представления и запускающий конвейер `$parsers`.

## Тестирование директив-оберток

В чистом модульном тесте можно было бы создать фиктивный виджет выбора даты, имеющий такой же прикладной интерфейс, как и виджет `datepicker` из библиотеки jQueryUI. Однако в данном случае мы решили пойти более практичным путем и использовать в тестах настоящий виджет выбора даты.

Преимущество такого подхода в том, что нам не придется скрупулезно определять интерфейс виджета, описанный в документации. Вызывая фактические методы и проверяя пользовательский интерфейс, можно с уверенностью сказать, правильно ли работает директива. Недостаток заключается в необходимости выполнения операций с деревом DOM, которые могут замедлять тестирование, и средств проверки корректности поведения пользовательского интерфейса.

В данном случае виджет `datepicker` из библиотеки jQueryUI экспортирует специализированную функцию, позволяющую имитировать выбор даты пользователем:

```
$.datepicker._selectDate(element);
```

Мы создадим вспомогательную функцию и будем использовать ее для имитации выбора даты в виджете:

```
var selectDate = function(element, date) {
    element.datepicker('setDate', date);
    $.datepicker._selectDate(element);
};
```

Порой бывает сложно симулировать подобные действия, и тогда следует подумать об имитации всего виджета в целом.

Сами тесты используют прикладной интерфейс виджета и эту вспомогательную функцию. Например:

```
describe('simple use on input element', function() {
    var aDate, element;
    beforeEach(function() {
        aDate = new Date(2010, 12, 1);
```

```
        element = $compile(
            "<input date-picker ng-model='x'/>")($rootScope);
    });
    it('should get the date from the model', function() {
        $rootScope.x = aDate;
        $rootScope.$digest();
        expect(element.datepicker('getDate')).toEqual(aDate);
    });
    it('should put the date in the model', function() {
        $rootScope.$digest();
        selectDate(element, aDate);
        expect($rootScope.x).toEqual(aDate);
    });
});
```

Здесь проверяется передача изменений из модели в виджет и из виджета в модель. Обратите внимание на отсутствие вызова `$digest()` после вызова `selectDate()`, поскольку в данном случае директива сама должна гарантировать выполнение цикла обработки после взаимодействия с пользователем.

Существует еще целое множество тестов для этой директивы, которые можно найти в в исходных текстах примера приложения.

## Реализация директивы datepicker

В реализации этой директивы так же используется функциональность контроллера `ngModelController`. В частности, в конвейер `$formatters` добавляется функция, гарантирующая передачу в модель объекта `Date`, в объект `options` добавляется поддержка функции обратного вызова `onSelect` и переопределяется функция `$render`, что позволяет обновлять виджет при изменении модели.

```
myModule.directive('datePicker', function () {
    return {
        require:'ngModel',
        link:function (scope, element, attrs, ngModelCtrl) {
            ngModelCtrl.$formatters.push(function(date) {
                if ( angular.isDefined(date) &&
                    date !== null &&
                    !angular.isDate(date) ) {
                    throw new Error('ng-Model value must be a
Date object');
                }
                return date;
```

```
            });

            var updateModel = function () {
                scope.$apply(function () {
                    var date = element.datepicker("getDate");
                    element.datepicker("setDate", element.val());
                    ngModelCtrl.$setViewValue(date);
                });
            };
            var onSelectHandler = function(userHandler) {
                if ( userHandler ) {
                    return function(value, picker) {
                        updateModel();
                        return userHandler(value, picker);
                    };
                } else {
                    return updateModel;
                }
            };
```

Обработчик `onSelect()` вызывает функцию `updateModel()` и передает новое значение даты в конвейер `$parsers` посредством `$setViewValue()`:

```
            var setUpDatePicker = function () {
                var options = scope.$eval(attrs.datePicker) || {};
                options.onSelect = onSelectHandler(options.
onSelect);
                element.bind('change', updateModel);
                element.datepicker('destroy');
                element.datepicker(options);
                ngModelCtrl.$render();
            };

            ngModelCtrl.$render = function () {
                element.datepicker("setDate", ngModelCtrl.$viewValue);
            };

            scope.$watch(attrs.datePicker, setUpDatePicker, true);
        }
    };
});
```

# В заключение

В этой главе мы рассмотрели различные приемы определения, тестирования и реализации директив. Мы увидели, как интегрировать свои директивы с директивой `ngModel` для реализации проверки, как

писать многократно используемые виджеты, и как обертывать виджеты сторонних разработчиков директивами AngularJS. На протяжении всей главы особое внимание уделялось тестированию, и описывались типичные стратегии тестирования директив в AngularJS.

В следующей главе мы постараемся глубже вникнуть в тему создания директив, рассмотрим некоторые более сложные особенности, такие как включение и компиляция наших собственных шаблонов.

# ГЛАВА 9. Создание продвинутых директив

В предыдущей главе рассказывалось, как создавать и тестировать собственные директивы. В этой главе будут описаны некоторые дополнительные возможности, которые могут пригодиться при разработке директив AngularJS, в том числе:

- прием включения: использование функций **включения** и контекста включения;
- определение контроллеров директив для создания директив, способных взаимодействовать друг с другом, а также различия контроллеров и функций связывания;
- завершение процедуры компиляции и перехват управления: динамическая загрузка собственных шаблонов, а также использование служб `$compile` и `$interpolate`.

## Включение

При перемещении элементов из одной части дерева DOM в другую необходимо решить, что делать с их контекстами.

Многие наивно полагают, что достаточно ассоциировать перемещенный элемент с контекстом в новой позиции. Однако это часто нарушает работоспособность приложения, потому что элементы могут при этом терять доступ к данным, имевшимся в первоначальном контексте.

В действительности чаще требуется «перенести контекст вместе с элементом». Перемещение элемента вместе с его контекстом называется **включением** (transclusion). Подробнее о перемещении контекста, входящего в состав включения, вы узнаете ниже, в разделе «Контекст включения». Но сначала рассмотрим несколько примеров.

## Использование включения в директивах

Прием включения требуется применять всегда, когда директива замещает оригинальное содержимое новыми элементами, но при этом оригинальное содержимое требуется сохранить для использования в новых элементах.

Например, как показано на рис. 9.1, в ходе итераций директива `ng-repeat` будет создавать копии оригинального элемента и включать их. Каждый из вновь созданных элементов будет ассоциирован с новым контекстом – потомком контекста оригинального элемента.

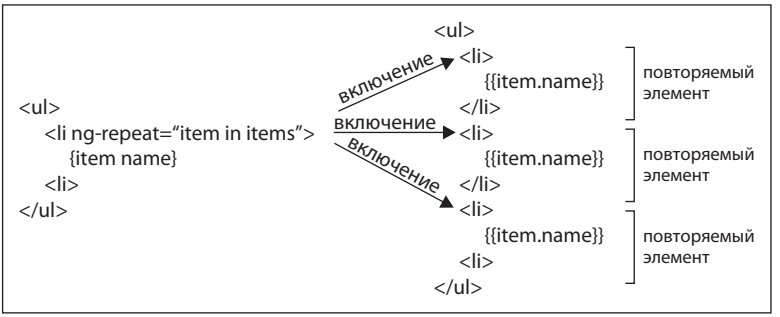


**Рис. 9.1.** Включение копий оригинального элемента директивой `ng-repeat`

## Включение в директивах с изолированным контекстом

Директива `ng-repeat` является довольно необычной, так как создает собственные копии, которые затем включаются. Более типичным примером использования приема включения являются директивы, реализующие виджеты на основе шаблонов, которые вставляют содержимое оригинального элемента в некоторую позицию в шаблоне.

## Директива вывода предупреждения на основе приема включения

Простейшим примером упомянутых выше директив виджетов на основе шаблонов является директива-элемент `alert`, реализующая вывод предупреждений (рис. 9.2).

Предупреждения – это сообщения, которые выводятся на экран с целью сообщить пользователю информацию о текущем состоянии приложения.

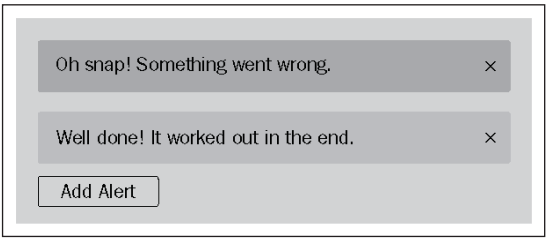


**Рис. 9.2.** Пример вывода предупреждений

Элемент `alert` содержит сообщение для отображения на экране. Его требуется включить в шаблон директивы. Список предупреждений можно отобразить с помощью директивы `ng-repeat`:

```
<alert type="alert.type" close="closeAlert($index)"
       ng-repeat="alert in alerts">
  {{alert.msg}}
</alert>
```

Атрибут `close` должен содержать выражение, выполняемое при закрытии окна с текстом предупреждения пользователем. Директива имеет достаточно простую реализацию:

```
myModule.directive('alert', function () {
    return {
        restrict:'E',
        replace: true,
        transclude: true,
        template:
            '<div class="alert alert-{{type}}">' +
                '<button type="button" class="close"' +
                        'ng-click="close()">&times;' +
                '</button>' +
                '<div ng-transclude></div>' +
            '</div>',
        scope: { type:'=', close:'&' }
    };
});
```

## Свойство replace в определениях директив

Свойство `replace` сообщает компилятору, что он должен заменить оригинальный элемент директивы шаблоном, указанным в поле

template. Если бы мы определили шаблон в поле template, но не определили свойство replace, тогда компилятор добавил бы шаблон в конец элемента директивы.

Когда компилятору предлагается заменить элемент шаблоном, он скопирует в шаблон все атрибуты, имеющиеся в оригинальном элементе.

### Свойство transclude в определениях директив

Свойство transclude может иметь значение true или 'element'. Оно предписывает компилятору извлечь содержимое оригинального элемента <alert> и сделать его доступным для включения в шаблон.

- Значение true в свойстве transclude требует включения элементов, вложенных в элемент директивы. Именно это и происходит в директиве alert, только потом элемент директивы замещается шаблоном.
- Значение 'element' в свойстве transclude требует включения всего элемента, в том числе и всех директив-атрибутов, которые еще не были скомпилированы. Так действует директива ng-repeat.

### Вставка включаемых элементов с помощью директивы ng-transclude

Директива ng-transclude извлекает включаемые элементы и добавляет их в конец элемента в шаблоне. Это самый простой и самый распространенный способ использования приема включения.

## Контекст включения

Все элементы DOM, скомпилированные фреймворком AngularJS, имеют связанный с ними контекст. В большинстве случаев элементы DOM не имеют собственного контекста, определенного непосредственно, а получают его от некоторого родительского элемента. Новые контексты создаются директивами, определяющими свойство scope в объекте определения директивы.

Лишь несколько базовых директив в AngularJS определяют новые контексты. В их числе: ng-controller, ng-repeat, ng-include, ng-view и ng-switch. Все они создают дочерние контексты, наследующие прототипы родительских контекстов.

В главе 8 рассказывалось, как создавать директивы виджетов, использующие изолированные контексты, чтобы исключить утечку данных между внутренним и внешним контекстами виджета. Наличие изолированного контекста означает, что выражения внутри шаблона не имеют доступа к значениям в родительском контексте, содержащем виджет. Это весьма полезная возможность, так как помогает избежать взаимовлияния свойств родительского контекста и происходящего внутри шаблона.

Оригинальное содержимое директивы-элемента, которое предполагается вставлять в шаблон, должно быть сохранено в оригинальном, а не в изолированном контексте. Выполняя включение оригинальных элементов, мы получаем возможность управлять контекстами этих элементов.

Наша директива `alert` – это виджет с изолированным контекстом. Давайте посмотрим, какие контексты создаются директивой `alert`. До компиляции директивы `alert`, дерево DOM и контексты в нем имели следующий вид:

```
<!-- определяет корневой контекст $rootScope -->
<div ng-app ng-init="type='success'">
  <!-- связывается с корневым контекстом $rootScope -->
  <div>{{type}}</div>
  <!-- связывается с корневым контекстом $rootScope -->
  <alert type="'info'" ...>Look at {{type}}</alert>
</div>
```

Элемент `<div>{{type}}</div>` не имеет собственного контекста, поэтому он неявно связывается с корневым контекстом `$rootScope`, так как является потомком элемента `ng-app`, где определяется `$rootScope`, и, соответственно, выражение `{{type}}` будет интерпретироваться как `'success'`.

В элементе `alert` имеется атрибут `type="'info'"`. Этот атрибут отображается в свойство `type` контекста шаблона. В процессе компиляции директива `alert` будет замещена ее шаблоном, и дерево DOM вместе с его контекстами приобретет следующий вид:

```
<!-- определяет корневой контекст $rootScope -->
<div ng-app ng-init="type='success'">
  <!-- связывается с корневым контекстом $rootScope -->
  <div>{{type}}</div>
   <!-- определяет изолированный контекст -->
  <div class="alert-{{type}}" ...>
    <!-- связывается с изолированным контекстом -->
```

```
      <button>...</button>
      <div ng-transclude>
        <!-- определяет новый контекст включения -->
        <span>Look at {{type}}</span>
      </div>
    </div>
  </div>
```

Внутри шаблона атрибут `class="alert-{{type}}"` неявно окажется связан с изолированным контекстом и в результате будет интерпретироваться как `class="alert-info"`.

Включаемое содержимое – `<span>Look at {{type}}</span>` – оригинального элемента `<alert>` напротив, будет связано с новым контекстом включения. Если бы мы просто поместили это содержимое в шаблон, оно оказалось бы привязано не к корневому контексту `$rootScope`, а к изолированному и выражение `{{type}}` вернуло бы `'info'`. Но это совсем не то, что нам требуется.

Новый контекст включения является потомком корневого контекста `$rootScope` и наследует его прототип. Это означает, что элемент `span` будет благополучно преобразован в `<span>Look at success</span>`. На рис. 9.3 показаны отношения между контекстами в окне дополнения Batarang:

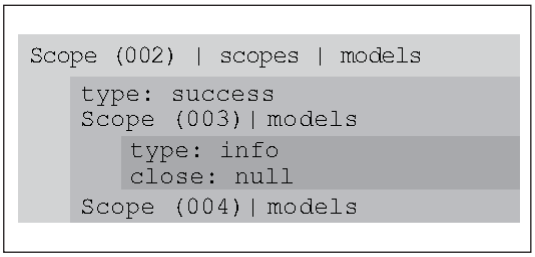


**Рис. 9.3.** Отношения между контекстами

Область `Scope (002)` на рис. 9.3 – это корневой контекст `$rootScope`, содержащий свойство `type="success"`. Область `Scope (003)` – это изолированный контекст шаблона директивы `alert`, он не наследует `$rootScope`. Область `Scope (004)` – это контекст включения, он наследует `$rootScope` и поэтому при обращении к его свойству `type` будет получено значение `'success'`.

Включаемые элементы получают оригинальный контекст. Точнее говоря, они получают новый контекст, наследующий прототип оригинального контекста, откуда произведено включение.

# Создание и использование функций включения

Включение в AngularJS выполняется с помощью **функций включения**. Эти функции являются обычными функциями связывания, которые создаются обращением к службе `$compile`.

Встретив директиву, требующую выполнить включение, AngularJS извлекает включаемые элементы из дерева DOM и компилирует их. Ниже показано, что примерно происходит, когда компилятор обнаруживает свойство `transclude: true` в определении директивы:

```
var elementsToTransclude = directiveElement.contents();
directiveElement.html('');
var transcludeFunction = $compile(elementsToTransclude);
```

Первая строка извлекает элемент, содержащийся в директиве, требующей включения. Вторая строка очищает этот элемент. Третья строка компилирует включаемое содержимое и создает функцию включения, которая затем возвращается директиве для последующего использования.

## Создание функции включения с помощью службы $compile

Компилятор AngularJS экспортирует службу `$compile`. Это та же самая функция, которая используется для компиляции других частей приложения на основе AngularJS. Чтобы воспользоваться этой службой, достаточно просто вызвать ее и передать ей список узлов DOM (или строку, которая может быть преобразована в список узлов DOM).

```
var linkingFn = $compile(
    '<div some-directive>Some {{"interpola-ted"}} values</div>');
```

Обратно служба `$compile` возвращает функцию связывания. Функция связывания принимает контекст и возвращает элемент DOM, содержащий скомпилированные элементы DOM, связанные с указанным контекстом:

```
var compiledElement = linkingFn(someScope);
```

Функции включения – это всего лишь специализированные версии функций связывания.

### Копирование оригинальных элементов при включении

Если функции связывания передать во втором параметре функцию обратного вызова, вместо оригинальных элементов она вернет их копии. Функция обратного вызова будет вызвана синхронно и получит копию элемента в виде параметра.

```
var clone = linkingFn(scope, function callback(clone) {
    element.append(clone);
});
```

Эта особенность может пригодиться, если потребуется создать копии потомков оригинального элемента, как, например, в директиве `ng-repeat`.

## Использование функций включения в директивах

Компилятор возвращает созданную им функцию включения директиве. В директиве имеется два места, где можно получить доступ к функции включения: функция `compile` и контроллер директивы. Подробнее о контроллерах директив рассказывается ниже в этой главе, а сейчас остановимся на функции `compile`.

```
myModule.directive('myDirective', function() {
    return {
        transclude: true,
        compile: function(element, attrs, transcludeFn) { ... };
        controller: function($scope, $transclude) { ... },
    };
});
```

В этом фрагменте мы особо подчеркнули, что директива должна включать свое содержимое. К функции включения можно обратиться из функции `compile`, через параметр `transcludeFn`, и из контроллера директивы, через параметр `$transclude`.

### Вызов функции включения из функции compile через параметр transcludeFn

Функция включения передается в функцию `compile` директивы в третьем параметре. На этапе компиляции контекст еще не известен и как результат функция включения не связана ни с каким контекстом.

Поэтому при вызове этой функции вы должны передать ей контекст в первом параметре.

В функции связывания контекст доступен, и поэтому чаще всего функция включения вызывается из нее.

```
compile: function(element, attrs, transcludeFn) {
    return function postLink(scope, element, attrs,
controller) {
        var newScope = scope.$parent.$new();
        element.find('p').first().append(transcludeFn(newScope));
    };
}
```

Здесь мы добавляем включаемые элементы в первый элемент <p>, находящийся внутри элемента директивы. Вызывая функцию включения, мы связываем включаемые элементы с контекстом. В данном случае мы создаем новый контекст, который является братским по отношению к контексту директивы, то есть, дочерним, по отношению к контексту $parent, являющемуся родительским для контекста директивы.

Это совершенно необходимо, когда директива имеет изолированный контекст, потому что контекст, передаваемый функции связывания, является изолированным и не наследует свойства родительского контекста, необходимые включаемым элементам.

### Вызов функции включения контроллера директивы через параметр $transclude

Благодаря внедрению в параметр $transclude, функцию включения можно также вызвать из контроллера директивы. В этом случае $transclude представляет функцию, уже связанную с новым контекстом, поэтому нет необходимости передавать ей контекст явно.

```
controller: function($scope, $element, $transclude) {
    $element.find('p').first().append($transclude());
}
```

Здесь так же включаемые элементы добавляются в конец первого вложенного элемента <p>.

В параметре $transclude передается функция включения, уже связанная с контекстом, наследующим прототип оригинального контекста, откуда взяты включаемые элементы.

## Создание директивы if, использующей включение

Рассмотрим пример реализации простой директивы, явно использующей функцию включения вместо директивы `ng-transclude`. В состав фреймворка AngularJS 1.0 уже входят директивы `ng-show` и `ng-switch`, позволяющие изменять видимость содержимого, но `ng-show` не удаляет невидимые элементы из дерева DOM, а `ng-switch` имеет слишком объемный синтаксис, из-за чего ее неудобно использовать в простых ситуациях.

Для случаев, когда требуется просто удалить ненужные элементы из дерева DOM, можно создать директиву `if`, которую можно было бы использовать на манер директивы `ng-show`:

```
<body ng-init="model= {show: true, count: 0}">
  <button ng-click="model.show = !model.show">
    Toggle Div
  </button>
  <div if="model.show" ng-init="model.count=model.count+1">
    Shown {{model.count}} times
  </div>
</body>
```

Каждый раз, когда выполняется щелчок на кнопке, свойство `model.show` переключается между значениями `true` и `false`. Чтобы показать, что элемент DOM действительно удаляется из дерева документа и вновь вставляется в него, предусмотрено наращивание счетчика `model.count`.

В модульном тесте нужно убедиться в корректном удалении и добавлении элемента DOM:

```
it('creates or removes the element as the if condition changes',
function () {
    element = $compile(
        '<div><div if="someVar"></div></div>')(scope);
    scope.$apply('someVar = true');
    expect(element.children().length).toBe(1);
    scope.$apply('someVar = false');
    expect(element.children().length).toBe(0);
    scope.$apply('someVar = true');
    expect(element.children().length).toBe(1);
});
```

В этом тесте проверяется увеличение или уменьшение количества дочерних элементов, когда выражение возвращает `true` или `false`.

Обратите внимание, что нам потребовалось заключить элемент с директивой `if` в элемент. Это обусловлено тем, что наша директива будет выполнять вставку элемента с помощью метода `jqLite.after()`, который требует наличия родителя у элемента.

А теперь рассмотрим реализацию директивы:

```
myModule.directive('if', function () {
    return {
        transclude: 'element',
        priority: 500,
        compile: function (element, attr, transclude) {
            return function postLink(scope, element, attr) {
                var childElement, childScope;

                scope.$watch(attr['if'], function (newValue) {
                    if (childElement) {
                        childElement.remove();
                        childScope.$destroy();
                        childElement = undefined;
                        childScope = undefined;
                    }
                    if (newValue) {
                        childScope = scope.$new();
                        childElement = transclude(childScope,
function(clone){
                            element.after(clone);
                        });
                    }
                });
    ...
```

Директива включает элемент целиком (`transclude: 'element'`). Мы реализовали функцию `compile`, дающую доступ к функции включения, которая возвращает функцию связывания, где с помощью функции `$watch` следим за изменением значения выражения в атрибуте `if`.

Здесь используется функция `$watch`, а не `$observe`, потому что атрибут `if` должен содержать выражение, а не интерполируемую строку.

Когда результат выражения изменяется, мы удаляем дочерний элемент и его контекст, если они существуют. Это следует сделать обязательно, чтобы избежать утечек памяти. Если выражение возвращает `true`, создается новый дочерний контекст, который затем передается

функции включения для связывания с новой копией включаемого элемента. Копия вставляется после элемента, содержащего директиву.

### Использование свойства priority директивы

Все директивы имеют определенный приоритет, который по умолчанию получает нулевое значение, как в случае с директивой `alert`. Внутри каждого элемента AngularJS сначала компилирует директивы с более высоким приоритетом. Таким образом, манипулируя свойством `priority` в объектах определения директив, можно изменять порядок их компиляции.

Если директива имеет свойство `transclude: 'element'`, компилятор выполнит включение директив-атрибутов, имеющих более низкий приоритет, чем у текущей, то есть, директив-атрибутов необработанных к этому моменту.

Директива `ng-repeat` имеет свойства `transclude: 'element'` и `priority: 1000`, поэтому обычно все атрибуты, присутствующие в элементе `ng-repeat`, включаются в копии повторяемых элементов.

Мы присвоили директиве `if`, в примере выше, приоритет 500, ниже приоритета директивы `ng-repeat`. Это означает, что если добавить ее в тот же элемент, где уже присутствует директива `ng-repeat`, выражение в директиве `if` будет использовать контексты, создаваемые в итерациях.

В этой директиве механизм включения позволяет получить содержимое элемента директивы, связать его с требуемым контекстом и при выполнении условия добавить в дерево DOM.

А теперь сменим тему и рассмотрим приемы реализации контроллеров для отдельных директив.

# Контроллеры директив

Контроллером в AngularJS называется объект, подключенный к элементу DOM, который инициализирует и добавляет поведение к контексту этого элемента.

Мы уже видели множество контроллеров приложений, создаваемых директивой ng-controller. От них не требовалось непосредственного взаимодействия с деревом DOM – достаточно было выполнения операций с текущим контекстом.

Контроллер директивы – это особая разновидность контроллера, объявляемого директивой и создаваемого всякий раз, когда директива встречается в элементе DOM. Его цель – инициализировать и придать динамические черты уже не контексту, а директиве.

Контроллер директивы определяется с помощью свойства `controller` объекта определения директивы. Значением этого свойства может быть строка с именем контроллера, уже имеющегося в модуле:

```
myModule.directive('myDirective', function() {
    return {
        controller: 'MyDirectiveController'
    };
});
myModule.controller('MyDirectiveController', function($scope) {
    ...
});
```

Или функция-конструктор, создающая экземпляр контроллера:

```
myModule.directive('myDirective', function() {
    return {
        controller: function($scope, ...) { ... }
    };
});
```

Если контроллер определен в модуле, его легко можно протестировать отдельно от директивы. Но это так же означает, что контроллер доступен всему приложению через механизм внедрения зависимостей, поэтому приходится проявлять особую осторожность, чтобы не вызвать конфликт имен с контроллерами в других модулях, используемых в приложении.

Встроенные модули, объявленные в виде анонимных функций, сложнее протестировать отдельно от директив, но они дают возможность обеспечить приватность реализации директив.

## Внедрение специальных зависимостей в контроллеры директив

Как и любые другие контроллеры, контроллеры директив могут внедряться как зависимости средствами AngularJS. Все контроллеры внедряются вместе со службой `$scope`, также можно указать другие службы для внедрения, такие как `$timeout` или `$rootScope`. Кроме того, контроллеры директив позволяют внедрять вместе с ними три специальные службы:

- `$element`: ссылка на элемент DOM директивы; обертка для доступа к функциям из библиотеки jQLite/jQuery;
- `$attrs`: нормализованный список атрибутов, присутствующих в элементе DOM директивы;
- `$transclude`: функция включения, уже связанная с контекстом.

## Создание директивы постраничного просмотра на основе контроллера

С функциональной точки зрения контроллеры директив и функции связывания имеют много общего. Часто функцию связывания можно заменить контроллером. Ниже представлена версия директивы постраничного просмотра из главы 8, но на этот раз реализованная с использованием контроллера директивы вместо функции связывания:

```
myModule.directive('pagination', function() {
    return {
        restrict: 'E',
        scope: { numPages: '=', currentPage: '=', onSelectPage: '&' },
        templateUrl: 'template/pagination.html',
        replace: true,
        controller: ['$scope, '$element', '$attrs',
                     function($scope, $element, $attrs) {
            $scope.$watch('numPages', function(value) {
                $scope.pages = [];
                for(var i=1;i<=value;i++) {
                    $scope.pages.push(i);
                }
                if ( $scope.currentPage > value ) {
                    $scope.selectPage(value);
                }
            });
            $scope.noPrevious = function() {
                return $scope.currentPage === 1;
            };
            ...
        }]
        ...
    });
```

В данном простом случае между этой версией и версией на основе функции связывания имеется только одно отличие – функция связывания получает контекст, элемент, атрибуты и контроллер в виде параметров `scope`, `element`, `attrs` и `controller`, а контроллер дирек-

тивы должен использовать аннотации механизма внедрения зависимостей, представленные службами `$scope`, `$element` и `$attrs`.

## Различия между контроллерами директив и функциями связывания

Делая выбор между функциями связывания и контроллерами директив полезно знать, чем они отличаются.

### Внедрение зависимостей

Во-первых, как было показано выше, контроллеры директив должны использовать аннотации механизма внедрения зависимостей, чтобы определить необходимые им службы, например: `$scope`, `$element` и `$attrs`. Функции связывания всегда получают одни и те же четыре параметра: `scope`, `element`, `attrs` и `controller`, независимо от имен этих параметров в определении функции.

### Процедура компиляции

Контроллеры директив и функции связывания вызываются на разных этапах процедуры компиляции. Если представить множество директив в элементах DOM, как показано на рис. 9.4

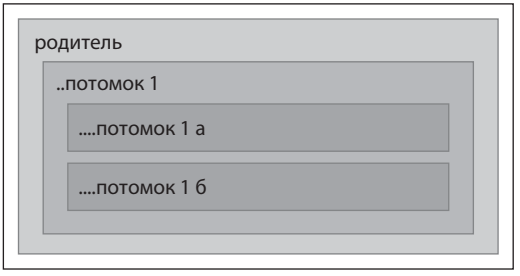


**Рис. 9.4.** Пример структуры элементов DOM

Контроллеры директив и функции связывания будут вызваны в следующем порядке:

- родитель (контроллер);
- родитель (функция предварительного связывания);
  - потомок 1 (контроллер) ;
  - потомок 1 (функция предварительного связывания) ;
  - потомок 1 а (контроллер) ;
  - потомок 1 а (функция предварительного связывания) ;

    - потомок 1 а (функция заключительного связывания) ;
    - потомок 1 б (контроллер) ;
    - потомок 1 б (функция предварительного связывания) ;
    - потомок 1 б (функция заключительного связывания) ;
    - потомок 1 (функция заключительного связывания) ;
- родитель (функция заключительного связывания).

Если элемент содержит несколько директив, тогда для этого элемента:

- при необходимости создается новый контекст;
- для каждой директивы создается контроллер;
- для каждой директивы вызывается функция предварительного связывания;
- связываются все дочерние элементы;
- для каждой директивы вызывается функция заключительного связывания.

Из этого следует, что когда создается контроллер директивы, элемент директивы и дочерние элементы еще не связаны полностью. Но, когда вызываются функции связывания (предварительного или заключительного), все контроллеры директив уже созданы. Именно поэтому контроллеры директив могут передаваться в функции связывания.

Функция предварительного связывания вызывается после того, как компилятор завершит компиляцию и компоновку элемента и всех его дочерних элементов. Это означает, что никакие изменения в дереве DOM, произведенные на этом этапе, не будут обнаружены компилятором AngularJS.

Знание этого обстоятельства может пригодиться при внедрении в элементы сторонних библиотек, таких как расширения jQuery, которые могут вносить изменения в дерево DOM, вводящие компилятор AngularJS в заблуждение.

## Доступ к другим контроллерам

Функции связывания принимают в четвертом параметре контроллеры любых директив, указанных в свойстве `required`. В главе 8 уже было показано, как получить доступ к контроллеру `ngModelController`.

```
myModule.directive('validateEquals', function() {
    return {
```

```
        require: 'ngModel',
        link: function(scope, elm, attrs, ngModelCtrl) {
            ...
        };
    });
```

Здесь директиве `validateEquals` требуется контроллер директивы `ngModel`, который передается функции связывания в виде параметра `ngModelCtrl`.

Контроллер директивы, напротив, не в состоянии обеспечить внедрение других контроллеров директив.

### Доступ к функции включения

Как описывалось в разделе, посвященном функциям включения, контроллеры директив позволяют внедрять функцию `$transclusion`, уже связанную с нужным контекстом.

Функции связывания имеют доступ к функциям включения только через замыкание в функции `compile`, но в этой функции контекст пока еще недоступен.

## *Комплект директив виджета «аккордеон»*

Контроллеры директив добавляют динамическое поведение в элементы директив и могут использоваться другими директивами. Это позволяет создавать наборы директив, способных взаимодействовать друг с другом.

В этом разделе мы познакомимся с реализацией виджета «аккордеон» (accordion), изображенного на рис. 9.5. Этот виджет представляет список групп, которые могут сворачиваться и разворачиваться щелчком мыши на их заголовках. Щелчок на заголовке группы вызывает ее разворачивание и сворачивание других групп.

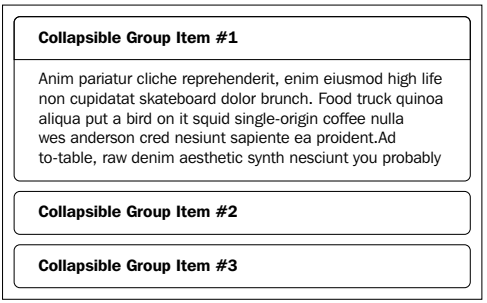


**Рис. 9.5.** Виджет «аккордеон»

Ниже представлена разметка HTML для виджета на рис. 9.5:

```
<accordion>
  <accordion-group heading="Heading 1">
    Group 1 <strong>Body</strong>
   </accordion-group>
   <accordion-group heading="Heading 2">
    Group 2 <strong>Body</strong>
   </accordion-group>
</accordion>
```

Здесь присутствуют две новые директивы: `accordion`, служащая контейнером для групп, и `accordion-group`, определяющая содержимое одной группы.

## Использование контроллера директивы в виджете «аккордеон»

Чтобы обеспечить взаимодействие групп, директива `accordion` определяет собственный контроллер `AccordionController`, а каждая директива `accordion-group` внедряет его.

Контроллер `AccordionController` экспортирует два метода, `addGroup` и `closeOthers`. С помощью первого директивы `accordion-group` регистрируют себя в виджете «аккордеон», а с помощью второго – сообщают остальным директивам `accordion-groups`, что они должны свернуться, когда данная директива разворачивается.

Тестирование контроллера директивы близко напоминает тестирование контроллера приложения (см. главу 2). Ниже приводится тест для проверки метода `closeOthers`:

```
describe('closeOthers', function() {
    var group1, group2, group3;
    beforeEach(function() {
        ctrl.addGroup(group1 = $scope.$new());
        ctrl.addGroup(group2 = $scope.$new());
        ctrl.addGroup(group3 = $scope.$new());
        group1.isOpen = group2.isOpen = group3.isOpen = true;
    });
    it('closes all groups other than the one passed', function() {
        ctrl.closeOthers(group2);
        expect(group1.isOpen).toBe(false);
        expect(group2.isOpen).toBe(true);
        expect(group3.isOpen).toBe(false);
    });
});
```

Здесь создаются три группы, каждая из которых имеет атрибут `isOpen=true`. После вызова метода `closeOthers` для группы `group2`

проверяется значение свойства `isOpen`, которое в группах `group1` и `group2` должно получить значение `false`.

Ниже приводится реализация контроллера `AccordionController`:

```
myModule.controller('AccordionController', ['$scope', '$attrs',
    function ($scope, $attrs) {
        this.groups = [];
        this.closeOthers = function(openGroup) {
            angular.forEach(this.groups, function (group) {
                if ( group !== openGroup ) {
                    group.isOpen = false;
                }
            });
        }
    };

    this.addGroup = function(groupScope) {
        var that = this;
        this.groups.push(groupScope);
        groupScope.$on('$destroy', function (event) {
            that.removeGroup(groupScope);
        });
    };

    this.removeGroup = function(group) {
        var index = this.groups.indexOf(group);
        if ( index !== -1 ) {
            this.groups.splice(this.groups.indexOf(group), 1);
        }
    };
}]);
```

Обратите внимание, что при удалении группы из списка автоматически уничтожается и соответствующий ей контекст. Это важно, потому что список групп может формироваться динамически, во время выполнения, с помощью директивы `ng-repeat`, которая может удалять элементы и контексты групп из приложения. Если приложение будет удерживать ссылки на контексты удаляемых групп, они не смогут быть утилизированы сборщиком мусора.

## Реализация директивы accordion

Главная директива `accordion` просто объявляет контроллер `AccordionController` контроллером директивы и добавляет CSS-класс `accordion` в свой элемент в функции связывания:

```
myModule.directive('accordion', function () {
    return {
        restrict:'E',
```

```
        controller:'AccordionController',
        link: function(scope, element, attrs) {
            element.addClass('accordion');
        }
    };
})
```

## Реализация директивы accordion-group

Каждая группа в виджете «аккордеон» определяется директивой `accordion-group`. Группа состоит из ссылки и тела. Ниже приводится шаблон разметки для этой директивы:

```
<div class="accordion-group">
  <div class="accordion-heading" >
    <a class="accordion-toggle"
       ng-click="isOpen=!isOpen">{{heading}}</a>
  </div>
  <div class="accordion-body" ng-show="isOpen">
    <div class="accordion-inner" ng-transclude></div>
  </div>
</div>
```

Оригинальные дочерние элементы директивы включаются в тело шаблона. Шаблон ссылается на свойства `isOpen` и `heading` в текущем контексте. Нам нужен полный контроль над этими значениями, поэтому директива `accordion-group` будет иметь изолированный контекст.

В тестах выполняется настройка директивы `accordion` и нескольких директив `accordion-group`, и затем проверяется корректное разворачивание и сворачивание групп. Ниже приводится пример реализации тестов:

```
describe('accordion-group', function () {
    var scope, element, groups;
    beforeEach(inject(function($rootScope, $compile) {
        scope = $rootScope;
        var tpl =
"<accordion>" +
"<accordion-group heading='title 1'>Content 1</accordion-group>" +
"<accordion-group heading='title 2'>Content 2</accordion-group>" +
"</accordion>";
        $compile(tpl)(scope);
        scope.$digest();
        groups = element.find('.accordion-group');
    });
    ...
    it('should change selected element on click', function () {
```

```
            groups.eq(0).find('a').click();
            expect(findGroupBody(0).scope().isOpen).toBe(true);
            groups.eq(1).find('a').click();
            expect(groups.eq(0).scope().isOpen).toBe(false);
            expect(groups.eq(1).scope().isOpen).toBe(true);
        });
        ...
});
```

Тест генерирует событие щелчка мышью на заголовке группы и проверяет свойство `isOpen` в контекстах других групп, которое должно получить значение `false`.

Реализация самой директивы чрезвычайно проста:

```
myModule.directive('accordionGroup', function() {
    return {
        require:'^accordion',
        restrict:'E',
        transclude:true,
        replace: true,
        templateUrl:'template/accordion/accordion-group.html',
        scope:{ heading:'@' },
        link: function(scope, element, attrs, accordionCtrl) {
            accordionCtrl.addGroup(scope);
            scope.isOpen = false;
            scope.$watch('isOpen', function(value) {
                if ( value ) {
                    accordionCtrl.closeOthers(scope);
                }
            });
        }
    };
});
```

Как видите, эта директива использует контроллер директивы `accordion`, чтобы включить себя в элемент DOM родительской директивы. Контроллер директивы, `accordionCtrl`, внедряется в четвертый параметр функции связывания. С его помощью директива `accordion-group` регистрирует себя с помощью функции `addGroup()` и вызывает `closeOthers()`, когда данная группа разворачивается.

# Управление процессом компиляции

Иногда возникают ситуации, когда требуется иметь более полный контроль над процессом компиляции и связывания элементов и их

потомков. Такая возможность может пригодиться для организации динамической загрузки шаблона или для управления включением элементов в шаблон директивы. В этих случаях нам предоставляется возможность завершить процесс компиляции, изменить элемент директивы и вложенные в него элементы, а затем скомпилировать их вручную.

## Создание директивы field

При создании приложений с большим количеством форм, разработчики часто замечают, что им приходится писать массу избыточной и повторяющейся разметки HTML для определения полей форм.

Например, для каждого поля требуется определить элементы `input` и `label`, заключенные в элементы `div` и `span`. В этих элементах требуется указать несколько атрибутов, таких как `ng-model`, `name`, `id` и `for`, которые обычно получают похожие или даже идентичные значения, а также различные классы CSS. Помимо этого необходимо предусмотреть вывод предупреждений, если пользователь ввел недопустимое значение.

В результате получается разметка HTML формы, как показано ниже:

```
<div class="control-group"
     ng-class="{'error' : form.email.$invalid && form.email.$dirty,
                'success' : form.email.$valid && form.email.$dirty}">
  <label for="email">E-mail</label>
  <div class="controls">
    <input type="email" id="email" name="email" ng-model="user.email"
           required>
    <span ng-show="form.email.$error['required'] && form.email.dirty"
          class="help-inline">Email is required</span>
    <span ng-show=" form.email.$error['email'] && form.email.dirty"
          class="help-inline">Please enter a valid email</span>
  </div>
</div>
```

От значительной массы повторяющегося кода можно избавиться, создав директиву `field`. Она будет вставлять соответствующий шаблон, содержащий элемент ввода с меткой. Ниже приводится пример использования директивы `field`:

```
<field type="email" ng-model="user.email" required >
  <label>Email</label>
  <validator key="required">$fieldLabel is required</validator>
  <validator key="email">Please enter a valid email</validator>
</field>
```

Здесь директивы `ng-model`, `type` и `validation` просто передаются в элемент ввода в виде атрибутов директивы `field`. Затем определяются дочерние элементы: `label` и сообщения об ошибках. Обратите так же внимание на возможность использовать свойство `$fieldLabel` внутри директив проверки. Это свойство будет добавлено директивой `field` в контекст директив проверки.

Директива `field` предъявляет ряд требований, которые сложно удовлетворить с использованием имеющегося прикладного интерфейса директив.

- Вместо единственного шаблона для всей директивы необходима возможность вставлять различные шаблоны, в зависимости от типа поля ввода. Мы не можем использовать свойство `template` (или `templateUrl`) объекта определения директивы.
- Требуется генерировать уникальные значения атрибутов `name` и `id` для элементов `input` и связывать атрибут `for` элемента `label` до компиляции директивы `ng-model`.
- Необходимо извлекать определения сообщений об ошибках из дочерних элементов `validator` и использовать их в шаблоне.

Объект определения директивы `field` выглядит, как показано ниже:

```
restrict:'E',
priority: 100,
terminal: true,
compile: function(element, attrs) {
    ...
    var validationMgs = getValidationValidationMessages(element);
    var labelContent = getLabelContent(element);

    element.html('');

    return function postLink(scope, element, attrs) {
        var template = attrs.template || 'input.html';
        loadTemplate(template).then(function(templateElement) {
            ...
        });
    };
}
```

Мы указали в свойстве `priority` этой директивы значение 100, чтобы гарантировать ее обработку компилятором перед директивой `ng-model` в том же элементе, и принудительно завершаем компиляцию. В функции `compile` мы извлекаем сообщения об ошибках вы-

зовом `getValidationMessageMap` и информацию о метке вызовом `getLabelContent`. После этого мы очищаем содержимое элемента, чтобы получить чистый элемент, подготовленный к загрузке шаблона. Функция `compile` возвращает функцию `postLink`, выполняющую загрузку нужного шаблона.

### Использование свойства terminal в директивах

Если в объекте определения директивы имеется свойство `terminal: true`, компилятор остановит работу и не будет обрабатывать дочерние элементы или любые другие директивы в данном элементе, имеющие более низкий приоритет.

Даже если директива принудительно прекращает компиляцию, контролер директивы, функция `compile` и функция связывания все равно вызываются.

Прервав компиляцию можно изменить элемент директивы и вложенные в него элементы, но при этом придется вручную создать и настроить соответствующие контексты, выполнить включение содержимого, а также произвести дальнейшую компиляцию дочерних элементов, которые тоже могут содержать директивы.

В большинстве случаев AngularJS автоматически интерполирует строки в выражения, когда в шаблонах используются фигурные скобки `{{}}`. Но в данной директиве нам необходимо вручную выполнить интерполяцию таких строк. Сделать это можно с помощью службы `$interpolate`.

## Использование службы $interpolate

Функция `getLabelContent` просто копирует содержимое элемента `label`, находящийся в директиве, в элемент `label`, находящийся в шаблоне, где оно будет скомпилировано вместе с шаблоном:

```
function getLabelContent(element) {
    var label = element.find('label');
    return label[0] && label.html();
}
```

Но, чтобы организовать вывод сообщений об ошибках только для тех проверок, которые в настоящий момент потерпели неудачу, нам придется задействовать директиву `ng-repeat`. То есть нам нужно сохранить сообщения в контексте шаблона, в свойстве

$validationMessages. Эти сообщения могут содержать строки, требующие интерполяции, поэтому выполним их интерполяцию на этапе компиляции:

```
function getValidationMessageMap(element) {
    var messageFns = {};
    var validators = element.find('validator');
    angular.forEach(validators, function(validator) {
        validator = angular.element(validator);
        messageFns[validator.attr('key')] =
            $interpolate(validator.text());
    });
    return messageFns;
}
```

Для каждого элемента <validator> вызывается служба $interpolate, создающая функцию интерполяции из текста элемента, и затем эта функция добавляется в хеш, с ключом, соответствующим значению атрибута key элемента validator. Данный хеш будет добавлен в контекст шаблона, как свойство $validationMessages.

Служба $interpolate используется фреймворком AngularJS повсеместно, для интерполяции строк, содержащих фигурные скобки {{}}. Если передать службе такую строку, она вернет функцию интерполяции, принимающую контекст и возвращающую интерполированную строку:

```
var getFullName = $interpolate('{{first}}{{last}}');
var scope = { first:'Pete',last:'Bacon Darwin' };
var fullName = getFullName(scope);
```

Здесь из строки '{{first}} {{last}}' создается функция интерполяции getFullName, которая затем вызывается с объектом scope. В результате переменной fullName будет присвоена строка 'Pete Bacon Darwin'.

### Привязка сообщений об ошибках

Чтобы обеспечить вывод сообщений об ошибках, необходимо предусмотреть в шаблонах полей примерно такую разметку:

```
<span class="help-inline" ng-repeat="error in $fieldErrors">
    {{$validationMessages[error](this)}}
</span>
```

Она реализует обход всех ключей error в $fieldErrors и связывает результат вызова функции интерполяции для каждого из них.

Функции интерполяции необходимо передать текущий контекст. В шаблоне это достигается передачей переменной `this`, которая ссылается на текущий контекст. Забывчивость в данном случае может привести к непредсказуемому поведению и сложным в отладке ошибкам.

Свойство `$fieldErrors` содержит список ключей текущих сообщений об ошибках. Он обновляется функцией-наблюдателем, созданной в обработчике успешного завершения для `loadTemplate()`.

## Динамическая загрузка шаблонов

Функция `loadTemplate` загружает указанный шаблон и преобразует его в объект jqLite/jQuery, обертывающий элемент DOM:

```
function loadTemplate(template) {
    return $http.get(template, {cache:$templateCache})
        .then(function(response) {
            return angular.element(response.data);
        }, function(response) {
            throw new Error('Template not found: ' + template);
        });
    }
```

Функция вызывается асинхронно и поэтому возвращает обернутый элемент в виде отложенного результата. Так же, как директивы, использующие свойство `templateUrl`, и директива `ng-include`, мы пользуемся службой `$templateCache` для кеширования загружаемых шаблонов.

## Настройка шаблона директивы field

Функция `loadTemplate` вызывается в функции связывания директивы `field` и ей передается значение атрибута `template` из элемента директивы (или `'input.html'`, если определение атрибута отсутствует).

```
loadTemplate(template).then(function(templateElement) {
```

Все основные действия, предусматриваемые директивой, выполняются, как только отложенный результат получает фактическое значение.

```
    var childScope = scope.$new();
    childScope.$validationMessages = angular.copy(validationMsgs);
```

```
        childScope.$fieldId = attrs.ngModel.replace('.', '_').toLowerCase()
    + '_' + childScope.$id;
        childScope.$fieldLabel = labelContent;

        childScope.$watch('$field.$dirty && $field.$error',
            function(errorList) {
                childScope.$fieldErrors = [];
                angular.forEach(errorList, function(invalid, key) {
                    if ( invalid ) {
                        childScope.$fieldErrors.push(key);
                    }
                });
        }, true);
```

Сначала создается новый дочерний контекст и дополняется необходимыми свойствами, такими как `$validationMessages`, `$fieldId`, `$fieldLabel` и `$fieldErrors`:

```
        var inputElement = findInputElement(templateElement);
        angular.forEach(attrs.$attr, function (original, normalized) {
            var value = element.attr(original);
            inputElement.attr(original, value);
        });
        inputElement.attr('name', childScope.$fieldId);
        inputElement.attr('id', childScope.$fieldId);
```

Затем копируются все атрибуты из элемента директивы `field` в элемент `input` шаблона и добавляются вычисленные значения атрибутов `name` и `id`:

```
        var labelElement = templateElement.find('label');
        labelElement.attr('for', childScope.$fieldId);
        labelElement.html(labelContent);
```

Потом копируется содержимое `labelContent` и к элементу `label` применяется атрибут `for`:

```
        element.append(templateElement);
        $compile(templateElement)(childScope);
        childScope.$field = inputElement.controller('ngModel');
    });
```

Наконец, полученный элемент `templateElement` добавляется в оригинальный элемент `field` и вызывается служба `$compile` для его компиляции и связывания с новым контекстом `childScope`. После связывания элемента становится доступным контроллер `ngModelController`, который тут же сохраняется в свойстве `$field` для последующего использования шаблоном.

# В заключение

В этой главе мы познакомились с некоторыми продвинутыми аспектами разработки директив. На примере реализации директивы `alert` мы увидели, как при создании виджетов можно использовать механизм включения, представленный директивой `ng-transclude`. На примере реализации виджета «аккордеон» было продемонстрировано, как можно организовать взаимодействия между контроллерами директив. В примере реализации директивы `field` мы даже взяли под свой контроль процесс компиляции, завершая его в текущей директиве и продолжая компиляцию вложенных элементов вручную, с помощью службы `$compile`.

В следующей главе рассказывается, как интернационализировать приложения, чтобы ими могли пользоваться люди, говорящие на разных языках.

# ГЛАВА 10. Создание интернациональных веб-приложений

Мы живем в мире, где любой, имеющий подключение к Интернету, может получить доступ к вашему веб-приложению. На начальном этапе ваш проект может поддерживать только один язык, но с ростом популярности веб-сайта на международной сцене у вас наверняка появится желание предложить пользователям содержимое на их родных языках. Или вы обязаны будете под давлением клиентов или законодательства реализовать несколько локализованных версий. Независимо от причин, проблемы интернационализации давно уже стали неотъемлемой частью жизни веб-разработчиков.

Существует множество аспектов, касающихся **интернационализации** (internationalization, i18n) и **локализации** (localization, l10n), но в этой главе мы сосредоточимся на проблемах и их решениях, характерных для веб-приложений, построенных на основе AngularJS. В частности, здесь вы увидите, как:

- настраивать форматы отображения дат, чисел и валюты, и как выбирать другие национальные настройки, опираясь на предпочтения пользователей;
- обрабатывать содержимое, переведенное на разные языки (встроенное в шаблоны AngularJS или «зашитое» в код на JavaScript).

В последней части этой главы будет представлено несколько шаблонов проектирования, советов и рекомендаций, которые могут пригодиться при создании интернациональных приложений на основе AngularJS.

# Использование национальных наборов символов и настроек

Фреймворк AngularJS содержит ряд модулей с национальными настройками. В этом разделе будут показаны шаги, которые необходимо выполнить для настройки этих модулей, а также описаны имеющиеся настройки и константы.

## Модули с национальными настройками

Если внимательно исследовать содержимое дистрибутива AngularJS, можно заметить в нем папку `i18n`. Внутри этой папки находятся файлы, имена которых следуют шаблону: `angular-locale_[locale name].js`, где часть имени `[locale name]` определяет локаль (locale) в виде комбинации кода языка и кода страны. Например, файл с настройками французского языка для Канады имеет имя `angular-locale_fr-ca.js`.

В состав дистрибутива AngularJS входит более 280 файлов с национальными настройками и константами для поддержки разных языков и стран. Хотя эти файлы и входят в состав дистрибутива, они не являются частью проекта AngularJS и не поддерживаются разработчиками фреймворка. Эти файлы регулярно извлекаются из библиотеки closure (http://closure-library.googlecode.com/).

По умолчанию фреймворк AngularJS использует настройки i18n для английского языка США (`en-us`). Если вы пишете приложение для использования на другом языке и в другой стране (или желаете предоставить пользователям возможность самим выбирать национальные настройки), вам следует подключить соответствующий файл и объявить зависимость от модуля с требуемыми национальными настройками. Например, чтобы настроить приложение для использования франкоговорящими канадцами (`fr-ca`), можно организовать подключение сценариев к странице, как показано ниже:

```
<!doctype html>
<html ng-app="locale">
<head>
<meta charset="utf-8">
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-locale_fr-ca.js"></script>
<script src="locale.js"></script>
</head>
```

```
<body ng-controller="LocaleCtrl">
...
</body>
```

где файл `locale.js` должен содержать определение модуля с зависимостью от модуля `ngLocale`:

```
angular.module('locale', ['ngLocale'])
```

## Использование доступных национальных настроек

Все файлы с национальными настройками, входящие в состав дистрибутива AngularJS, содержат определение модуля `ngLocale`. Модуль `ngLocale` экспортирует только одну службу: `$locale`. В официальной документации описывается только один общедоступный элемент службы `$locale` – переменная `$locale.id`, которую можно использовать для получения локали, используемой в настоящий момент.

В действительности служба `$locale` экспортирует гораздо большее количество констант. Среди них имеются константы, определяющие формат отображения даты и времени (`$locale.DATETIME_FORMATS`), а так же чисел (`$locale.NUMBER_FORMATS`). Упомянутые константы являются объектами JavaScript с настройками, которые могут потребоваться для форматирования дат, времени, чисел и валют. Например, в `$locale.DATETIME_FORMATS.MONTH` можно обнаружить названия всех месяцев.

### Национальные настройки и фильтры AngularJS

Наличие доступа к национальным настройкам может пригодиться при создании собственных директив и фильтров. Кстати, во встроенных фильтрах AngularJS эти настройки уже используются.

### Фильтр date

Фильтр `date` преобразует даты в соответствии с указанным форматом. Целевой формат можно указать явно, независимым (например, `'mm/dd/yy hh:mm a'`) или зависимым от национальных настроек способом. Во втором случае вместо определения точного формата можно использовать предопределенные имена. AngularJS понимает следующие имена предопределенных форматов: `medium`, `short`, `fullDate`, `longDate`, `mediumDate`, `shortDate`, `mediumTime`, `shortTime`.

Рассмотрим в качестве примера выражение `{{now | date:'fullDate'}}` (где `now` инициализируется вызовом конструктора `new Date()`), возвращающее разные результаты для разных национальных настроек:

- Tuesday, April 9, 2013 – для локали `en-us` (английский язык, США);
- mardi 9 avril 2013 – для локали `fr-fr` (французский язык, Франция).

### Фильтр currency

Фильтр `currency` служит для форматирования денежных сумм в числовом выражении. По умолчанию он использует символ валюты из текущих национальных настроек. Например, выражение `{{100 | currency}}` будет возвращать разные результаты для разных национальных настроек:

- $100.00 – для локали `en-us` (английский язык, США);
- 100.00 € – для локали `fr-fr` (французский язык, Франция).

Поведение по умолчанию фильтра `currency` некоторым пользователям вашего веб-приложения может показаться ненормальным при отображении цены в некоторой валюте. Было бы довольно странно увидеть снижение цены со 100.00 € (евро) до 100.00 $ (долларов)[1], получившееся простым изменением локали.

Если только вы не используете какие-то фиксированные региональные настройки, мы рекомендуем всегда явно указывать символ валюты в фильтре `currency`:

```
{{100 | currency:'€'}}
```

Однако существуют и другие сложности, связанные с фильтром `currency`. Несмотря на то, что он позволяет явно указывать символ валюты, в нем отсутствует возможность определения местоположения этого символа или выбора символа, отделяющего целую и дробную части (эти параметры будут определяться текущими национальными настройками). Это означает, что предыдущий пример отформатирует число, как € 100.00, что несколько необычно для символа евро.

Если фильтр `currency` справляется со своей задачей в вашем приложении, тогда используйте его. Но в некоторых случаях он оказывается неприменим и вы должны быть готовы создать собственный фильтр.

---

1 Учитывая, что доллар дешевле евро. – *Прим. перев.*

### Фильтр number

Фильтр `number` действует в точности, как ожидается. Он форматирует числа, применяя разделители в соответствии с региональными настройками. Например, выражение:

```
{{1000.5 | number}}
```

вернет результат:

- 1,000.5 – для локали `en-us` (английский язык, США);
- 1000,5 – для локали `fr-fr` (французский язык, Франция).

# Поддержка переводов

Возможность форматирования дат и чисел в соответствии с региональными настройками – это лишь малая часть возможностей механизма локализации. Обычно, когда люди задумываются о локализации, первое что приходит им на ум – это перевод текста на разные языки.

В приложении на основе AngularJS существует как минимум два места, где можно найти текст для перевода на другие языки: шаблоны и строки в коде на JavaScript.

В оставшейся части этой главы будем полагать, что перевод строк уже выполнен и хранится в удобном для нас формате, например, JSON. Это мог бы быть объект, ключи в котором соответствуют логическим именам переведенных фрагментов (например, `crud.user.remove.success`), а значения – фактическим строкам текста на требуемом языке. Например, фрагмент JSON, содержащий перевод строк для локали `en-us`, мог бы выглядеть так:

```
{
    'crud.user.remove.success': 'A user was removed successfully.',
    'crud.user.remove.error': 'There was a problem removing a user.'
    ...
}
```

а соответствующий ему фрагмент для локали `ru-ru`, мог бы содержать следующие строки:

```
{
    'crud.user.remove.success': 'Учетная запись была успешно удалена.',
    'crud.user.remove.error': 'При удалении учетной записи возникла
ошибка.'
    ...
}
```

## Перевод строк в шаблонах AngularJS

Обычно большая часть строк для перевода находится в шаблонах AngularJS. Давайте рассмотрим простой пример «Hello, World!»:

```
<span>Hello, {{name}}!</span>
```

Чтобы этот шаблон можно было использовать с разными национальными настройками, необходим способ замены строки «Hello» ее переводом на выбранный в данный момент язык. Существует множество приемов, которые можно было бы использовать здесь и каждый из них имеет свои достоинства недостатки, о чем рассказывается в следующих разделах.

### С использованием фильтров

Предположим, что имеется следующая структура JSON с переведенными строками:

```
{
    'greetings.hello': 'Hello'
    ...
}
```

Мы могли бы определить фильтр (назовем его `i18n`), используемый, как показано ниже:

```
<span>{{'greetings.hello' | i18n}}, {{name}}!</span>
```

Написать простую версию фильтра `i18n` совсем несложно, и можно было бы начать со следующей простенькой заготовки:

```
angular.module('i18nfilter', ['i18nmessages'])
    .filter('i18n', function (i18nmessages) {
        return function (input) {
            if (!angular.isString(input)) {
                return input;
            }
                return i18nmessages[input] || '?'+input+'?';
            };
    });
```

Фильтр `i18` мог бы опираться на массив переведенных строк (`i18nmessages`). Сами сообщения можно было бы определить в виде значений в отдельном модуле, например:

```
angular.module('i18nmessages', [])
    .value('i18nmessages', {
        'greetings.hello': 'Hello'
    });
```

Представленный выше фильтр `i18n` очень прост и дает обширное поле для улучшений и расширений. Например, в него можно добавить загрузку переведенных строк посредством службы `$http` с последующим кешированием, переключение между национальными настройками «на лету», и так далее. Однако, несмотря на кажущуюся простоту фильтра `i18n`, мы могли бы потратить массу времени на его доработку, устраняя проблемы, связанные с производительностью.

Преобразовав текст «Hello» в фильтруемое выражение (`{{'greetings.hello' | i18n}}`), мы добавили еще одно выражение, требующее интерпретации фреймворком AngularJS. Как будет показано в главе 11, надежность веб-приложений на основе AngularJS обратно пропорциональна количеству выражений, находящихся под наблюдением фреймворка и требующих вычисления. Добавление нового выражения для каждой отдельной строки может привести к снижению производительности приложения до неприемлемого уровня.

Решение проблемы перевода строк на основе фильтров выглядит достаточно простым и гибким, однако оно влечет существенные отрицательные последствия для производительности. Накладные расходы могут оказаться незначительными для небольших страниц, с малым количеством строк, требующих перевода. Но большие страницы с множеством переводимых строк могут стать узким местом с точки зрения производительности.

### С использованием директив

Чтобы избавиться от проблем производительности, свойственных решению на основе фильтров, можно было бы обратить свои взоры на директивы. Представьте себе такой синтаксис:

```
<span><i18n key='greetings.hello'></i18n>, {{name}}!</span>
```

Используя директивы, можно было бы избавиться от выражений, находящихся под наблюдением AngularJS, и тем самым решить проблему производительности, свойственную решению на основе фильтра. Но беда в том, что директивы влекут за собой собственные проблемы.

Во-первых, синтаксис таких директив оказывается слишком избыточным и не очень наглядным. Здесь можно было бы поэкспериментировать с альтернативными вариантами (например, задействовать атрибуты директив), но в любом случае такой подход ухудшает читаемость шаблонов и усложняет их модификацию. Кроме того, сущест-

вует еще более серьезная проблема: директивы могут использоваться не везде. Представьте поле ввода с атрибутом `placeholder`:

```
<input ng-model='name' placeholder='Provide name here'>
```

Мы не сможем использовать решение на основе директивы, чтобы перевести строку "`Provide name here`". AngularJS просто не поддерживает возможность интерпретации директив в атрибутах HTML.

Как видите, решение на основе директив годится не на все случаи жизни, поэтому продолжим поиски более удачного решения.

### Перевод фрагментов на этапе сборки

Последний подход, который мы исследуем в этой главе, заключается в переносе процедуры перевода в систему сборки. Идея состоит в том, чтобы обработать все шаблоны и сгенерировать множество шаблонов для всех поддерживаемых языков, которые будут загружаться браузером. В этом случае шаблоны будут выглядеть для AngularJS как статические ресурсы, не требующие дополнительной обработки на стороне клиента.

Точная технология перевода шаблонов во время сборки приложения зависит от системы сборки. Так, при использовании `Grunt.js` можно было бы воспользоваться способностью этой системы создавать новые шаблоны. Рассмотрим следующий шаблон с именем `hello.tpl.html`:

```
<div>
  <h3>Hello, {{name}}!</h3>
  <input ng-model='name' placeholder='Provide name here'>
</div>
```

Мы могли бы преобразовать его в шаблон `Grunt.js`, чтобы обеспечить дополнительную его обработку на этапе сборки:

```
<div>
  <h3><%= greeting.hello %>, {{name}}!</h3>
  <input ng-model='name' placeholder='<%= input.name %>'>
</div>
```

Тогда, опираясь на список поддерживаемых языков и файлов перевода, система сборки `Grunt.js` могла бы создать переведенные шаблоны и сохранить их в папках с именами, соответствующими кодам регионов. Например:

```
/en-us/hello.tpl.html
/fr-ca/hello.tpl.html
/ru-ru/hello.tpl.html
```

Настройка перевода во время сборки может оказаться достаточно сложной задачей, но обычно ее приходится решать всего один раз, на самых начальных этапах развития проекта. Зато она имеет огромное преимущество: позволяет избежать любых проблем, связанных с производительностью и вызванных локализацией, и дает возможность переводить строки, находящиеся в любом месте внутри шаблона.

## Перевод строк в коде JavaScript

Большинство строк, требующих перевода, находится в шаблонах AngularJS, но иногда может возникать необходимость обрабатывать текст в коде JavaScript. Например, может понадобиться выводить локализованные сообщения об ошибках, предупреждения и так далее. Независимо от причин, мы всегда должны быть готовы организовать перевод строк в коде JavaScript.

В AngularJS отсутствуют встроенные механизмы, которые могли бы помочь в этом, поэтому нам придется закатать рукава и написать простую службу, которую можно было бы использовать как инструмент для перевода. Но, прежде чем писать код, рассмотрим пример использования такой службы. Допустим, что нам потребовалось подготовить к отображению текст сообщения об успешном удалении объекта из хранилища, об отсутствии искомого объекта в хранилище, и так далее.

В приложении, которое должно поддерживать несколько языков, мы не можем просто «зашить» текст сообщения в код JavaScript. Нам нужен некоторый механизм, позволяющий получить локализованное и параметризованное сообщение на основе ключа. Например, было бы желательно, чтобы вызов:

```
localizedMessages.get('crud.user.remove.success', {id: 1234})
```

возвращал для локали `en-us` сообщение, выглядящее так: `"A user with id '1234' was removed successfully."`, а для локали `ru-ru` так: `"Учетная запись пользователя с идентификатором '1234' была успешно удалена."`.

Написать службу, которая отыскивала бы сообщение по ключу и коду текущей локали, не составляет никакого труда. Единственная сложность – обработка параметров внутри локализованной строки. К счастью, в этом случае мы можем положиться на AngularJS и использовать его службу `$interpolate` – ту самую службу, которую сам фреймворк применяет для обработки директив интерполяции в шаблонах. Благодаря ей мы можем определять локализованные сообщения как:

```
"A user with id '{{id}}' was removed successfully."
```

Пример приложения SCRUM включает полную реализацию службы локализации, основанную на только что описанной идее. Реализация, фрагмент которой приводится ниже, очень проста:

```
angular.module('localizedMessages', [])
    .factory('localizedMessages', function ($interpolate, i18nmessages) {

        var handleNotFound = function (msg, msgKey) {
            return msg || '?' + msgKey + '?';
        };

        return {
            get : function (msgKey, interpolateParams) {
                var msg = i18nmessages[msgKey];
                if (msg) {
                    return $interpolate(msg)(interpolateParams);
                } else {
                    return handleNotFound(msg, msgKey);
                }
            }
        };
    });
```

# Шаблоны проектирования, советы и рекомендации

Последняя часть этой главы посвящена вопросам применения некоторых шаблонов проектирования для интернационализации и локализации приложений. Сначала мы познакомимся со способами инициализации приложения и переключения между национальными настройками. Затем мы перейдем к изучению приемов, которые можно применять внутри выполняющихся приложений: переопределение форматов представления данных и обработка ввода пользователя в соответствии с выбранными национальными настройками.

## *Инициализация приложений с учетом выбранных национальных настроек*

Как мы узнали в начале главы, в состав дистрибутива AngularJS входят файлы с определениями национальных настроек. Для каждой локали имеется отдельный файл, содержащий определение модуля `ngLocale`. Если в приложении потребуется использовать нацио-

нальные настройки, необходимо объявить зависимость от модуля `ngLocale`. В качестве напоминания, ниже приводится один из способов настройки приложения на использование определенной локали:

```
<head>
<meta charset="utf-8">
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-locale_fr-ca.js"></script>
<script src="locale.js"></script>
</head>
```

Он прекрасно подходит для случаев, когда приложение должно работать только с одной, фиксированной локалью. Однако в действительности часто бывает желательно инициализировать национальные настройки в соответствии с предпочтениями пользователя. Ниже приводятся различные источники информации, откуда можно узнать об этих предпочтениях:

- настройки браузера;
- заголовки HTTP-запросов (например, `Accept-Language`);
- адрес URL или параметры в строке запроса;
- настройки сервера (параметры учетной записи пользователя, географическое местоположение сервера, и так далее).

Взглянув на этот список, можно смело сказать, что определение национальных настроек уместнее производить на стороне сервера. Именно поэтому мы рекомендуем выполнять обработку начальной страницы приложения на сервере.

Для практической демонстрации процедуры выбора локали, рассмотрим стратегию, реализованную в примере приложения SCRUM. Она заключается в определении целевой локали, исходя из:

- наличия названия локали в адресе URL начальной страницы приложения;
- наличия заголовка `Accept-Language` в запросе;
- поддерживаемых национальных настроек.

Согласно этой стратегии пользователь может использовать для доступа к приложению следующие адреса URL.

- `http://host.com/fr-ca/admin/users/list`: здесь локаль (`fr-ca`) указана явно. Нам остается только проверить, поддерживается ли она приложением, и переадресовать пользователя на другой адрес URL, если запрошенная локаль не поддерживается. Например, мы могли бы переадресовать пользователя по адресу с локалью по умолчанию (такой как,

en-us), если локаль fr-ca не поддерживается: http://host.com/en-us/admin/users/list.

- http://host.com/admin/users/list: здесь локаль не указана. Однако мы можем попытаться определить ее по заголовку Accept-Language HTTP-запроса и переадресовать пользователя по адресу URL с требуемой локалью. Безусловно, в этой ситуации мы так же должны убедиться, что локаль в заголовке запроса, поддерживается приложением.

На практике часто используются более сложные алгоритмы определения настроек из разных источников с более надежной стратегией возврата к настройкам по умолчанию. Конкретная стратегия определения национальных настроек в значительной степени зависит от требований, предъявляемых приложением.

После определения национальных настроек можно отправить браузеру соответствующую начальную страницу приложения. Начальная страница (index.html или с подобным именем) должна создаваться на стороне сервера динамически. В примере приложения SCRUM файл index.html хранит следующий шаблон:

```
<head>
<meta charset="utf-8">
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-locale_<%= locale %>.js"></script>
...
</head>
```

Здесь значение локали locale определяется в соответствии с описанным алгоритмом, и перед отправкой браузеру шаблон обрабатывается на стороне сервера.

### Следствия включения локали в адрес URL

После добавления нового элемента пути в URL приложения необходимо будет позаботиться о решении двух новых проблем: перенастройка маршрутов и загрузка шаблонов.

Обычно новый элемент пути должен обрабатываться как часть определения маршрута. Конечно, можно вернуться к исходным текстам приложения и переопределить все маршруты. Например, маршрут /admin/users/list можно преобразовать в /:locale/admin/users/list.

Однако такой подход, заключающийся в добавлении префикса /:locale во все маршруты, достаточно утомителен и чреват ошиб-

ками. Вместо этого можно просто определить тег `base` в начальной странице приложения и указать в нем путь к требуемой папке:

```
<head>
<meta charset="utf-8">
<script src="/lib/angular/angular.js"></script>
<script src="/lib/angular/angular-locale_<%= locale %>.js"></script>
<base href="/<%= locale %>/">
...
</head>
```

Служба `$location` в AngularJS (и, соответственно, система маршрутизации) распознает тег `<base>` и откладывает все маршруты относительно пути, указанного в атрибуте `href`.

Использование тега `base`, ссылающегося на папку с локализованными файлами, дает еще одно преимущество – благодаря ему можно использовать относительные адреса URL для загрузки шаблонов AngularJS. Если будет выбран способ локализации во время сборки приложения, в результате будет сгенерирован множество файлов шаблонов в папках с именами, соответствующими названиям локалей. Правильно сформированный тег `<base>` обеспечит загрузку шаблонов для выбранной локали.

Использование тега `<base>` означает, что для загрузки любых ресурсов, не зависящих от текущих национальных настроек, должны указываться абсолютные адреса URL.

В реальных сценариях развертывания, шаблоны, скорее всего, будут загружаться заранее, на начальном этапе запуска приложения, а не динамически, поэтому с точки зрения шаблонов определение тега `<base>` имеет меньшую практическую ценность. Более подробно о различных приемах развертывания веб-приложений на основе AngularJS рассказывается в главе 12.

## Переключение между национальными настройками

В текущей версии AngularJS все необходимые приложению модули должны быть перечислены как зависимости главного модуля до его запуска. Модуль `ngLocale` не является исключением из этого правила, а это означает, что он должен быть загружен браузером и объявлен как зависимость до того, как приложение будет запущено. Как следствие, модуль с национальными настройками должен быть выбран до

запуска приложения. Кроме того, выбранный модуль нельзя заместить другим без полной инициализации приложения.

В текущей версии AngularJS национальные настройки (локаль) должны быть выбраны заранее, до инициализации приложения. Выбранную локаль нельзя изменить динамически, так как для этого требуется повторно инициализировать приложение с другим модулем `ngLocale`.

Учитывая особенности работы системы модулей в текущей версии AngularJS, переключение национальных настроек лучше всего реализовать, как перезапуск приложения, переадресовав пользователя по адресу URL с новым значением локали. Это означает, что текущее состояние приложения в браузере будет полностью утрачено. К счастью, механизм глубокого связывания в AngularJS автоматически перенесет пользователя в то же место приложения после изменения национальных настроек.

Чтобы обеспечить возможность переключения локали путем переадресации пользователя, необходимо сначала подготовить новую строку URL. Это легко можно сделать с помощью методов службы `$location`. Например, если предположить, что в настоящий момент используется URL `http://host.com/en-us/admin/users/list` и требуется перейти по адресу `http://host.com/fr-ca/admin/users/list`, мы могли бы написать следующую функцию `switchLocaleUrl`:

```
$scope.switchLocaleUrl = function(targetLocale) {
    return '/'+targetLocale+'/'+$location.url();
};
```

Функция `switchLocaleUrl` вычислит адрес URL, эквивалентный текущему, но с другим значением локали, указанным в `targetLocale`. Эту функцию можно использовать в стандартном теге `<a>`, чтобы дать пользователям возможность выбрать другие национальные настройки:

```
<a ng-href="switchLocaleUrl('fr-ca')" target="_self">Fransais</a>
```

## Нестандартное форматирование дат, чисел и валют

Некоторые фильтры AngularJS способны обрабатывать данные с учетом текущих национальных настроек. Как было показано в начале этой главы, фильтр `date` распознает именованные форматы, зависящие от

национальных настроек, например, `{{now | date:'fullDate'}}`. Точное отображение строки `fullDate` в фактический формат определяется в файле с национальными настройками как ключ `$locale.DATETIME_FORMATS.fullDate`. Например, формат `fullDate` для локали `fr-ca` отображается в строку формата `EEEE d MMMM y`.

Форматы по умолчанию обычно в точности соответствуют требованиям для данного языка и страны, но иногда может потребоваться немного скорректировать некоторые форматы. Этого легко можно добиться, создав декоратор для фильтра.

В AngularJS можно создать декоратор для любой существующей службы или фильтра. Таким декоратором можно обернуть любую существующую службу и «декорировать» (дополнить) ее новыми функциональными возможностями. Декораторы AngularJS являются ярким примером применения шаблона проектирования (http://ru.wikipedia.org/wiki/Декоратор_(шаблон_проектирования)).

Например, чтобы изменить формат `fullDate` для локали `fr-ca`, можно написать обертку вокруг фильтра `date`:

```
angular.module('filterCustomization', [])
    .config(function ($provide) {
        var customFormats = {
            'fr-ca': {
                'fullDate': 'y'
            }
        };

        $provide.decorator('dateFilter', function ($delegate, $locale) {
            return function (input, format) {
                return $delegate(input, customFormats[$locale.
id][format]
                    || format);
            };
        });
    })
```

В данной реализации декоратора используются возможности системы внедрения зависимостей AngularJS. Определяя новый декоратор (`$provide.decorator()`) мы можем завернуть в него существующую службу, сохранив доступ к оригинальной службе (`$delegate`). Чтение остального кода в реализации новой службы не должно вызывать у вас затруднений. Мы просто проверяем хеш с переопределенными значениями форматов для указанной локали (`customFormats`), и если требуемый формат найден, мы используем его в вызове ори-

гинального фильтра `data`. В противном случае используем формат, переданный в оригинальный вызов фильтра `date`.

Наш декоратор заменит фильтр `date`, потому что мы регистрируем его под тем же именем. Преимущество такого подхода в том, что он не требует изменять программный код приложения, чтобы заставить его использовать нестандартную версию фильтра.

Мы могли бы также предусмотреть в этом фильтре выбор формата по умолчанию, если он не был указан явно.

Другое, более грубое решение, заключается в простом редактировании файла с национальными настройками для данной локали. Недостаток этого подхода заключается в том, что он требует помнить о необходимости внести соответствующие изменения при обновлении версии AngularJS.

# В заключение

В этой главе мы коснулись проблем интернационализации и локализации, характерных для приложений на основе AngularJS. Мы узнали, что AngularJS предоставляет модуль `ngLocale`, где хранятся национальные форматы для отображения дат, валют и чисел. Эти настройки взяты из библиотеки closure и определяют поведение некоторых встроенных фильтров.

Большая часть усилий по интернационализации приложений связана с переводом существующих шаблонов и строк в коде JavaScript. Мы исследовали различные способы обслуживания шаблонов, переведенных на разные языки: с помощью фильтров, директив и решений, основанных на системах сборки приложений. Подход на основе фильтров выглядит достаточно простым и удобным, но он влечет за собой значительные накладные расходы, отрицательно сказывающиеся на производительности приложения, которые во многих проектах могут оказаться неприемлемыми. Подход на основе директив решает проблему потери производительности, но он не обладает достаточной гибкостью, чтобы его можно было использовать на практике. Именно по этим причинам мы исследовали третий, рекомендованный подход, основанный на подготовке локализованных шаблонов во время сборки приложения, когда перевод строк выполняется до того, как шаблоны будут загружены браузером.

В заключение этой главы мы познакомились с некоторыми шаблонами проектирования, имеющими отношение к интернационали-

зации и локализации. Мы начали с обзора вариантов инициализации и переключения национальных настроек. Текущая версия фреймворка AngularJS требует, чтобы национальные настройки выбирались до запуска приложения, и именно поэтому мы сосредоточились на определении и инициализации локали на стороне сервера, описав прием динамического создания начальной страницы приложения. Переключение национальных настроек «на лету» невозможно реализовать в текущей версии AngularJS, поэтому было предложено решение, наиболее удачное в этой ситуации, заключающееся в переадресации пользователя на начальную страницу для выбранной локали.

Другие шаблоны проектирования, описанные в этой главе, имеют отношение к настройке форматов отображения дат, чисел и валют. Фреймворк AngularJS предоставляет неплохие варианты по умолчанию, но иногда может понадобиться немного изменить их. Отличным способом изменить формат представления данных является создание декораторов для существующих фильтров.

К настоящему моменту у нас имеется полнофункциональное, интернационализированное веб-приложение на основе AngularJS. В следующей главе мы сосредоточимся на изучении приемов повышения надежности приложений. В частности, мы будем учиться определять и решать потенциальные проблемы, связанные с производительностью.

# ГЛАВА 11. Создание надежных веб-приложений на основе AngularJS

Производительность веб-приложений является одним из нефункциональных требований, которые приходится совмещать с другими задачами и функциональными требованиями. Очевидно, что мы не можем позволить себе пренебречь производительностью. Даже если приложение имеет умопомрачительный интерфейс и обладает всеми необходимыми функциональными возможностями, люди могут отказываться использовать его, если в работе будут наблюдаться задержки.

Существует множество факторов, влияющих на производительность законченного веб-приложения: интенсивность использования сети, размер дерева DOM, количество и сложность правил CSS, алгоритмы, реализованные на JavaScript, размеры и количество используемых структур данных, и многие другие. Не говоря уже о версии браузера и психологии пользователя! Некоторые проблемы производительности носят универсальный характер и приемы их решения не зависят от используемых технологий. Другие характерны только для AngularJS и в этой главе мы остановимся именно на них.

Чтобы лучше понимать характеристики производительности фреймворка AngularJS, необходимо разбираться в его внутреннем устройстве. Именно поэтому данная глава начинается с глубокого погружения в ядро AngularJS. Далее приводится обзор шаблонов проектирования, нацеленных на повышение производительности, с попутным обсуждением вариантов и компромиссов.

После прочтения этой главы вы будете.

- знать, как работает механизм отображения AngularJS и получите представление о внутренней организации фреймворка,

что очень важно для понимания его характеристик производительности;
- понимать теоретические ограничения производительности приложений на основе AngularJS и сможете быстро определять, вписывается ли ваш проект в установленные ограничения;
- способны выявлять и ликвидировать узкие места в приложениях на основе AngularJS, связанные с повышенным использованием процессора и потреблением памяти;
- подготовлены к определению ситуаций, когда возможно применение шаблонов проектирования, способствующих повышению производительности, еще до того, как будет написан программный код, а также к использованию инструментов профилирования производительности для решения проблем производительности в уже существующем коде;
- понимать причины отрицательного влияния директивы `ng-repeat` на производительность при работе с большими наборами данных.

# Внутренние механизмы AngularJS

Чтобы лучше понимать характеристики производительности приложений на основе AngularJS, необходимо заглянуть под капот фреймворка. Знакомство с особенностями работы внутренних механизмов AngularJS позволит легко идентифицировать ситуации и шаблоны программирования, оказывающие наибольшее влияние на общую производительность приложения.

## *Это не механизм строковых шаблонов*

Увидев короткие и простые примеры кода, можно посчитать, что AngularJS – это еще один клиентский механизм поддержки шаблонов. И действительно, следующий фрагмент кода AngularJS:

```
Hello, {{name}}!
```

невозможно отличить от кода обычной системы шаблонов, такой как Mustache (http://mustache.github.io/). Различия становятся очевидными, только после добавления директивы `ng-model`:

```
<input ng-model="name">
Hello, {{name}}!
```

После добавления этой директивы дерево DOM будет обновляться автоматически, в ответ на ввод пользователя, без дополнительного вмешательства со стороны разработчика. Первое время поддержка двунаправленного связывания может казаться волшебством. Но это не так, AngularJS использует только проверенные алгоритмы, чтобы вдохнуть жизнь в дерево DOM! В последующих разделах мы разберем эти алгоритмы и исследуем, как изменения в дереве DOM попадают в модель и как изменения в модели приводят к обновлению дерева DOM.

### Обновление модели в ответ на события DOM

Изменения в дереве DOM переносятся фреймворком AngularJS в модель с помощью обработчиков событий DOM, регистрируемых различными директивами. Обработчик события изменяет модель, присваивая новые значения переменным в контексте `$scope`.

Ниже представлен упрощенный эквивалент директивы `ng-model` (с именем `simple-model`), иллюстрирующий основные механизмы, вовлеченные в обновление модели:

```
angular.module('internals', [])
    .directive('simpleModel', function ($parse) {
        return function (scope, element, attrs) {

            var modelGetter = $parse(attrs.simpleModel);
            var modelSetter = modelGetter.assign;

            element.bind('input', function(){
                var value = element.val();
                modelSetter(scope, value);
            });
        };
    });
```

Важнейшей частью директивы `simple-model` является обработчик события DOM `input`, извлекающий новое значение из элемента ввода и сохраняющий его в модели.

Сохранение фактического значения в модели выполняется с помощью службы `$parse`. Эта служба может использоваться и для вычисления выражения AngularJS, с подстановкой значений из модели, и для сохранения значения модели в контексте. Когда служба `$parse` вызывается с выражением в качестве аргумента, она возвращает функцию чтения (getter). Эта функция может иметь свойство `assign`

(функция записи – setter), если выражение AngularJS допускает возможность присваивания.

### Передача изменений из модели в DOM

С помощью службы `$parse` можно так же реализовать упрощенную версию директивы `ng-bind`, способную отображать значения модели в текстовые узлы DOM:

```
.directive('simpleBind', function ($parse) {
    return function (scope, element, attrs) {

        var modelGetter = $parse(attrs.simpleBind);
        element.text(modelGetter(scope));
    }
});
```

Директива `simple-bind`, представленная выше, принимает выражение (передаваемое ей в виде атрибута DOM), вычисляет его с учетом контекста `$scope`, и в соответствии с полученным результатом изменяет текстовое содержимое указанного элемента DOM.

### Синхронизация дерева DOM и модели

Теперь можно опробовать обе директивы в разметке HTML, как показано в следующем фрагменте, в ожидании, что они действительно будут работать как упрощенные версии директив `ng-model` и `ng-bind`:

```
<div ng-init='name = "World"'>
  <input simple-model='name'>
  <span simple-bind='name'></span>
</div>
```

К сожалению, если запустить этот пример, мы не получим ожидаемого результата! Первый раз страница отобразится без ошибок, но ввод символов в элемент `<input>` не будет вызывать изменение содержимого элемента `<span>`.

Очевидно, что мы что-то упустили. Если вернуться к директиве `simple-bind`, можно заметить, что в ней предусмотрено только первоначальное отображение значения из модели. Директива не следит за изменениями в модели и не реагирует на них. Эту проблему легко исправить, добавив вызов метода `$watch` контекста:

```
.directive('simpleBind', function ($parse) {
    return function (scope, element, attrs) {

        var modelGetter = $parse(attrs.simpleBind);
```

```
            scope.$watch(modelGetter, function(newVal, oldVal){
                element.text(modelGetter(scope));
            });
        }
    });
```

Метод `$watch` позволяет организовать мониторинг изменений в модели и вызывать функции в ответ на эти изменения. Сигнатуру этого метода в упрощенном виде можно было бы представить, как показано ниже:

```
scope.$watch(watchExpression, modelChangeCallback)
```

Параметр `watchExpression` может быть функцией или выражением AngularJS (указывающим на значение в модели, за которым нужно следить). Параметр `modelChangeCallback` – это функция обратного вызова, вызываемая при каждом изменении значения `watchExpression`. Функция обратного вызова имеет доступ к обоим значениям `watchExpression`, новому и старому.

После знакомства с механизмом `$watch` можно заметить, что наша директива `simple-model` тоже могла бы извлечь выгоду из слежения за моделью и обновлять значение в элементе ввода при изменении значения в модели:

```
.directive('simpleModel', function ($parse) {
    return function (scope, element, attrs) {

        var modelGetter = $parse(attrs.simpleModel);
        var modelSetter = modelGetter.assign;

        // Передача изменений модель -> DOM
        scope.$watch(modelGetter, function(newVal, oldVal){
            element.val(newVal);
        });

        // Передача изменений DOM -> модель
        element.bind('input', function () {
            modelSetter(scope, element.val());
        });
    };
})
```

Последние изменения в коде наших простых директив реализуют слежение за значением в модели. Вроде бы все выглядит логично и, казалось бы, вполне можно ожидать, что теперь-то все заработает как надо. Увы, запустив пример вы обнаружите, что директивы все еще не работают.

Как оказывается, мы упустили из виду еще одну важную деталь: *когда* и *как* AngularJS проверяет наличие изменений в модели. Нам нужно разобраться, при каких условиях AngularJS начнет вычислять все выражения, находящиеся под наблюдением, проверяя наличие изменений в модели.

### scope.$apply – ключ в мир AngularJS

До выхода первой общедоступной версии AngularJS, ходило множество домыслов, касающихся алгоритма слежения за изменениями в модели. Наиболее распространенным было заблуждение, что для этой цели используется некоторый механизм опроса. Этот механизм, как предполагалось, запускается через определенные интервалы времени, проверяет наличие изменений в модели и при их обнаружении инициирует отображение дерева DOM. В действительности все совсем не так.

Фреймворк не использует никаких механизмов опроса для периодической проверки модели.

Идея механизма отслеживания изменений в модели основана на наблюдении, что так или иначе, существует конечное (и небольшое) количество ситуаций, могущих привести к изменению модели. В их числе:

- события DOM (пользователь изменяет значение в поле ввода, щелкает на кнопке, вызывающей функцию JavaScript, и так далее);
- обратные вызовы, производимые объектом XHR при получении ответов от сервера;
- изменение адресной строки в браузере;
- срабатывание таймера (`setTimout`, `setInterval`).

Фактически, если не произошло ни одно из перечисленных событий (пользователь не взаимодействует со страницей, не завершился какой-нибудь вызов XHR, не сработал таймер), нет никакой необходимости проверять изменения в моделях. Когда в странице ничего не происходит, модель просто не может измениться и поэтому нет причин повторно отображать дерево DOM.

Механизм мониторинга моделей в AngularJS запускается, только если это будет затребовано явно. Чтобы привести этот сложный механизм в действие, необходимо вызвать метод `$apply` объекта контекста.

Вернемся к нашему примеру реализации упрощенных директив. Мы могли бы вызывать упомянутый выше метод при каждом изменении значения в поле ввода (чтобы обеспечить передачу изменений в модели при каждом нажатии на клавиши – именно так действует директива `ng-model` по умолчанию):

```
.directive('simpleModel', function ($parse) {
    return function (scope, element, attrs) {

        var modelGetter = $parse(attrs.simpleModel);
        var modelSetter = modelGetter.assign;

        // Передача изменений модель -> DOM
        scope.$watch(modelGetter, function(newVal, oldVal){
            element.val(newVal);
        });

        // Передача изменений DOM -> модель
        element.bind('input', function () {
            scope.$apply(function () {
                modelSetter(scope, element.val());
            });
        });
    };
})
```

Можно было бы изменить стратегию по умолчанию и запускать механизм обновления модели только после того, как поле ввода потеряет фокус:

```
// Передача изменений DOM -> модель
element.bind('blur', function () {
    scope.$apply(function () {
        modelSetter(scope, element.val());
    });
});
```

Неважно, какая стратегия будет выбрана, важно, что процедура проверки изменений в модели должна запускаться явно. Для кого-то это может оказаться неожиданностью, так как при обычном использовании встроенных директив AngularJS «волшебство» происходит как бы само по себе, без вызова метода `$apply` с нашей стороны. В действительности эти вызовы выполняются... в коде встроенных директив. Стандартные директивы и службы (`$http`, `$timeout`, `$location` и другие) сами заботятся о запуске механизма проверки изменений в моделях.

Механизм проверки наличия изменений в моделях запускается вызовом метода `$apply` объекта контекста. В стандартных службах и директивах этот вызов выполняется в ответ на события сети, DOM, таймеров JavaScript или изменения адресной строки браузера.

## Цикл обработки событий $digest

В терминологии AngularJS процесс проверки изменений в модели называется циклом обработки событий `$digest` (или просто цикл `$digest`). Название происходит от имени метода `$digest`, доступного в объектах контекста `Scope`. Этот метод вызывается внутри метода `$apply` и вычисляет все выражения, находящиеся под наблюдением, зарегистрированные во всех контекстах.

Но какие задачи решает цикл `$digest` в AngularJS? И как он определяет, что модель изменилась? Цикл `$digest` в AngularJS предназначен для решения двух взаимосвязанных задач.

- Определить, какие части модели изменились, и какие свойства DOM следует обновить в ответ на эти изменения. Главная цель состоит в том, чтобы максимально упростить жизнь разработчиков. Нам достаточно просто изменить свойства модели, а директивы AngularJS автоматически определят, какие части документа следует обновить.
- Устранить излишние циклы обновления документа, чтобы увеличить производительность и избежать эффекта мерцания пользовательского интерфейса. В AngularJS это достигается за счет откладывания фактического обновления DOM на максимально долгий срок, пока модель не стабилизируется (то есть, когда все значения в модели будут вычислены и готовы к отображению в пользовательском интерфейсе).

Чтобы понять, как AngularJS достигает своих целей, необходимо помнить, что браузеры имеют единственный поток выполнения, обслуживающий пользовательский интерфейс. Да, в браузерах существуют и другие потоки выполнения (например, отвечающие за выполнение сетевых операций), но только один поток может отображать на экране элементы DOM, обрабатывать события DOM и выполнять код JavaScript. Браузеры постоянно переключаются между контекстом выполнения JavaScript и контекстом отображения DOM.

Прежде всего AngularJS убеждается, что все значения в модели вычислены и «стабильны», и только потом возвращает управление контексту отображения дерева DOM. При таком подходе обновление

пользовательского интерфейса выполняется одним махом, а не как серия мелких обновлений в ответ на изменение отдельных значений в модели. Это повышает скорость выполнения (за счет уменьшения количества переключений между контекстами выполнения) и уменьшает количество побочных визуальных эффектов (так как все необходимые изменения отображаются на экране за один присест). Если бы изображение на экране обновлялось в ответ на изменение каждого отдельного свойства модели, мы получили бы медленный и мерцающий пользовательский интерфейс.

### Анатомия $watch

Для определения изменений фреймворк AngularJS использует простое сравнение состояний объектов (dirty checking). То есть, он сравнивает ранее сохраненную копию модели с новой, полученной после обработки одного из событий, которые могут вызвать изменение модели (события DOM, события XHR, и так далее).

Вспомним, как выглядит синтаксис регистрации нового выражения, требующего наблюдения за ним:

```
$scope.$watch(watchExpression, modelChangeCallback)
```

Когда новое выражение добавляется в контекст, AngularJS вычислит выражение `watchExpression` и сохранит результат. После входа в цикл `$digest` выражение `watchExpression` будет вычислено снова и новое значение будет сопоставлено с сохраненным. Если новое и сохраненное значение отличаются, будет вызвана функция `modelChangeCallback`. Затем новое значение будет сохранено для последующих сравнений и процесс повторится.

Как разработчики, мы будем хорошо осведомлены о выражениях, зарегистрированных вручную (в контроллерах приложений или в собственных директивах). Но мы должны помнить, что любая директива (встроенная в ядро AngularJS или входящая в стороннюю библиотеку) может определять собственные выражения. Любое выражение интерполяции (`{{expression}}`) также регистрирует себя в механизме наблюдения.

### Стабильность модели

Модель считается достигшей стабильного состояния (и готовой к отображению), если ни одно из выражений не вернуло новый результат. Достаточно обнаружить изменение в единственном выражении, чтобы вынудить AngularJS пометить весь цикл `$digest` как «гряз-

ный» и запустить новую итерацию. Здесь действует принцип «одна паршивая овца все стадо испортит», и это вполне оправданно.

AngularJS продолжает выполнять цикл `$digest` и повторно вычислять все выражения во всех контекстах, пока обнаруживается хотя бы одно изменение. Возможно потребуется выполнить несколько циклов `$digest`, так как функции обратного вызова могут производить побочные эффекты. Любой обратный вызов, произведенный в результате изменения модели, может изменить значение, уже проверенное в этом цикле и признанное стабильным.

Рассмотрим пример простой формы с двумя полями ввода даты: `startDate` и `endDate`. В этой форме значение поля `endDate` всегда должно быть больше значения поля `startDate`:

```
<div>
<form>
  Start date: <input ng-model="startDate">
  End date: <input ng-model="endDate">
</form>
</div>
```

Чтобы гарантировать, что значение `endDate` не окажется меньше значений `startDate`, необходимо зарегистрировать выражение:

```
function oneDayAhead(dateToIncrement) {
    return dateToIncrement.setDate(dateToIncrement.getDate() + 1);
};
$scope.$watch('startDate', function (newValue) {
    if (newValue <= $scope.starDate) {
        $scope.endDate = oneDayAhead($scope.starDate);
    }
});
```

Выражение, зарегистрированное в контроллере, делает два значения в модели взаимозависимыми, когда изменение одной переменной вызывает изменение другой. Функция обратного вызова, выполняемая в ответ на изменение модели, может иметь побочный эффект в виде изменения значения, которое уже считается «стабильным».

Разобравшись с алгоритмом простого сравнения состояний объектов, вы легко сможете понять, почему данное выражение `watchExpression` вычисляется как минимум дважды в каждом цикле `$digest`. Убедиться в этом можно с помощью следующей разметки:

```
<input ng-model='name'>
{{getName()}}
```

и выражения `getName()`, объявленного как функция:

```
$scope.getName = function() {
    console.log('dirty-checking');
    return $scope.name;
}
```

Если запустить этот пример и понаблюдать за выводом в консоли, можно увидеть, что каждому изменению в поле `<input>` соответствует две строки в консоли.

Любой цикл `$digest` выполнит как минимум одну итерацию, обычно две. Это означает, что каждое отдельно взятое выражение, находящееся под наблюдением, будет вычислено дважды в каждом цикле `$digest` (прежде чем браузер оставит контекст выполнения кода JavaScript и перейдет к отображению пользовательского интерфейса).

### Нестабильные модели

Иногда возникают ситуации, когда двух итераций цикла `$digest` оказывается недостаточно, чтобы стабилизировать модель. Хуже того, возможна даже ситуация, когда модель никогда не стабилизируется! Рассмотрим очень простой пример разметки:

```
<span>Random value: {{random()}}</span>
```

где функция `random()` определена в контексте, как:

```
$scope.random = Math.random;
```

Здесь выражение эквивалентно функции `Math.random()` и в каждой итерации цикла `$digest` будет (практически всегда) возвращать разные значения. Это означает, что каждая итерация будет отмечаться как «грязная» и вызывать необходимость выполнить еще одну итерацию. Эта ситуация будет повторяться много раз, снова и снова, пока AngularJS не решит, что модель в принципе нестабильна и не прервет цикл `$digest`.

По умолчанию AngularJS будет пытаться выполнить до 10 итераций, прежде чем объявить модель нестабильной и прервать цикл `$digest`.

После прерывания цикла `$digest` AngularJS сообщит об ошибке (с помощью службы `$exceptionHandler`, которая по умолчанию выводит сообщения об ошибках в консоль). Сообщение об ошибке будет содержать информацию о последних пяти нестабильных выражени-

ях (с их новыми и старыми значениями). В большинстве случаев существует только одно нестабильное выражение, что упрощает поиск причин.

После того, как выполнение цикла `$digest` будет прервано, поток выполнения JavaScript покинет «мир AngularJS» и уже ничто не помешает браузеру перейти к контексту отображения. В этом случае пользователь увидит на странице значения, вычисленные в последней итерации цикла `$digest`.

Страница будет отображена на экране, даже если будет превышен предел 10 итераций на цикл `$digest`. Ошибку сложно будет обнаружить, если не заглянуть в консоль, поэтому она долго может оставаться незамеченной. Тем не менее, мы всегда должны помнить о проблеме нестабильных моделей и стараться решить их.

### Цикл $digest и иерархия контекстов

В каждой итерации цикла `$digest` повторно вычисляются все выражения во всех контекстах, начиная с корневого контекста `$rootScope`. На первый взгляд может показаться, что достаточно было бы повторно вычислить только выражения в контексте, где обнаружены изменения и во всех его дочерних контекстах. К сожалению, это может привести к рассинхронизации пользовательского интерфейса и модели. Причина в том, что изменения в одном из дочерних контекстов могут привести к изменениям в родительском контексте. Ниже приводится простой пример приложения с двумя контекстами (один из них – корневой контекст `$rootScope`, а другой – созданный директивой `ng-controller`):

```
<body ng-app ng-init='user = {name: "Superhero"}'>
  Name in parent: {{user.name}}
  <div ng-controller="ChildCtrl">
    Name in child: {{user.name}}
    <input ng-model='user.name'>
  </div>
</body>
```

Изменение модели (`user.name`) выполняется в дочернем контексте (созданном директивой `ng-controller`), но фактически изменяется свойство объекта, объявленное в `$rootScope`.

Подобные конфигурации вынуждают AngularJS вычислять все выражения, находящиеся под наблюдением, начиная с корневого контекста `$rootScope` и далее вниз по дереву дочерних контекстов

(с использованием алгоритма обхода в глубину). Если бы AngularJS вычислял только выражения в контексте, где обнаружены изменения (плюс в его дочерних контекстах) возник бы риск рассинхронизации значений в модели с отображаемыми значениями на экране. В примере, обсуждаемом здесь, выражение интерполяции `Name in parent: {{user.name}}` не вычислялось бы и отображалось неправильно.

В каждой итерации цикла `$digest` фреймворк AngularJS должен вычислить все выражения во всех контекстах, начиная с корневого контекста `$rootScope` и далее во всех дочерних контекстах.

## Соединяем все вместе

Давайте подытожим все, что узнали о внутреннем устройстве AngularJS, исследовав простой пример поля ввода, передающего свои изменения в дерево DOM:

```
<input ng-model='name'>
{{name}}
```

Этот код зарегистрирует в контексте два выражения для наблюдения за ними, и каждое из них будет наблюдать за значением переменной `name` в модели. После начального отображения страницы, в ней не происходит никаких событий и браузер будет простаивать в цикле ожидания. Только когда пользователь начнет ввод в поле `<input>`, весь механизм придет в действие.

- Будет сгенерировано событие ввода. Браузер войдет в контекст выполнения JavaScript.
- Обработчик событий DOM, зарегистрированный директивой `input`, получит управление. Он обновит значение модели и вызовет метод `scope.$apply` контекста.
- Контекст выполнения JavaScript войдет в «мир AngularJS» и запустится цикл `$digest`. В первой итерации цикла `$digest` выражение (зарегистрированное директивой интерполяции `{{name}}`) будет помечено как «грязное», что вызовет необходимость выполнить вторую итерацию.
- В результате обнаружения изменений в модели метод `$watch` обратится к функции обратного вызова. Она обновит свойство `text` элемента DOM, в котором находится выражение интерполяции.
- Во второй итерации цикла `$digest` повторно будут вычислены все выражения, находящиеся под наблюдением. На этот

раз никаких изменений обнаружено не будет, AngularJS пометит модель как «стабильную» и выйдет из цикла `$digest`.
- Контекст выполнения JavaScript произведет обработку другого кода JavaScript, не имеющего отношения к AngularJS, которого в приложениях на основе AngularJS может и не быть, и браузер выйдет из контекста выполнения JavaScript.
- Поток обслуживания пользовательского интерфейса сможет войти в контекст отображения и отобразить узел DOM с изменившимся свойством `text`.
- Завершив отображение, браузер вновь вернется в состояние ожидания новых событий.

Как видите, передача изменений из дерева DOM в модель и обратно выполняется в несколько этапов.

# Настройка производительности – определить требования, измерить, настроить и повторить

Процесс настройки производительности требует систематического подхода.

- Для начала необходимо четко определить требования к производительности в измеримых значениях, а также обозначить условия, при которых должны производиться измерения.
- Измерить текущую производительность системы и сопоставить с обозначенными требованиями.
- Если производительность не соответствует требованиям, необходимо выявить и устранить узкие места. После этого следует повторить процедуру измерения, чтобы убедиться, что производительность удалось поднять до требуемого уровня, или чтобы выявить где продолжить искать узкие места.

Из описания последовательности действий, представленного выше, следует два очень важных вывода. Прежде всего, необходимо четко обозначить требования и условия, при которых будут измеряться параметры производительности. Во-вторых, настройка производительности не является главной целью – главная цель заключается в том, чтобы привести систему в такое состояние, когда она будет показывать желаемую производительность.

Фреймворк AngularJS, как любая другая хорошо продуманная библиотека, конструировалась при определенных ограничениях, которые прекрасно описал Мишко Хеври (Miško Hevery), основатель AngularJS (http://stackoverflow.com/a/9693933/1418796):

> *Кому-то может показаться, что низкая производительность приложения обусловлена неэффективностью алгоритма простого сравнения состояний объектов (dirty checking). Однако в этом вопросе необходимо руководствоваться фактическими значениями характеристик производительности, а не теоретическими аргументами. Но давайте сначала определим некоторые ограничения.*
>
> *Люди:*
>
> *Медлительны – любые операции, выполняющиеся меньше 50 мсек, незаметны для человека и потому можно сказать, что они выполняются «мгновенно».*
>
> *Ограниченны – вы едва ли сможете показать на одной странице более 2000 единиц информации, а если и сможете, такой пользовательский интерфейс нельзя назвать ни практичным, ни удобным, так как человек все равно не в состоянии переварить такой объем за один присест.*
>
> *Таким образом, вопрос производительности сводится к следующему: можно ли выполнить 2000 сравнений за 50 мсек даже в очень медлительных браузерах? То есть, выполнить одно сравнение за 25 мксек. Я полагаю, что это не проблема, даже для самых медленных из современных браузеров.*
>
> *Здесь есть одно «но»: сравнение должно быть достаточно простым, чтобы вписаться в интервал 25 мксек. К сожалению, слишком просто можно добавить сложное и медленное сравнение, и, соответственно, слишком просто написать медлительное приложение, если вы не понимаете, что делаете. Но мы надеемся помочь вам решить эту проблему, реализовав вспомогательный модуль, который покажет, какие именно сравнения выполняются слишком медленно.*

Обозначенные здесь условия являются отличными ориентирами для определения практических и теоретических пределов производительности приложений на основе AngularJS, а также направлений поиска узких мест.

К счастью, «вспомогательный модуль», о котором говорит Мишко, уже существует и называется Batarang. Он доступен в виде расши-

рения для браузера Chrome на сайте Chrome Web Store. На рис. 11.1 изображено расширение Batarang в действии. Наряду с прочей полезной информацией, Batarang может измерять скорость вычисления выражений, находящихся под наблюдением, и показывать:

- время вычисления каждого отдельного выражения;
- относительный вклад данного выражения в общее время выполнения цикла `$digest`.

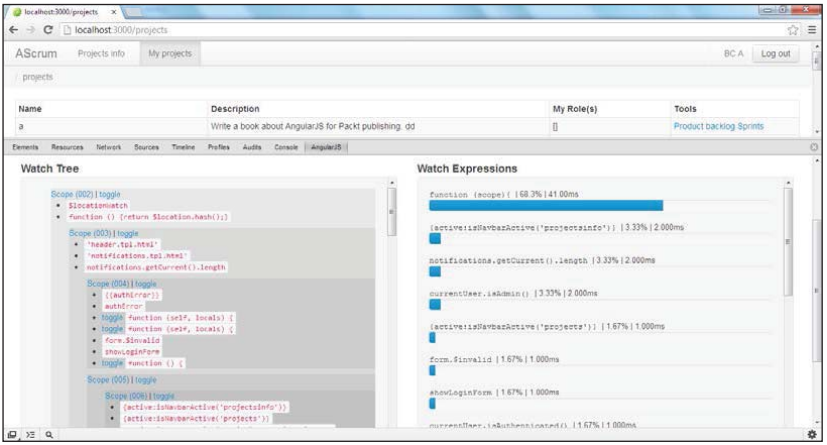


**Рис. 11.1.** Расширение Batarang в действии

Расширение Batarang позволяет быстро находить самые медленные выражения и получать количественную оценку наших усилий по настройке производительности.

## Настройка производительности приложений на основе AngularJS

Любой из нас хотел бы пользоваться «быстрыми» приложениями. Но каждый из нас имеет собственное представление о понятии «быстрое» и именно поэтому нам нужно сосредоточиться на измеримых характеристиках, влияющих на общую производительность. В этой главе мы рассматриваем только проблемы производительности в границах браузеров и не касаемся вопросов, связанных с производительностью сети (они обсуждаются в главе 12). Ограничив рассматриваемую область браузером, мы должны взять в рассмотрение две основные характеристики – использование процессора и потребление памяти.

## Оптимизация использования процессора

Помимо прикладной логики, в приложении выполняется множество операций, связанных с работой фреймворка AngularJS, которые также потребляют процессорное время. Особого внимания требует цикл `$digest`. Нам нужно убедиться, что каждый цикл `$digest` выполняется достаточно быстро, и организовать прикладной код так, чтобы цикл `$digest` запускался фреймворком как можно реже.

### Увеличение скорости выполнения циклов $digest

Итак, одна из основных задач заключается в том, чтобы удержать продолжительность выполнения цикла `$digest` в пределах 50 мсек, чтобы задержки не были заметны человеку. Это важно, потому что в этом случае приложение будет восприниматься, как мгновенно откликающееся на действия пользователя (события DOM).

Существует две основные рекомендации, следование которым поможет остаться в границах «50 мсек на один цикл `$digest`»:

- обеспечить максимально высокую скорость вычисления отдельных выражений;
- ограничить количество сопоставлений, выполняемых в отдельных циклах `$digest`.

### Обеспечьте максимально высокую скорость вычисления выражений

В общем виде синтаксис определения выражения, находящегося под наблюдением фреймворка, выглядит следующим образом:

```
$scope.$watch(watchExpression, modelChangeCallback)
```

Не забывайте, что каждое отдельное выражение, находящееся под наблюдением, состоит из двух отдельных частей: выражения `watchExpression`, результат которого сравнивается со значением модели, и функции `modelChangeCallback`, которая вызывается при обнаружении изменений в модели. Выражение `watchExpression` вычисляется гораздо чаще, чем происходит вызов функции `modelChangeCallback`, и именно поэтому выражение `watchExpression` требует особого внимания.

Каждое выражение `watchExpression` вычисляется как минимум один раз (чаще два раза) в каждом цикле `$digest`. Поэтому, если для вычисления выражения требуется значительное время, его присутствие может существенно ухудшить общую производительность приложения на основе AngularJS. Мы всегда должны уделять особое

внимание выражениям и стараться избегать использования тяжеловесных и дорогостоящих вычислений в них.

В идеале выражение `watchExpression` должно быть максимально простым вычисляться максимально быстро. Существует масса антишаблонов проектирования, которых следует избегать, чтобы добиться высокой скорости вычисления выражений.

Прежде всего следует минимизировать дороговизну вычисления выражений. Обычно в шаблонах используются простые выражения, которые вычисляются достаточно быстро, но существуют две ситуации, когда легко можно допустить появление нетривиальных выражений.

Особое внимание уделяйте выражениям, вызывающим функции, например:

```
{{myComplexComputation()}}
```

так как в этом случае вызов `myComplexComputation()` станет частью выражения, находящегося под наблюдением фреймворка! Другое неочевидное следствие использования функций – в них могут находиться инструкции, осуществляющие вывод отладочных сообщений. Не секрет, что вызов `console.log` значительно замедляет вычисление выражения. Рассмотрим две функции:

```
$scope.getName = function () {
    return $scope.name;
};

$scope.getNameLog = function () {
    console.log('getting name');
    return $scope.name;
};
```

Если исследовать производительность следующих элементов разметки в расширении Batarang:

```
<span>{{getName()}}</span>
<span>{{getNameLog()}}</span>
```

можно заметить существенную разницу, как показано на рис. 11.2.

Всегда удаляйте отладочные вызовы `console.log` при подготовке версии приложения для промышленной эксплуатации и используйте инструменты, такие как `jshint`, которые помогут вам в этом. Никогда не используйте инструкции журналирования в ходе измерения характеристик производительности, так как они сильно искажают результаты.

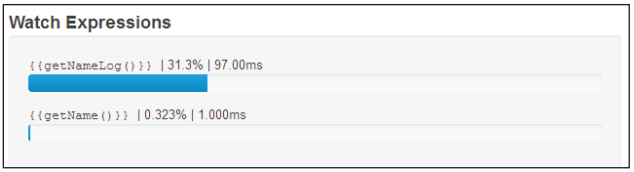


**Рис. 11.2.** Разница в производительности между двумя схожими выражениями

Реализация фильтров – еще одно место, куда неосознанно можно включить дорогостоящие вычисления. Взгляните на следующий пример:

```
{{myModel | myComplexFilter}}
```

Фильтр – это не что иное, как функция, вызываемая как часть выражения под наблюдением с применением специального синтаксиса. Так как вызов функции является частью выражения, она будет вызываться минимум один раз (чаще два раза) в каждом цикле `$digest`. Если фильтр имеет тяжеловесную логику, он замедлит выполнение всего цикла `$digest`. Фильтры особенно обманчивы в отношении производительности, так как могут объявлять зависимости от различных служб и вызывать потенциально дорогостоящие методы.

Каждый фильтр в выражениях AngularJS выполняется как минимум один раз (чаще два раза) в каждом цикле `$digest`. Исследуйте логику работы фильтров, чтобы убедиться, что они не ухудшают производительность приложения.

## Избегайте операций с деревом DOM в выражениях

Иногда возникает соблазн прочитать значение свойства элемента DOM внутри выражения `watchExpression`. Такой подход влечет за собой две проблемы.

Во-первых, доступ к свойствам элементов DOM выполняется медленно, очень медленно. Дело в том, что значения свойств элементов DOM вычисляются в момент чтения. Одного этого уже достаточно, чтобы обращение к свойству внутри выражения значительно замедлило выполнение цикла `$digest`. Например, попытка прочитать свойство, связанное с координатами элемента или его размерами вынудит браузер пересчитать координаты и размеры элемента, а эта операция выполняется на порядки медленнее, чем обращение к свойству объекта JavaScript.

Любая операция с деревом DOM выполняется медленно, а обращения к вычисляемым свойствам выполняются особенно медленно. В действительности проблема заключается в том, что объектная модель документа (DOM) реализована на C++, а вызов функций на C++ из JavaScript является весьма дорогостоящей операцией. Поэтому любое обращение к дереву DOM выполняется на порядки медленнее, чем обращение к свойству объекта JavaScript.

Вторая проблема относится к разряду концептуальных. Вся философия AngularJS основана на том факте, что источником истины является модель, которая декларативно управляет пользовательским интерфейсом. Но, наблюдая за свойствами DOM, мы ставим все с ног на голову! Ни с того ни с сего DOM вдруг начинает управлять моделью, которая в свою очередь управляет пользовательским интерфейсом. В результате мы получаем проблему циклической зависимости.

Иногда бывает заманчиво организовать слежение за изменениями в свойствах DOM, особенно при интеграции сторонних компонентов на JavaScript в приложения на основе AngularJS. Помните, что это может крайне отрицательно сказаться на скорости выполнения цикла `$digest`. Избегайте использования свойств DOM в выражениях, находящихся под наблюдением или, хотя бы, измеряйте производительность цикла `$digest`, после добавления таких выражений.

### Ограничение количества выражений под наблюдением

Если вы максимально оптимизировали все имеющиеся выражения и ликвидировали все узкие места, а скорость выполнения приложения все равно остается неудовлетворительной, значит есть смысл предпринять более радикальные меры.

### Удалите ненужные выражения

Механизм двунаправленного связывания данных, поддерживаемый фреймворком AngularJS, настолько привлекателен и прост в использовании, что иногда может применяться неоправданно, например, для отображения статических данных.

В главе 10 мы обсуждали пример одной такой ситуации использования механизма связывания данных для перевода текста на другой язык. Перевод текста изменяется очень редко (если вообще изменяется!), а добавляя выражение AngularJS для каждой строки, требу-

ющей перевода, мы добавляем массу дополнительных вычислений в каждый цикл `$digest`.

Добавляя новое выражение интерполяции в шаблон, постарайтесь оценить выгоды, которые несет механизм двунаправленного связывания данных. Возможно, значение для шаблона выгоднее будет генерировать на стороне сервера.

### Думайте о пользовательском интерфейсе

Каждое выражение, зарегистрированное в контексте, представляет «переменную часть» страницы. Даже после удаления всех ненужных привязок (которые являются частью цикла `$digest`, но в действительности не изменяются, как описывается выше) нам все еще необходимо решить проблему теоретического предела «2000 переменных на страницу». Практическое решение этой проблемы оказывается еще более сложным, так как требует учесть скорость вычисления каждого отдельного выражения.

Число 2000 является отличным показателем. Но не слишком ли оно мало? Во многом это зависит от конкретных обстоятельств, но, ставя под сомнение это число, в первую очередь необходимо думать о своих пользователях. Смогут ли они «переварить» 2000 «переменных элементов» за один присест? Не оттолкнем ли мы пользователей таким изобилием информации? Может быть лучше реорганизовать приложение так, чтобы сосредоточить внимание пользователей на наиболее важных данных и элементах управления? Принцип очевидности, используемый при построении пользовательских интерфейсов, гласит (http://en.wikipedia.org/wiki/Principles_of_user_interface_design)[1]:

> *Пользовательский интерфейс должен содержать все элементы управления и материалы, необходимые для решения данной задачи, и не отвлекать внимание пользователя посторонней или избыточной информацией. Хороший интерфейс не перегружает пользователя альтернативами и не приводит в замешательство ненужной информацией.*

Упростив пользовательский интерфейс, можно убить сразу двух зайцев – повысить скорость выполнения цикла `$digest` и упростить работу с приложением. Исследования в этом направлении стоят потраченных усилий!

1 Аналогичные рекомендации на русском языке можно найти по адресу: http://ru.wikipedia.org/wiki/Человеко-компьютерное_взаимодействие. – *Прим. перев.*

## Исключите наблюдение за выражениями в невидимых элементах

В AngularJS имеется две директивы, очень удобные, когда элементы DOM требуется отображать только при определенных условиях: `ng-show` и `ng-hide`. Как обсуждалось в главе 4, эти директивы не удаляют элементы из дерева DOM, а просто скрывают с применением соответствующего стиля (`display: none`). Недостаток такого «сокрытия» элементов заключается в том, что эти элементы продолжают оставаться в дереве DOM и все выражения, зарегистрированные этими элементами (и вложенными в них элементами) продолжают вычисляться в каждом цикле `$digest`.

Рассмотрим следующий пример и его характеристики производительности, полученные с помощью расширения Batarang (рис. 11.3):

```
<input ng-model='name'>
<div ng-show="false">
  <span>{{getNameLog()}}</span>
</div>
```

Здесь функция `getNameLog()` просто возвращает имя и использует функцию `console.log`, чтобы имитировать «дорогостоящую» операцию. Начав вводить символы в поле `<input>`, можно заметить, что выражение `getNameLog()` вычисляется для каждого введенного символа. Это происходит, даже если результат вычисления не отображается на экране!


**Рис. 11.3.** Результаты измерения производительности примера с помощью расширения Batarang

Помните, что результаты дорогостоящих вычислений могут не отображаться на экране. Если скрытые элементы снижают скорость работы приложения, подумайте о применении семейства директив `ng-switch`. Эти директивы физически удаляют невидимые элементы DOM из дерева.

### Вызывайте scope.$digest вместо scope.$apply, когда влияние выражения ограничено определенным контекстом

В первой части этой главы говорилось, что AngularJS должен выполнить обход всех контекстов, имеющихся в приложении при выполнении цикла `$digest`. Такой порядок действий предусмотрен, чтобы обработать изменения в родительских контекстах, инициированные в дочерних контекстах. Но иногда бывает точно известно, какие контексты будут затронуты данным конкретным изменением, и мы с успехом можем использовать это знание к своей выгоде.

Если точно известно, какие контексты будут затронуты изменениями в модели, вместо `scope.$apply` можно вызвать метод `scope.$digest` самого верхнего затронутого контекста. Метод `scope.$digest` выполнит цикл `$digest` на ограниченном подмножестве контекстов, и вычислит выражения, зарегистрированные только в этом контексте и в его дочерних контекстах. Этот прием может существенно уменьшить количество вычисляемых выражений и тем самым ускорить выполнение цикла `$digest`.

В качестве примера применения этой методики рассмотрим использование директивы `typeahead` (реализующей функцию автодополнения, как показано на рис. 11.4).

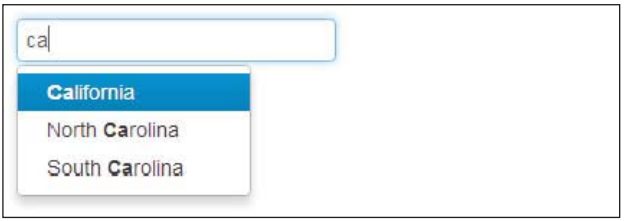


**Рис. 11.4.** Пример использования директивы `typeahead`

Эта директива создает собственный контекст всплывающего окна, где хранятся варианты автодополнения и ссылка на текущий выбранный элемент (**California**). Пользователи могут перемещаться по списку предлагаемых вариантов с помощью клавиатуры (клавиш со стрелками вверх и вниз), выбирая нужный вариант. Когда возникает событие от клавиатуры, нам необходимо вызвать `scope.$apply()`, чтобы активизировать механизм двунаправленного связывания в AngularJS и выделить соответствующий элемент в списке. Но навигация по списку затрагивает только контекст всплывающего окна, поэтому вычисление всех выражений во всех контекстах было бы явным рас-

точительством. Мы можем действовать более тонко и вызывать метод `$digest()` контекста всплывающего окна.

### Удаляйте неиспользуемые выражения

Вы можете столкнуться с ситуацией, когда зарегистрированное выражение должно вычисляться только на определенном этапе работы приложения. Фреймворк AngularJS дает возможность удалять ненужные больше выражения, как показано ниже:

```
var watchUnregisterFn = $scope.$watch('name', function (newValue,
oldValue){

    console.log("Watching 'name' variable");
    ...
});

// позднее, когда выражение станет ненужным:

watchUnregisterFn();
```

Как видно из примера выше, метод `scope.$watch` возвращает функцию, которую можно использовать для удаления данного выражения, когда оно станет ненужным.

### Реже выполняйте цикл $digest

Обычно запуск цикла `$digest` происходит из встроенных директив и служб AngularJS, и этот процесс мы не можем контролировать. Однако мы должны знать об обстоятельствах, вызывающих запуск цикла `$digest`. В общем случае существует четыре типа событий, обработка которых директивами и службами AngularJS влечет за собой вызов метода `scope.$apply()`:

- **события навигации**: пользователь щелкнул на гиперссылке, на кнопке **Back** (Назад) или **Forward** (Вперед), и так далее;
- **события сети**: все обращения к службе `$http` (и операции создания ресурсов с помощью `$resource`) вызывают выполнение цикла `$digest` при получении ответа (положительного или отрицательного);
- **события DOM**: все директивы AngularJS, соответствующие событиям DOM (`ng-click`, `ng-mouseover`, и другие) вызывают выполнение цикла `$digest`, когда запускается обработчик события;
- **таймеры JavaScript**: служба `$timeout`, которая является оберткой вокруг функции `setTimeout`, вызывает выполнение цикла `$digest` при срабатывании таймера.

Как видите, цикл `$digest` может запускаться достаточно часто, особенно когда в приложении возникает множество событий DOM. В большинстве случаев мы не можем воспрепятствовать этому, но иногда есть возможность использовать определенные приемы, позволяющие сократить частоту запуска цикла `$digest`.

Прежде всего можно попробовать уменьшить количество обращений к серверу, реорганизовав прикладной интерфейс серверной части так, чтобы одно действие пользователя инициировало только один запрос XHR. Разумеется, это не всегда возможно, но если серверная часть находится под полным вашим контролем, обязательно следует подумать о возможности применения этого приема. Это дало бы возможность не только уменьшить количество сетевых операций, но и сократить количество циклов `$digest`, выполняемых в единицу времени.

Далее, особое внимание следует уделить использованию таймеров, особенно обернутых службой `$timeout`. По умолчанию эта служба вызывает `scope.$apply` каждый раз, когда генерируется событие от таймера, что может повлечь за собой тяжелые последствия, если не проявлять достаточную осторожность. Чтобы понять, какие проблемы могут возникать, рассмотрим простую директиву, отображающую текущее время:

```
.directive('clock', function ($timeout, dateFilter) {
    return {
        restrict: 'E',
        link: function (scope, element, attrs) {

            function update() {
                // получить текущее время, отформатировать и
                // вывести в элементе DOM
                element.text(dateFilter(new Date(), 'hh:mm:ss'));
                // повторить через 1 секунду
                $timeout(update, 1000);
            }

            update();
        }
    };
})
```

Эта директива используется как разметка `<clock></clock>` и вызывает выполнение цикла `$digest` каждую секунду. Именно поэтому служба `$timeout` предусматривает передачу ей третьего необязательного аргумента, определяющего необходимость вызова `scope.$apply`. Лучше всего реализовать функцию `update()`, как показано ниже:

```
function update() {
    element.text(dateFilter(new Date(), 'hh:mm:ss'));
    $timeout(update, 1000, false);
}
```

Существует возможность избежать автоматического запуска цикла `$digest` службой `$timeout`, если при регистрации таймера передать ей в третьем аргументе значение `false`.

Наконец, легко можно вызвать массу циклов `$digest`, зарегистрировав большое количество обработчиков событий DOM, особенно тех, что связаны с событиями перемещения указателя мыши. Например, ниже представлен наиболее декларативный способ реализации изменения класса CSS элемента (например, чтобы показать его активность) при наведении указателя мыши:

```
<div ng-class='{active: isActive}' ng-mouseenter ='isActive=true'
     ng-mouseleave='isActive=false'>Some content</div>
```

Такая реализация прекрасно работает и позволяет указывать классы CSS в разметке HTML, но она вызывает выполнение цикла `$digest` при каждом пересечении указателем мыши границ данного элемента DOM. Этот прием не порождает каких-то особых проблем, если используется экономно (применяется лишь к нескольким элементам DOM), но он также способен «поставить на колени» приложение, при применении к большому количеству элементов. Если вы начинаете наблюдать падение производительности, связанное с событиями от мыши, подумайте о создании собственной директивы, где можно было бы реализовать непосредственные операции с элементами DOM в ответ на события DOM и изменение модели.

### Ограничивайте количество итераций в каждом цикле $digest

Модели, плохо достигающие стабильного состояния, требуют большего количества итераций цикла `$digest`. Как результат, все выражения, находящиеся под наблюдением, вычисляются по несколько раз. Обычно цикл `$digest` выполняет две итерации, но это число легко может достичь максимально возможного предела (по умолчанию 10), если модель не может достичь стабильного состояния.

Попробуйте перечислить в уме все условия, необходимые для стабилизации вновь добавляемого выражения. Нестабильные выраже-

ния не должны передаваться под наблюдение и вычисляться за рамками цикла `$digest`.

## Оптимизация потреблении памяти

Определение факта изменения модели в AngularJS выполняется с помощью простого сравнения состояний объектов (dirty checking), и если факт изменения подтвердился, выполняется соответствующая операция (обновляются свойства элементов DOM, изменяются другие значения в модели, и так далее). Эффективный алгоритм сравнения играет крайне важную роль для механизма простого сравнения состояний объектов.

### Избегайте глубокого сравнения по мере возможности

По умолчанию для выявления изменений в модели AngularJS сравнивает ссылки (для объектов) или значения (для простых типов). Такие сравнения выполняются очень быстро, но в некоторых ситуациях может понадобиться сравнить объекты по значениям их свойств (так называемое «глубокое сравнение»).

Рассмотрим типичный объект `User`, имеющий несколько разных свойств:

```
$scope.user = {
    firstName: 'AngularJS',
    lastName: 'Superhero',
    age: 4,
    superpowers: 'unlimited',
    // далее следует множество других свойств...
};
```

Допустим, что нам понадобилось хранить полное имя пользователя в единственной переменной внутри модели, обновляемой автоматически, при изменении имени (`firstName`) или фамилии (`lastName`). Для этого мы могли бы зарегистрировать новое выражение в контексте `$scope`:

```
$scope.$watch('user', function (changedUser) {
    $scope.fullName =
        changedUser.firstName + ' ' + changedUser.lastName;
}, true);
```

Мы можем передать методу `$watch` значение `true` в третьем аргументе, чтобы показать, что для объектов должно выполняться глубокое сравнение. В этом случае сравниваться будут объекты целиком

(по значениям свойств) с помощью функции `angular.equal`. В режиме **глубокого сравнения** процесс определения изменений значительно замедляется, так как приходится копировать (вызовом `angular.copy`) и сохранять объекты целиком. Очевидно, что с точки зрения потребления памяти это является расточительством, так как объекты копируются целиком, тогда как нас интересует лишь подмножество их свойств.

К счастью, существует несколько альтернативных решений, позволяющих ограничиться подмножеством свойств объекта. Во-первых, можно просто организовать наблюдение за результатом вычисления полного имени:

```
$scope.$watch(function(scope) {
    return scope.user.firstName + ' ' + scope.user.lastName;
}, function (newFullName) {
    $scope.fullName = newFullName;
});
```

Преимущество данного решения в том, что в памяти (для последующего сравнения) сохраняется только результат вычисления полного имени. Недостаток в том, что полное имя будет вычисляться в каждой итерации цикла `$digest`, даже если свойства объекта вообще не изменялись. Этот прием позволяет сэкономить память за счет увеличения времени вычисления. Того же эффекта можно добиться вызовом функции из выражения AngularJS в шаблоне:

```
{{fullName()}}
```

где функция `fullName()` может быть определена в контексте, как показано ниже:

```
$scope.fullName = function () {
    return $scope.user.firstName + ' ' + $scope.user.lastName;
};
```

Глубокое сравнение влечет за собой два отрицательных следствия для производительности. Оно не только вынуждает AngularJS сохранять копию объекта в памяти, но и само сравнение выполняется медленнее. Попробуйте использовать альтернативные решения, представленные здесь, если интерес представляет лишь подмножество свойств заданного объекта.

Настройка производительности часто сводится к поиску баланса. Это очень хорошо видно в примерах, представленных выше, где нам часто приходилось искать компромисс между потреблением памяти

и нагрузкой на процессор. Оптимальное решение для вашего приложения будет зависеть от выявленных узких мест в нем.

### Учитывайте размеры выражений, передаваемых под наблюдение

Даже если удалось избежать глубокого сравнения, нам все еще нужно принимать во внимание точное значение, которое будет наблюдаться и сравниваться фреймворком. Размер выражения не всегда очевиден, особенно в случае выражений, регистрируемых самим фреймворком AngularJS в процессе обработки шаблонов.

Взгляните на следующий пример с длинным текстом внутри выражения AngularJS:

```
<p>Это очень длинный фрагмент текста, содержащий ссылку на
единственную переменную {{variable}}, которая определена в
текущем контексте. Данный фрагмент может быть очень и очень
длинным, и занимать большой объем памяти. Этот фрагмент такой
длинный потому, что...</p>
```

Возможно вы подумали, что обработав этот шаблон AngularJS создаст выражение, включающее только переменную `variable`. К сожалению это не так – в выражение попадет весь текст, заключенный в тег `<p>`. Это означает, что весь, потенциально очень длинный текст будет скопирован и сохранен в памяти. Мы можем ограничить размер данного выражения, добавив тег `<span>` для выделения части текста, которая должна быть связана с данными:

```
<p>Это очень длинный фрагмент текста, содержащий ссылку на
единственную переменную <span ng-bind='variable'></span>,
которая определена в текущем контексте. Данный фрагмент может
быть очень и очень длинным, и занимать большой объем памяти.
Этот фрагмент такой длинный потому, что...</p>
```

Благодаря такому простому изменению в разметке AngularJS зарегистрирует выражение, состоящее только из переменной `variable`.

## Директива ng-repeat

Если организовать конкурс на самую полезную директиву, его наверняка выиграет директива `ng-repeat`. Она сочетает в себе простой синтаксис с огромными возможностями. К сожалению, директива `ng-repeat` также относится к разряду самых требовательных к производительности директив. Объясняется это двумя причинами. Во-первых, она должна выполнять нетривиальный поиск в коллекции в каждой итерации цикла `$digest`. И во-вторых, при обнаружении

изменений, соответствующие элементы DOM должны быть переупорядочены, что может потребовать выполнения множества операций с деревом DOM.

### Наблюдение за коллекцией в директиве ng-repeat

Директива `ng-repeat` должна следить за изменениями в коллекции, по элементам которой она выполняет итерации. Для корректной работы она должна иметь возможность определять элементы, добавленные в коллекцию, перемещенные внутри коллекции и удаленные из коллекции. Это достигается за счет выполнения сложного алгоритма в каждом цикле `$digest`. Не углубляясь в особенности алгоритма, отметим лишь, что его производительность напрямую связана с размером коллекции.

### Легко создает множество привязок

При использовании с большими наборами данных директива `ng-repeat` легко может зарегистрировать большое количество выражений в текущем контексте.

Простота вычислений играет здесь особую роль. Представьте простую таблицу с пятью столбцами. Каждому столбцу соответствует как минимум одна привязка. Это означает, что каждая строка в таблице создает пять привязок. Если предположить, что теоретический предел приемлемой производительности AngularJS находится в районе 2000 привязок, можно считать, что таблицы с более чем 400 строками будут показывать неудовлетворительную производительность. Конечно, для таблиц с большим количеством столбцов (или с большим количеством привязок на столбец), количество строк, при котором производительность будет оставаться на приемлемом уровне, будет меньше.

Коллекции, включающие более 500 строк, вероятно плохо будут уживаться с директивой `ng-repeat` Точное число строк может зависеть от количества привязок, но не следует ожидать приемлемой производительности от коллекций из тысяч элементов.

К сожалению, директива `ng-repeat` не предоставляет каких-то особых возможностей оптимизации при работе с большими наборами данных. Мы должны стараться фильтровать и урезать коллекции, прежде чем передавать их директиве `ng-repeat`. Для этого подойдут любые приемы: фильтрация, разбиение на страницы, и так далее.

Если вы работаете в предметной области, где действительно необходимо отображать тысячи строк, встроенная директива `ng-repeat`,

вероятно, будет не самым подходящим инструментом. В этом случае вам следует подумать о создании собственной директивы. Такая директива должна не создавать двунаправленные привязки данных для отдельных элементов, а просто отображать элементы DOM, опираясь на содержимое коллекции. Это позволит избежать создания большого количества привязок.

# В заключение

В этой главе мы познакомились с внутренним устройством фреймворка AngularJS. Это совершенно необходимо, чтобы лучше понимать его характеристики производительности и теоретические пределы.

Любые улучшения в приложении, имеющие отношение к производительности, должны начинаться со скрупулезных измерений, чтобы с их помощью выявить узкие места и понять причины их появления. Попытка приступить к оптимизации производительности без точных данных напоминает стрельбу в темноте. К счастью, существует замечательное расширение Batarang для браузера Chrome, позволяющее исследовать выполняющееся приложение.

В первую очередь обратите внимание на время выполнения цикла `$digest`, так как эта характеристика определяет, как пользователь будет воспринимать приложение в целом. Стоит позволить циклу `$digest` выполняться дольше 50–100 мсек, и пользователи начнут чувствовать задержки в реакции приложения. Именно поэтому мы потратили так много времени в этой главе на обсуждение особенностей работы цикла `$digest` и способов повысить его скорость.

Время выполнения цикла `$digest` пропорционально количеству выражений под наблюдением и времени их вычисления. Уменьшить время выполнения цикла `$digest` можно, ограничив количество наблюдаемых выражений и/или увеличив скорость их вычисления. Кроме того, можно также сократить количество вызовов цикла в единицу времени.

Потребление памяти – это еще один важный аспект, влияющий на общую производительность. В приложениях на основе AngularJS дополнительные драгоценные байты памяти могут потребляться при использовании приема глубокого сравнения. Его следует избегать по мере возможности, причем не только когда выражения регистрируются вручную (вызовом `scope.$watch`), но при определении шаблонов.

Особенно сильное влияние на производительность, как в смысле нагрузки на процессор, так и в смысле потребления памяти, оказыва-

ет директива `ng-repeat`. Она легко может стать источником проблем с производительностью, при применении к коллекции из нескольких сотен элементов. Чтобы извлечь максимум пользы из директивы `ng-repeat`, следует ограничивать размеры коллекций. Если вам понадобится организовать отображение коллекций из нескольких сотен элементов, подумайте о том, чтобы написать собственную директиву, отвечающую вашим конкретным требованиям.

В следующей главе мы узнаем, как подготовить настроенное приложение к развертыванию в промышленной среде, а также обсудим шаблоны проектирования, связанные с увеличением производительности сетевых операций.

# ГЛАВА 12. Подготовка и развертывание веб-приложений на основе AngularJS

После создания, тестирования и настройки производительности наступает момент развертывания приложения в промышленной среде. Но не будем торопиться, нам еще нужно кое о чем позаботиться, прежде чем приложение будет полностью готово к эксплуатации!

Сначала нужно убедиться, что приложение экономно использует ресурсы сети. Для этого следует ограничить количество HTTP-запросов, а также объем данных, загружаемых с каждым отдельным запросом. Предварительная загрузка и минификация статических ресурсов – вот два основных способа уменьшения сетевого трафика, и в этой главе мы посмотрим, как можно применять эти приемы в контексте приложений на основе AngularJS.

Начальная страница любого веб-приложения позволяет пользователям составить первые впечатления о нем и чего ожидать от него. Если самый первый опыт окажется негативным, это может оттолкнуть пользователей от приложения, в которое мы вложили столько сил и времени. Именно поэтому так важно оптимизировать производительность первой страницы приложения. В этой главе мы обсудим различные приемы, которые помогут улучшить впечатление от первой страницы.

В последнем разделе рассказывается о поддержке различных браузеров в AngularJS, с особым упором на Internet Explorer.

В этой главе вы узнаете:

- как минимизировать сетевой трафик при загрузке статических ресурсов за счет минификации и объединения программного

кода на JavaScript, а так же за счет предварительной загрузки шаблонов;
- как оптимизировать начальную страницу;
- какие браузеры поддерживаются фреймворком AngularJS и какие действия следует предпринять, чтобы обеспечить нормальную его работу в Internet Explorer.

# Повышение производительности сетевых операций

Как веб-разработчики, мы должны создавать для своих пользователей функциональные приложения с простым и понятным интерфейсом. Пользователи должны получать первые положительные впечатления еще до того, как приложение запустится. С этой целью мы можем уменьшить объем HTTP-трафика и ускорить запуск приложения, загружая меньший объем данных с каждым HTTP-запросом и ограничив их количество.

## *Минификация статических ресурсов*

Один из способов уменьшить время загрузки заключается в уменьшении объема данных, передаваемых между сервером и браузером. В настоящее время для этого часто используется прием минификации кода JavaScript, CSS и HTML. Благодаря применению этого приема уменьшается объем данных, которые требуется загрузить в браузер. Дополнительно он делает код менее читаемым, что можно рассматривать как минимальную защиту от любопытных глаз.

Для минификации кода CSS и HTML вы можете применять любые привычные вам инструменты минификации, но что касается кода JavaScript, использующего особенности фреймворка AngularJS, – здесь необходимо проявить некоторое внимание, прежде чем обрабатывать его стандартными инструментами.

### Как AngularJS определяет зависимости?

Внутри фреймворка AngularJS широко используется **механизм внедрения зависимостей** (Dependency Injection, DI). И в наших собственных приложениях на основе AngularJS мы легко можем определять зависимости от зарегистрированных служб и внедрять их в свой код.

Фреймворк AngularJS способен определять зависимости той или иной функции, получать их с помощью службы `$injector` и передавать их функции в виде аргументов. Механизм AngularJS DI анализирует зависимости функций, используя на удивление простой прием. Чтобы увидеть, как он работает, возьмем в качестве примера контроллер из приложения SCRUM:

```
angular.module('projects', ['resources.projects'])
    .controller('ProjectsViewCtrl', function ($scope, $location) {
    // реализация контроллера
});
```

Функция, объявленная в параметре контроллера `ProjectsViewCtrl`, определяет зависимость от двух служб: `$scope` и `$location`. Фреймворк AngularJS распознает эти зависимости, преобразуя определение функции в строку и используя регулярные выражения для поиска совпадений с именами аргументов функции. Разберем этот прием на следующем простом примере:

```
var ctrlFn = function($scope, $location) {};
console.log(ctrlFn.toString());
```

В результате в консоль будет выведено определение функции в виде строки:

```
"function ($scope, $location) {}"
```

Наличия определения функции в виде строки вполне достаточно для сопоставления с некоторыми простыми регулярными выражениями с целью поиска имен аргументов и, соответственно, требуемых зависимостей.

Теперь посмотрим, что произойдет, если подвергнуть определение функции минификации. Точный вид минифицированной версии может отличаться при использовании разных инструментов, тем не менее большинство минификаторов переименовывают аргументы, из-за чего минифицированная версия будет напоминать следующий код:

```
angular.module('projects', ['resources.projects'])
    .controller('ProjectsViewCtrl', function (e, f) {
    // минифицированный код контроллера ссылается на новые
имена аргументов
});
```

Очевидно, что AngularJS не сможет опознать зависимости в минифицированной функции, так как `e` и `f` не являются именами служб, известных фреймворку.

## Создание кода JavaScript, поддерживающего минификацию

Стандартные инструменты минификации программного кода на JavaScript уничтожают информацию, заключенную в именах параметров, так необходимую фреймворку AngularJS для определения зависимостей. Поскольку эта информация исчезает из сигнатуры функции, нам придется использовать другой механизм.

AngularJS поддерживает несколько способов определения зависимостей в минифицированном коде, однако мы рекомендуем использовать аннотации в виде массивов, как показано в следующем примере:

```
angular.module('projects', ['resources.projects'])
    .controller('ProjectsViewCtrl',
                ['$scope', '$location', function ($scope, $location) {
        // реализация контроллера
}]);
```

После минификации этот код будет выглядеть так:

```
angular.module('projects', ['resources.projects'])
    .controller('ProjectsViewCtrl', ['$scope', '$location',
function (e, f) {
    // реализация контроллера
}]);
```

Даже если аргументы функции `function` изменятся, элементы массива не будут затронуты минификацией и AngularJS получит достаточный объем информации, чтобы найти все зависимости для данной функции.

Если вам потребуется обеспечить поддержку минификации в своем коде, замените все функции (объявляющие хотя бы одну зависимость) массивами, начальными элементами которых являются строки с именами аргументов, а последним – сама функция.

Синтаксис определения зависимостей в виде массива на первый взгляд может выглядеть странным, поэтому, чтобы выработать привычку, рассмотрим еще несколько примеров из приложения SCRUM. В следующих разделах будет представлено еще несколько наиболее типичных примеров.

Весь код приложения SCRUM написан с использованием аннотаций зависимостей в виде массивов. Вы можете загрузить его на сайте GitHub и использовать как справочник по использованию аннотаций в различных ситуациях.

### Модули

Функции config и run на уровне модуля так же позволяют внедрять зависимости. Ниже приводится пример определения зависимостей времени настройки с поддержкой минификации:

```
angular.module('app')
    .config(['$routeProvider', '$locationProvider',
             function ($routeProvider, $locationProvider) {
        $locationProvider.html5Mode(true);
        $routeProvider.otherwise({redirectTo:'/projectsinfo'});
    }]);
```

Функции, вызываемые в блоке run, могут аннотироваться, как показано ниже:

```
angular.module('app')
    .run(['security', function(security) {
        security.requestCurrentUser();
}]);
```

### Провайдеры

Фреймворк AngularJS поддерживает несколько способов регистрации провайдеров (рецептов создания экземпляров объектов). Фабрики являются, пожалуй, наиболее типичным способом определения новых объектов с единственным экземпляром (singletons):

```
angular.module('services.breadcrumbs', [])
    .factory('breadcrumbs', ['$rootScope', '$location',
                             function($rootScope, $location){
        ...
}]);
```

Возможность объявлять декораторы служб позволяет легко добавлять в существующий код новые возможности. Синтаксис определения декораторов, и без того не простой, читается еще труднее, если в него добавить аннотации зависимостей в виде массивов. Чтобы вы могли лучше познакомиться с этим синтаксисом, ниже приводится пример определения декоратора с зависимостями:

```
angular.module('services.exceptionHandler')
    .config(['$provide', function($provide) {
        $provide.decorator('$exceptionHandler',
             ['$delegate', 'exceptionHandlerFactory',
              function ($delegate, exceptionHandlerFactory) {
                return exceptionHandlerFactory($delegate);
        }]);
    }]);
```

### Директивы

Определение директив с поддержкой минификации почти не отличается от определения других провайдеров. Например, определение директивы `field`, описанной в главе 9, выглядит так:

```
.directive('field', ['$compile', '$http', '$templateCache',
    '$interpolate', function($compile, $http, $templateCache,
    $interpolate) {
    ...
    return {
        restrict:'E',
        priority: 100,
        terminal: true
        ...
    };
}]);
```

Определение контроллеров директив требует дополнительного внимания, так как первое время синтаксис может казаться несколько необычным. Ниже приводится пример директивы `accordion` (так же обсуждается в главе 9) объявляющей контроллер:

```
.directive('accordion', function () {
    return {
        restrict:'E',
        controller:['$scope', function ($scope) {
        // Этот массив хранит группы accordion-group
        this.groups = [];

        ...
        }],
        link: function(scope, element, attrs) {
            element.addClass('accordion');
            ...
        }
    };
})
```

### Недостатки аннотаций зависимостей в виде массивов

Аннотации зависимостей в виде массивов обеспечивают поддержку минификации в коде, использующем особенности фреймворка AngularJS, однако эта поддержка достигается за счет дублирования кода. Фактически, этот прием требует дважды повторить список аргументов функции: в самой функции и в массиве. Как и любое дублирование кода, это может превратиться в проблему, особенно когда потребуется выполнить рефакторинг существующего кода.

При таком подходе легко можно забыть добавить имя нового аргумента в массив аннотаций или перепутать порядок следования аргументов, как показано в следующем примере:

```
angular.module('app')
    .config(['$locationProvider', '$routeProvider',
             function ($routeProvider, $locationProvider) {
        $locationProvider.html5Mode(true);
        $routeProvider.otherwise({redirectTo:'/projectsinfo'});
}]);
```

Определение аннотаций требует концентрации, так как ошибки, вызванные неправильными аннотациями, отыскать будет очень непросто.

Вы можете попробовать избавить себя от сложностей определения аннотаций зависимостей, взяв на вооружение инструменты, используемые на заключительных стадиях сборки и добавляющие аннотации автоматически. Такие инструменты сложны в разработке (так как в них требуется реализовать анализ кода на JavaScript) и пока не получили широкого распространения. Тем не менее, если вы используете систему сборки на основе `Grunt.js`, попробуйте применить инструмент `ngmin` (https://github.com/btford/ngmin).

## Предварительная загрузка шаблонов

Благодаря AngularJS мы имеем массу возможностей разделить разметку HTML на более мелкие шаблоны, пригодные к многократному использованию. AngularJS позволяет хранить шаблоны в отдельных файлах и загружать их динамически по мере необходимости. Деление на мелкие файлы дает определенные удобства на стадии разработки и способствует более удачной организации кода, но оно отрицательно сказывается на производительности.

Рассмотрим типичное определение маршрута:

```
angular.module('routing_basics', [])
    .config(function($routeProvider) {
        $routeProvider
            .when('/admin/users/list', {templateUrl: 'tpls/users/
list.html'})
            ...
})
```

Шаблон маршрута хранится в отдельном файле (`tpls/users/list.html`), поэтому AngularJS будет вынужден загружать этот файл,

перед тем как выполнить переход по этому маршруту. Очевидно, что для загрузки шаблона потребуется дополнительное время. Тратя дополнительное время на выполнение сетевых взаимодействий, мы теряем возможность получить быстрый и отзывчивый пользовательский интерфейс.

Определение маршрута – это лишь один пример конструкций, которые могут ссылаться на шаблоны. Ссылки на шаблоны могут также содержаться в директиве `ng-include` и в свойствах `templateUrl` ваших собственных директив.

Предварительная загрузка шаблонов может существенно уменьшить объем сетевого трафика и улучшить отзывчивость пользовательского интерфейса. Фреймворк AngularJS поддерживает два немного отличающихся способа предварительной загрузки шаблонов перед их первым использованием: директива `<script>` и служба `$templateCache`.

Фреймворк достаточно интеллектуален и сохраняет в кеше (служба `$templateCache`) все загруженные им шаблоны. Проще говоря, AngularJS сначала проверит содержимое службы `$templateCache` и лишь потом выполнит сетевой запрос, если это понадобится. В результате, один и тот же шаблон никогда не загружается из сети дважды. Заправив службу `$templateCache`, можно быть уверенными, что все шаблоны будут готовы к использованию после запуска приложения и больше не будут загружаться из сети.

### Предварительная загрузка шаблонов директивой <script>

В AngularJS имеется очень удобная директива `<script>`, которую можно использовать для предварительной загрузки отдельных шаблонов в службу `$templateCache`. Обычно загрузку шаблонов можно выполнять в начальной странице приложения (`index.html` или другой), загружаемой браузером. Возвращаясь к предыдущему примеру маршрута, загрузку шаблонов можно было бы реализовать так:

```
<script type="text/ng-template" id="tpls/users/list.html">

  <table class="table table-bordered table-condensed">
    <thead>
      <tr>
        <th>E-mail</th>
        <th>Last name</th>
        <th>First name</th>
      </tr>
    </thead>
```

```
      <tbody>
        ...
      </tbody>
    </table>
  </script>
```

Чтобы выполнить предварительную загрузку шаблона, необходимо заключить его содержимое в тег `<script>` с типом `type: text/ng-template`, поддерживаемым фреймворком AngularJS. Адрес URL шаблона должен указываться как значение атрибута `id`.

Любой тег `<script>`, содержащий шаблон, должен определяться как дочерний элемент DOM с директивой `ng-app`. Теги `<script>`, которые располагаются за пределами поддерева DOM, управляемого фреймворком AngularJS, не распознаются и как результат, соответствующие шаблоны не будут загружаться предварительно, а только при первой попытке их использования.

Было бы очень тяжело сопровождать шаблоны, внедренные в начальную страницу приложения (`index.html` или другую), к тому же в процессе разработки предпочтительнее хранить все шаблоны в отдельных файлах. По этой причине окончательный файл `index.html` желательно генерировать динамически, в ходе процедуры сборки приложения.

### Заполнение службы $templateCache

Прием предварительной загрузки с помощью директивы `<script>` отлично подходит для небольшого количества шаблонов, но он малопригоден для использования в крупных проектах. Главная проблема заключается в необходимости создавать файл `index.html` в процессе сборки приложения. Проблема обусловлена тем, что в этом файле обычно находится масса рукописного кода. Смешивание сгенерированного динамически и рукописного кода – дурной тон, к тому же список загружаемых шаблонов лучше хранить в отдельном файле. Другая проблема, сопутствующая использованию директивы `<script>`, заключается в невозможности предварительной загрузки шаблонов для директив в тестовом окружении. К счастью существует средство, решающее обе эти проблемы.

Мы можем организовать наполнение `$templateCache` содержимым на этапе запуска приложения. Каждый элемент в службе `$templateCache` представлен парой ключ/значение, где ключ – это адрес URL шаблона, а значение – тело шаблона (преобразованное в строку JavaScript).

Например, пусть имеется шаблон со следующим содержимым:

```
<div class='hello'>Hello, {{world}}!</div>
```

и адресом URL `tpls/hello.tpl.html`. Этот шаблон можно было бы поместить в службу `$templateCache`, как показано ниже:

```
angular.module('app', []).run(function($templateCache){
    $templateCache.put('tpls/hello.tpl.html',
                       '<div class=\'hello\'>Hello,
{{world}}!</div>');
});
```

Добавлять шаблоны в кеш «вручную» очень неудобно, так как при этом необходимо позаботиться об экранировании кавычек (как показано в примере выше, в атрибуте `class`). И снова решить эту проблему можно с помощью задания, выполняемого на этапе сборки, которое выполнит обход всех шаблонов в проекте и сгенерирует код JavaScript, ответственный за наполнение `$templateCache`. Именно такое решение используется в примере приложения SCRUM – задание генерирует модуль (`templates`) с блоком `run`, выполняющим заполнение `$templateCache`. Получившийся модуль `templates` можно добавить как зависимость в модуль приложения, наряду с другими функциональными модулями:

```
angular.module('app', ['login', 'dashboard', 'projects', 'admin',
    'services.breadcrumbs', 'services.i18nNotifications', 'services.
    httpRequestTracker', 'directives.crud' 'templates']);
```

Предварительная загрузка шаблонов в `$templateCache` – настолько распространенная практика в сообществе AngularJS, что в `Grunt.js` была реализована отдельная задача автоматизации этого процесса: `grunt-html2js` (https://github.com/karlgoldstein/grunt-html2js). Эта задача появилась во многом благодаря разработанной процедуре сборки примера приложения SCRUM, описываемого в этой книге.

В действительности система сборки приложения SCRUM создает отдельный модуль для каждого шаблона, где роль имени шаблона играет адрес URL, как показано ниже:

```
angular.module("header.tpl.html", [])
    .run(["$templateCache", function($templateCache) {
        $templateCache.put("header.tpl.html",
            "<div class=\"navbar\" ng-controller=\"HeaderCtrl\">" +
            ...
            "</div>");
```

```
}]);

angular.module("login/form.tpl.html", [])
    .run(["$templateCache", function($templateCache) {
        $templateCache.put("login/form.tpl.html",
            "<div modal=\"showLoginForm\" close=\"cancelLogin()\">" +
            ...
            "</div>" +
            "");
}]);
```

Все модули шаблонов затем добавляются как зависимости модуля `templates`, поэтому приложение может просто объявить зависимость от этого модуля и тем самым подключить сразу все шаблоны:

```
angular.module('templates', ['header.tpl.html', 'login/form.tpl.html',
...]);
```

Наличие отдельного модуля для каждого шаблона очень удобно для тестирования/ Благодаря такой организации можно точно управлять перечнем загружаемых шаблонов в службу `$templateCache` и гарантировать максимальную изолированность отдельных тестов.

### Комбинирование различных приемов предварительной загрузки

Предварительная загрузка шаблонов, как и многие другие приемы оптимизации производительности, фактически является компромиссным решением. Помещая шаблоны в службу `$templateCache`, мы уменьшаем объем сетевого трафика и улучшаем отзывчивость пользовательского интерфейса за счет повышенного потребления памяти. Каждая запись в службе `$templateCache` потребляет дополнительные байты, пропорционально размеру соответствующего шаблона. Это может порождать проблемы в действительно крупных приложениях (имеются в виду приложения, включающие несколько сотен шаблонов), особенно если приложение делится на множество логических разделов. В зависимости от приложения и особенностей его использования может так получиться, что большая часть страниц останутся неиспользованными и значительная часть шаблонов будет загружена напрасно.

К счастью от нас не требуется использовать логику «все или ничего» для реализации предварительной загрузки шаблонов. Мы можем комбинировать различные стратегии. Например, наиболее часто используемые шаблоны можно загружать на начальном этапе запуска приложения, а дополнительные шаблоны загружать по мере необхо-

димости. Кроме того, загружать шаблоны в службу `$templateCache` можно и после запуска приложения. Например, когда пользователь войдет в определенный раздел приложения, можно загрузить все шаблоны для этого раздела.

# Оптимизация начальной страницы

Оптимизация производительности начальной страницы играет важную роль в любом веб-приложении. В конце концов – это первая страница, которую увидит пользователь и по которой будет составлено начальное мнение о приложении.

Реализация загрузки и «правильного» отображения первой страницы в одностраничных веб-приложениях является не самым простым делом. Обычно в этот момент требуется выполнить значительное количество сетевых операций и загрузить множество сценариев, прежде чем механизм отображения на основе JavaScript сможет приступить к работе. В данном разделе описываются некоторые приемы, характерные для приложений на основе AngularJS, которые можно использовать для улучшения впечатлений пользователя от скорости загрузки и отображения начальной страницы.

## *Избегайте отображения шаблонов в необработанном виде*

Веб-приложения на основе AngularJS должны сначала загрузить код фреймворка и прикладные сценарии, и только потом они получат возможность обрабатывать и отображать шаблоны. Это означает, что пользователи могут на мгновение увидеть шаблоны AngularJS в необработанном виде. Возьмем в качестве примера классическое приложение «Hello World!» и допустим, что для загрузки кода JavaScript требуется достаточно продолжительный интервал времени. В этом случае пользователи могли бы видеть страницу, изображенную на рис. 12.1, пока не загрузятся и не запустятся сценарии.

После загрузки всех сценариев приложение запустится, обработает выражение интерполяции (`{{name}}`) и на экране появится значение, определенное в контексте. Процесс запуска приложения в данном случае весьма далек от идеала: множество пользователей не только увидят странное выражение на экране, но и заметят мерцание

пользовательского интерфейса, когда выражения будут замещены их значениями. В AngularJS имеется две директивы, позволяющие избежать этих отрицательных эффектов: `ng-cloak` и `ng-bind`.

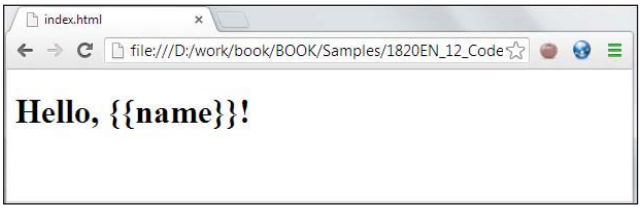


**Рис. 12.1.** Если загрузка сценариев будет выполняться достаточно долго, пользователи увидят необработанный шаблон

## Сокрытие фрагментов дерева DOM с помощью ng-cloak

Директива `ng-cloak` позволяет скрывать (`display:none`) фрагменты дерева DOM до момента, пока AngularJS не будет готов обработать страницу и подготовить ее к отображению. Чтобы скрыть выбранные части дерева DOM, достаточно поместить директиву `ng-cloak` в соответствующие элементы DOM, как показано ниже:

```
<div ng-controller="HelloCtrl" ng-cloak>
  <h1>Hello, {{name}}!</h1>
</div>
```

Если первая страница приложения состоит из нескольких динамических частей, можно скрыть страницу целиком, поместив директиву `ng-cloak` в элемент `<body>`. Напротив, если первая страница кроме динамического имеет еще и статическое содержимое, лучшим выбором будет поместить директиву в элементы, окружающие динамические части, так как в этом случае пользователи смогут ознакомиться со статическим содержимым, пока выполняется загрузка и запуск AngularJS.

Директива `ng-cloak` скрывает элементы с динамическим содержимым, применяя к ним правила CSS, и делает их видимыми вновь, когда AngularJS будет готов к работе и скомпилирует дерево DOM. Для сокрытия элементов DOM используется правило CSS, соответствующее HTML-атрибуту `ng-cloak`:

```
[ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak], .ng-cloak,
.x-ng-cloak {
    display: none;
}
```

Эти правила создаются динамически, в главном сценарии AngularJS, поэтому их не требуется определять вручную.

Когда фреймворк AngularJS загрузится и запустится, он обойдет дерево DOM, скомпилирует и свяжет все директивы, и удалит атрибут `ng-cloak` из всех элементов DOM, где он присутствует. В результате правила CSS, соответствующие атрибуту `ng-cloak`, перестанут действовать и элементы DOM станут видимыми.

### Сокрытие отдельных выражений с помощью ng-bind

Директива `ng-cloak` – отличный инструмент сокрытия больших фрагментов DOM до момента, когда AngularJS будет готов к работе. Она устраняет нежелательные эффекты в пользовательском интерфейсе, но не отменяет тот факт, что пользователи не будут видеть содержимое, пока динамические сценарии не будут полностью готовы к работе.

В идеале было бы желательно создать на стороне сервера полностью готовую первую страницу и затем дополнить ее динамической функциональностью после инициализации фреймворка AngularJS. Разработчики экспериментируют с разными серверными технологиями отображения, но к моменту написания этих строк в AngularJS официально отсутствовала поддержка таких технологий. Тем не менее, если первая страница приложения в основном состоит из статического содержимого, вместо выражений интерполяции можно использовать директиву `ng-bind`.

Допустим, что в преимущественно статической разметке HTML присутствует пара-тройка директив интерполяции, как показано ниже:

```
<div ng-controller="HelloCtrl">
  Hello, {{name}}!
</div>
```

Тогда выражение `{{name}}` можно заменить атрибутом `ng-bind`:

```
<div ng-controller="HelloCtrl">
  Hello, <span ng-bind="name"></span>!
</div>
```

Преимущество такого решения заключается в отсутствии выражений AngularJS в фигурных скобках внутри разметки HTML, благодаря чему они не будут видны пользователям при отображении необработанных страниц. Нестандартные атрибуты (`ng-bind`) не

распознаются браузером и просто игнорируются до момента, пока AngularJS не получит возможность обработать их. Используя данный прием можно даже подставить в разметку значение по умолчанию:

```
<div ng-controller="HelloCtrl">
  Hello, <span ng-bind="name">Your name</span>!
</div>
```

Значение по умолчанию (`Your name`) будет отображаться до момента, пока AngularJS не обработает директиву `ng-bind`.

Прием на основе директивы `ng-bind`, описанный выше, должен использоваться только в начальной странице приложения. Во всех последующих страницах можно смело использовать выражения интерполяции, так как к моменту их отображения AngularJS уже будет загружен и обработает их.

## AngularJS и подключение прикладных сценариев

Данный раздел книги, посвященный оптимизации начальной страницы, был бы неполным без обсуждения оптимизации загрузки сценариев. Эта тема является еще более важной, если учесть рост популярности асинхронных загрузчиков сценариев.

### Ссылки на сценарии

Размещение тегов `<script>` в самом конце начальной страницы считается хорошей практикой. Но этот прием, как и любой другой, должен оцениваться и применяться, только если он действительно подходит. Когда теги `<script>` помещаются в конец страницы, загрузка и интерпретация файлов JavaScript не будут блокировать загрузку и отображение разметки HTML. Этот прием имеет смысл использовать в страницах со значительным объемом статического содержимого с редкими вкраплениями кода JavaScript. Но для высокодинамичных веб-приложений ценность этого приема весьма сомнительна.

Если начальная страница в основном состоит из статического содержимого, с небольшим количеством динамических фрагментов, перенесите теги `<script>` в самый конец страницы и используйте директиву `ng-bind` для сокрытия привязок, пока страница будет загружаться и обрабатываться. В противном случае перенесите теги `<script>` в тег `<head>` и используйте директиву `ng-cloak`.

## AngularJS и определение асинхронных модулей

Спецификация **определения асинхронных модулей** (Asynchronous Module Definition, AMD), получившая популярность благодаря `Require.js` и другим похожим библиотекам на JavaScript, описывает набор правил создания модулей многократного пользования, которые могут загружаться асинхронно. Такие модули могут загружаться по требованию, с учетом всех зависимостей. С другой стороны определения модулей AMD можно использовать в сценариях сборки для подготовки объединенной версии всех модулей, необходимых приложению для работы.

В AngularJS имеется собственная система модулей. Несмотря на то, что и в AngularJS, и в спецификации AMD используется слово «модуль», в него вкладывается разный смысл, как описывается ниже.

- Модули в AngularJS определяют как отдельные классы и объекты JavaScript должны комбинироваться во время выполнения приложения. Модули AngularJS не требуют использования какой-то определенной стратегии загрузки. Напротив, AngularJS предполагает, что к моменту запуска приложения все модули будут загружены браузером.
- Основной задачей спецификации AMD, напротив, является определение стратегии загрузки сценариев. Она позволяет разбить приложение на несколько маленьких сценариев и асинхронно загружать сценарии по мере необходимости.

Система модулей AngularJS и спецификация AMD решают разные задачи и их не следует смешивать.

В текущей версии (1.1.x), AngularJS ожидает, что все сценарии, составляющие приложение, будут загружены до его запуска. Здесь нет места для применения технологий асинхронной загрузки модулей. У нас просто нет возможности регистрировать новые модули и провайдеры (службы, контроллеры, директивы и так далее) после инициализации приложения на основе AngularJS.

В текущей версии AngularJS асинхронная загрузка модулей может использоваться для загрузки библиотек и сценариев JavaScript только до инициализации приложения. Загрузка дополнительного прикладного кода по требованию не поддерживается.

Отсутствие поддержки загрузки модулей по требованию в AngularJS может выглядеть как большой недостаток, но этот факт следует оценивать в правильном контексте. Спецификация AMD была разработана с целью:

- обеспечить асинхронную загрузку, которая не блокирует браузер и позволяет ему загружать разные ресурсы одновременно;
- дать возможность загружать сценарии по требованию, в ходе навигации по приложению, тем самым избавляя от необходимости загружать весь код сразу;
- на основе определений модулей выявлять зависимости между модулями с целью развертывания приложения в промышленной среде.

Вы можете применять спецификацию AMD для организации загрузки библиотеки AngularJS, всех сторонних зависимостей и прикладного кода, но только до инициализации приложения. Преимущество такого решения очевидно – оно позволит организовать асинхронную загрузку сценариев и избежать блокировки браузера для обработки тегов `<script>`. Это может привести к увеличению скорости загрузки приложения (а может и не привести!), в зависимости от количества загружаемых библиотек.

При использовании технологии AMD для загрузки модулей вы лишаетесь возможности использовать директиву `ng-app` для инициализации приложения на основе AngularJS. Причина в том, что AngularJS начинает обработку дерева DOM по событию готовности документа. В этот момент модули, загружаемые асинхронно, могут оказаться еще не загруженными в браузер, и AngularJS попытается инициализировать приложение еще до того, как все необходимые сценарии JavaScript будут получены. На этот случай в AngularJS предусмотрен программный способ инициализации приложения из кода JavaScript: `angular.bootstrap`.

Если вы решите использовать технологию асинхронной загрузки модулей в приложении на основе AngularJS, необходимо удалить директиву `ng-app` и вызывать метод `angular.bootstrap` из своего сценария JavaScript. Таким способом вы сможете управлять моментом запуска инициализации своего приложения.

Необходимо также рассмотреть тему загрузки сценариев по требованию. Этот прием уменьшает время начальной загрузки приложения и потребление памяти за счет увеличения объема сетевого трафика. Загрузку по требованию имеет смысл применять, только если файлы с кодом JavaScript являются «достаточно большими» и цена затрат времени на дополнительные сетевые операции окажется приемлемой. Но обычно код, опирающийся на использование AngularJS, действительно получается очень компактным, в сравнении с альтер-

нативными фреймворками. А минификация и сжатие его позволяет получить совсем небольшие файлы, которые не имеет смысла загружать асинхронно.

Создатели асинхронных модулей понимают, что для загрузки большого количества маленьких файлов требуется выполнить большое количество запросов XHR. Падение производительности и увеличение сетевого трафика могут оказаться столь значительными, что на этом фоне более привлекательно будет выглядеть прием загрузки всех сценариев, объединенных в один файл. Именно поэтому было создано множество инструментов AMD, способных проанализировать зависимости между модулями и пакетами для нужд развертывания. Но объединение файлов JavaScript можно также реализовать и в сценарии сборки. Это особенно легко сделать в веб-приложениях на основе AngularJS, так как модули AngularJS допускают возможность загрузки в произвольном порядке.

Учитывая, что асинхронная загрузка прикладного кода не поддерживается текущей версией фреймворка AngularJS, сложна в настройке, а выгоды от ее применения весьма сомнительны, мы не советуем использовать асинхронные загрузчики (`Require.js` и аналогичные) в приложениях на основе AngularJS 1.1.x.

# Поддержка браузеров

Исходный код AngularJS подвергается всестороннему тестированию с применением **сервера непрерывной интеграции** (Continuous Integration, CI). Любое изменение в коде фреймворка запускает целую серию модульных тестов. Никакой фрагмент кода не может попасть в фреймворк, без соответствующих модульных тестов. Такой строгий подход к тестированию гарантирует стабильность фреймворка в долгосрочной перспективе и плавное его развитие.

На момент написания этих строк сервер непрерывной интеграции выполнял более 2000 отдельных тестов в разных браузерах: в последних версиях Chrome, Firefox, Safari и Opera, а также в версиях 8, 9 и 10 Internet Explorer. Количество тестов и список браузеров, используемых для тестирования, могут служить наглядным доказательством зрелости фреймворка и его надежности. AngularJS гарантированно будет работать в браузерах, перечисленных здесь. Весьма вероятно, что он так же будет работать и в других современных браузерах, не указанных здесь (например, в браузерах для мобильных устройств).

## Поддержка Internet Explorer

Едва ли у кого-то вызовет удивление, если заявить, что поддержка Internet Explorer имеет свои особенности. Если версии IE9 и IE10 должны поддерживаться «из коробки», то поддержка IE8 требует особого внимания.

Обычно инициализация приложения на основе AngularJS реализуется с помощью директивы `ng-app="application_name"`. К сожалению, в IE8 этого недостаточно и необходимо добавить еще один атрибут: `id="ng-app"`.

IE8 не будет распознавать нестандартные HTML-теги, если не предпринять дополнительных действий. Например, мы не сможем подключать шаблоны, как показано ниже, пока не научим IE8 распознавать тег `<ng-include>`:

```
<ng-include="'myInclude.tpl.html'"></ng-include>
```

Сделать это можно, предварительно создав свой элемент DOM, как показано в следующем фрагменте:

```
<head>
  <!--[if lte IE 8]>
    <script>
      document.createElement('ng-include');
      document.createElement('ng-view');
      ...
  <![endif]-->
</head>
```

Нестандартные элементы должны создаваться в сценарии, находящемся в разделе `<head>` страницы, чтобы обеспечить его выполнение до того, как парсер браузера обнаружит элемент неизвестного ему типа.

Разумеется, от проблемы нестандартных тегов можно полностью избавиться, применив версию директивы-атрибута `ng-include`:

```
<div ng-include="'myInclude.tpl.html'"></div>
```

Поддержка IE7 выглядит еще более проблематичной. Для начала необходимо будет предусмотреть те же действия, что и для версии IE8. Но беда в том, что IE7 не используется в системе тестирования, поэтому нет никаких гарантий, что все функциональные возможности AngularJS будут поддерживаться в этом браузере. Более того, в IE7 отсутствуют многие прикладные интерфейсы, присутствующие

в современных браузерах. Примером таких интерфейсов может служить JSON API, не реализованный в IE7, из-за чего требуется загружать заменяющую его библиотеку (http://bestiejs.github.io/json3/).

Версия IE6 вообще не поддерживается, ни в каком виде.

В документации к фреймворку AngularJS имеется исчерпывающий раздел, описывающий все шаги, необходимые для поддержки IE 8 и IE 7: http://docs.angularjs.org/guide/ie.

# В заключение

В этой главе мы рассмотрели некоторые вопросы, связанные с развертыванием приложений на основе AngularJS. В начале главы были рассмотрены некоторые приемы оптимизации сетевых взаимодействий. В общем случае можно уменьшить время, затрачиваемое браузерами на выполнение сетевых операций, уменьшив объем передаваемых по сети данных и сократив количество запросов.

Часто для уменьшения объема загружаемых данных применяется прием минификации кода HTML, CSS и JavaScript. Особенности механизма внедрения зависимостей в AngularJS вынуждают нас использовать в коде JavaScript дополнительные приемы поддержки минификации. На практике это означает необходимость использования аннотаций в виде массивов во всех функциях, объявляющих зависимости, которые разрешаются подсистемой DI в AngularJS.

Чтобы ограничить количество запросов, выполняемых браузером, можно объединять шаблоны и организовать их предварительную загрузку. В этой главе мы познакомились с двумя разными приемами предварительной загрузки шаблонов: с помощью директивы `<script>` и обращением к службе `$templateCache`.

AngularJS – это фреймворк, используемый для создания динамических страниц в браузере и прочно основанный на JavaScript. Он может создавать разметку HTML на стороне клиента, только после загрузки в браузер всех необходимых библиотек JavaScript. Недостатком такой организации является возможность для пользователя увидеть необработанную начальную страницу приложения. Для устранения этого нежелательного эффекта можно использовать две директивы AngularJS: `ng-cloak` и `ng-bind`. Директива `ng-cloak` позволяет скрывать большие фрагменты дерева DOM, и ее удобно использовать в приложениях, когда начальная страница имеет большой

объем динамического и незначительный объем статического содержимого. Директива `ng-bind`, напротив, лучше подходит для использования в начальных страницах, состоящих преимущественно из статического содержимого.

Стремясь улучшить впечатления пользователя от начальной страницы приложения, необходимо так же подумать о порядке загрузки различных элементов страницы. Размещение ссылок на файлы JavaScript в самом конце страницы считается хорошей практикой, но ценность такого приема в высокодинамичных веб-приложениях может оказаться невысокой. С другой стороны, если начальная страница состоит в основном из статического содержимого и имеет значительный объем разметки HTML, перенос тегов `<script>` в конец страницы может принести некоторые выгоды. Загрузка сценариев внутри тега `<head>` в динамичных страницах часто дает лучшие результаты, особенно в сочетании с директивой `ng-cloak`.

В этой главе мы также кратко коснулись проблемы асинхронной загрузки сценариев. Несмотря на высокую популярность приемов асинхронной загрузки в наши дни, их ценность для приложений на основе AngularJS оказывается недостаточно высокой. Следует признать, что текущая версия AngularJS плохо поддерживает асинхронную загрузку сценариев, в особенности это относится к прикладному коду. Учитывая ограниченную ценность и сложность настройки, мы советуем не использовать приемы асинхронной загрузки в приложениях на основе AngularJS.

Если вы собираетесь предоставить доступ к своему приложению для широких масс пользователей, вы должны позаботиться о поддержке разных браузеров. Приятно отметить, что сам фреймворк AngularJS обеспечивает безупречную поддержку всех основных браузеров. Для поддержки Internet Explorer придется предпринять дополнительные действия, которые обсуждались в конце данной главы.

# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

## $


$anchorScroll, служба 202;
 навигация 202
$anchorScrollProvider, служба 203
$attrs 298
$compile, служба;
 создание функции включения 291
$dialog, служба 235
$digest, цикл обработки событий 337;
 увеличение скорости 346
$element 298
$exceptionHandler, служба 340
$fieldErrors, свойство 309
$filter, служба 163
$first, переменная 144
$http, служба 99, 242;
 Promise API 120;
 взаимодействие с конечными точками REST 128;
 дополнительные возможности 132;
 обзор 100;
 прикладной интерфейс 100, 102;
 тестирование 133
$http.jsonp, метод 105
$http.post, метод 102
$http.put, метод 102
$httpBackend, служба 134
$httpBackend, фиктивный объект 134
$index, переменная 144, 145
$injector, встроенная служба 46
$injector, служба 364
$interpolate, служба 308, 321
$interpolateProvider, объект 139
$last, переменная 144
$locale.id, переменная 315
$location, служба 197, 202, 221, 325;
 навигация вручную 205;
 прикладной интерфейс 201;
 применение 200
$location.hash(), метод 202
$location.path(), компонент 207
$locationChangeSuccess, событие 36
$locationProvider, служба 221
$middle, переменная 144
$on, метод 36
$parse, служба 332
$provide, встроенная служба 46
$provide, служба 327
$q, служба 110;
 Promise API 109;
 интеграция в AngularJS 119;
 основы 110
$q.all, метод 117, 119
$q.defer, метод 111
$q.when, метод 117, 119
$resource, служба 122, 123;
 ограничения 128
$rootScope.$digest, метод 112
$route, служба 208;
 обработка нескольких элементов пользовательского интерфейса с помощью директивы ng-include 217, 218;
 ограничения 216
$routeParams, служба 211
$routeProvider, провайдер службы 209
$routeProvider, служба 247
$scope, экземпляр 44
$templateCache, служба 229, 369;
 наполнение 370
$transclude 298
$transclude, параметр 293
$watch, метод 333, 338
$watch, функция 266


## <


<script>, директива;
 и предварительная загрузка 369

## А

автоматическое тестирование 62, 84, 86
авторизация на стороне клиента;
  обработка ошибок авторизации 241;
  перехват ответов 242;
  поддержка 240;
  служба securityInterceptor 242
авторизация на стороне сервера;
  обработка неавторизованного доступа 227;
  реализация 227
аккордеон, виджет 301, 303;
  использование контроллера директивы 302;
  реализация 303
аннотации зависимостей;
  недостатки 367, 368
аргументы функций;
  минификация 364
асинхронная проверка модели 275;
  реализация 278;
  тестирование 276
асинхронное программирование 91
асинхронные операции;
  объединение в цепочки 114
атака через посредника;
  предотвращение 230
аутентификация на стороне клиента;
  поддержка 240
аутентификация на стороне сервера;
  прикладной интерфейс 228;
  реализация 228

## Б

безопасность;
  приложений 226;
  шаблонов разметки 228
безопасность на стороне клиента 235;
  добавление поддержки 235;
  отображение формы аутентификации 236;
  создание меню 238;
  создание панелей инструментов 238;
  создание службы security 236;
  сокрытие элементов меню 238
библиотеки 24

## В

веб-компоненты, стандарт 60
веб-приложения 330
взаимодействие объектов 44
включение;
  if, директива;
    создание 294;
  в директивах с изолированным контекстом 286;
  использование в директивах 285
вложенные формы 189;
  повторение вложенных форм 189;
  проверка полей повторяющихся форм 191
внедрение данных в формате JSON;
  предотвращение 233
внедрение зависимостей 44, 299, 363
выражения-генераторы 176
выражения AngularJS;
  защита содержимого 231;
  небезопасное отображение разметки HTML 232;
  разметка HTML 139, 140
выражения связывания 178

## Г

генераторы данных 178

## Д

данные;
  двунаправленное связывание в AngularJS 27
двунаправленное связывание данных 27
декларативное представление шаблонов 38
динамическая загрузка шаблонов 310
директива-обертка для виджета jQueryUI 279;
  реализация 282;
  тестирование 281
директива постраничного просмотра 261
директивы;
  включение 285;
  знакомство 137;
  использование 319;

использование свойства priority 296;
использование функций включения 292;
контроллеры директив 296;
виджет аккордеон 301;
внедрение специальных зависимостей 297;
доступ к другим контроллерам 300;
использование свойства priority 297;
и функции связывания 299;
процедура компиляции 299;
реализация виджета аккордеон 303;
определение 367
директивы AngularJS 251;
compile, поле 257;
controller, поле 257;
link, поле 257;
name, поле 257;
priority, поле 257;
replace, поле 257;
require, поле 257;
restrict, поле 257;
scope, поле 257;
template, поле 257;
templateUrl, поле 257;
transclude, поле 257;
виджеты 261;
использование шаблонов 263;
реализация 267;
в разметке HTML 253;
встроенные 252;
изолированные от родительского контекста 264;
контроллер директивы 270;
внедрение 270;
необязательный 271;
поиск родительского 271;
определение 257;
оформление кнопок 258;
порядок компиляции 253;
проверки 269;
тестирование 255
директивы ввода 173;
кнопки-флажки 175;
проверка обязательного наличия значения 174;
радиокнопки 175;
текстовые элементы ввода 174;
элементы выбора из списка 176
директивы интерполяции 138
доступ к вильтрам из кода на JavaScript 163

## Ж

жизненный цикл отложенных заданий 114

## З

защищенные маршруты;
предотвращение переходов 246
значения моделей;
отображение с помощью ngBind 139

## И

избыточный синтаксис 150
интеграционные тесты 93;
отладка 97
интерполяция атрибутов с префиксом @ 265

## К

каталоги верхнего уровня 74
каталоги с исходными текстами 75
каталоги с тестами 78
кнопки-флажки 175
компиляция;
динамическая загрузка шаблонов 310;
управление в директиве field 305
константы 49
контекст;
включения 288
контексты 29;
дополнительные подробности 30;
иерархии 30;
наследование 33
контроллер директивы 270;
внедрение 270;
необязательный 271;
поиск родительского 271
контроллеры 29, 78;
директив 296;
виджет аккордеон 301;
внедрение специальных зависимостей 297;

доступ к другим контроллерам 300;
использование свойства priority 297;
и функции связывания 299;
процедура компиляции 299;
реализация виджета аккордеон 303;
создание директивы постраничного просмотра 298;
тестирование 89

## Л

локаль;
в адресе URL 323

## М

маршруты;
вложенные 218;
гибкое сопоставление 210;
ограничения 216;
определение 208;
организация определений 222;
отображение в адреса URL 206;
отображение содержимого 209;
повторное использование шаблонов с разными контроллерами 212;
по умолчанию 211;
предотвращение изменения 215;
предотвращение мерцания пользовательского интерфейса 213;
ссылки на внешние страницы 222
массивы;
как источники данных 177
межсайтовый скриптинг (Cross-Site Scripting, XSS);
предотвращение 231
методы конструктора;
и методы экземпляра 124;
собственные методы 125
методы экземпляра;
и методы конструктора 124
минификация;
и аргументы функций 364;
кода на JavaScript 365;
статических ресурсов 363
модели 30;
обновление в ответ на события DOM 332;
передача изменений в DOM 333;
синхронизация с деревом DOM 333, 335;
стабильность 338, 340
модули 42, 80, 366;
блоки, объявление 83;
блоки выполнения 83;
блоки настройки 83;
видимость служб 53;
внедрение зависимостей 42;
жизненный цикл 51;
зависимости 52;
организация 80;
преимущества 56;
регистрация провайдеров 81;
с национальными настройками 314;
и фильтры 315;
фаза выполнения 51;
фаза настройки 51
модульные тесты 86;
тестирование асинхронного кода 91;
тестирование контроллеров 89;
тестирование объектов 87;
тестирование служб 88;
фиктивные объекты 91

## Н

навигация вручную 205;
контроллеры шаблонов 207;
недостающие мелочи 208;
отображение маршрутов в адреса URL 206;
структурирование страниц на основе маршрутов 205
нападения;
внедрение данных в формате JSON 233;
подделка межсайтовых запросов 234;
противостояние 229
настройка производительности 343;
вызывайте scope.$digest вместо scope.$apply 352;
и директива ng-repeat 358;
избегайте глубокого сравнения 356;
избегайте операций с деревом DOM в выражениях 348;
исключите выражения в невидимых элементах 351;

обеспечьте высокую скорость вычисления выражений 346;
ограничивайте количество итераций в цикле $digest 355;
оптимизация потреблении памяти 356;
реже выполняйте цикл $digest 353;
удалите ненужные выражения 349;
учитывайте размеры выражений 358
начальная страница;
оптимизация 373;
сокрытие отдельных выражений 375;
сокрытие фрагментов DOM 374
нестабильные фильтры 166

## О

обработка ссылок 220
обработчики событий 149
объединение асинхронных операций 114
объединение функций обратного вызова 113
объект с настройками, пример 101;
cache, свойство 102;
headers, свойство 101;
method, свойство 101;
params, свойство 101;
timeout, свойство 101;
transformRequest, свойство 102;
transformResponse, свойство 102;
url, свойство 101
объекты;
взаимодействие 44;
значения 47;
как источники данных 177
объекты AngularJS 87;
фиктивные 91
объекты отложенных заданий 111
ограничения политики общего происхождения 104
определение функций обратного вызова в атрибутах с префиксом & 267
определения маршрутов;
повторяющийся код 223;
распределение по нескольким файлам 222
оптимизация потреблении памяти 356
организация определений маршрутов 222
отключение процедуры проверки, встроенной в браузер 188
отложенные задания 111;
жизненный цикл 114
отображение единственной строки с дополнительной информацией 146

## П

панель управления сеансом;
создание 239
перевод строк;
в коде JavaScript 321
перевод шаблонов;
на этапе сборки 320
передача репрезентативного состояния (Representational State Transfer, REST), архитектурное решение 121
переключение между национальными настройками 325
перехват cookie;
предотвращение 230
перехватчики ответов 242
подделка межсайтовых запросов;
предотвращение 234
поддержка переводов 317
подсчет элементов после фильтрации 158
политика общего происхождения (Same-Origin Security Policy) 104;
ограничения 104;
преодоление ограничений 104, 106
Порядок компиляции директив 253
порядок компиляции директив 253
предварительная загрузка;
директива <script> 369;
комбинирование приемов 372;
шаблонов 368
представления 37
преобразование данных запроса 102
преобразование данных ответа 103
преобразование моделей с помощью фильтров 153
преобразования данных в фильтрах 165
приложения;
безопасность 226
пример приложения 63
провайдеры 50, 366;
регистрация в модулях 81

проксирование запросов на стороне сервера 108
противостояние нападениям 229
пустые пункты в директиве select 179

## Р

радиокнопки 175
разработка через тестирование (Test Driven Development, TDD) 257
расширение возможностей объектов ресурсов 126
расширения 24
реализация директивы-обертки для виджета jQueryUI 282
реализация директивы асинхронной проверки 278
реализация директивы проверки 274
регистрация провайдеров в модулях 81
регистрация функций обратного вызова 114
рецепты, создания объектов 46

## С

связывание данных в атрибутах с префиксом = 266
серверное окружение 66
серверы непрерывной интеграции 379
сетевые операции;
 увеличение производительности 363
сеть доставки содержимого (Content Delivery Network, CDN) 26
система сборки 68;
 принципы построения 69
скрытые поля ввода 181
службы;
 регистрация 46;
 тестирование 88
службы маршрутизации 208;
 гибкое сопоставление маршрутов 210;
 ограничения 216;
 определение 208;
 отображение содержимого 209;
 повторное использование шаблонов с разными контроллерами 212;
 предотвращение изменения маршрута 215;
 предотвращение мерцания пользовательского интерфейса 213;
 советы 220;
 ссылки, реагирующие на щелчок мышью 220;
 ссылки на внешние страницы 222
события;
 ввода 149;
 клавиатуры 149;
 мыши 149;
 щелчка мышью 149
соглашения по именованию файлов 79
создание директивы проверки 269
создание собственных фильтров 161
специальные переменные 144
ссылки;
 обработка 220;
 реагирующие на щелчок мышью 220
статические ресурсы;
 минификация 363
сторонние библиотеки JavaScript 68
сценарии;
 ссылки 376

## Т

текстовые элементы ввода 174
тестирование;
 автоматическое 62;
 директивы-обертки для виджета jQueryUI 281;
 директивы pagination 262;
 директивы асинхронной проверки 276;
 директивы проверки 272;
 интеграционное 93; отладка 97
тестирование асинхронного кода 91

## Ф

фабрики 48
файлы;
 соглашения по именованию 79
файлы и каталоги 73;
 каталоги верхнего уровня 74;
 каталоги с исходными текстами 75;
 каталоги с тестами 78;
 организация 73
фиктивные объекты 91

фильтры 153;
встроенные 154;
доступ из кода на JavaScript 163;
и дорогостоящие преобразования 165;
и операции с деревом DOM 164;
нестабильные 166;
подсчет элементов после фильтрации 158;
постраничного просмотра 161;
правила использования 164;
преобразование моделей 153;
преобразования массивов 155;
собственные 161;
создание собственных фильтров 161;
сортировка с помощью фильтра orderBy 159;
форматирования 154
формы;
вложенные 189;
отправка 192;
повторение вложенных форм 189;
проверка полей повторяющихся форм 191;
сброс в исходное состояние 194;
события отправки 193
формы AngularJS 169;
вложенные 189;
вывод сообщений об ошибках 186;
обзор 169;
отправка 192;
повторение вложенных форм 189;
проверка 184;
проверка полей повторяющихся форм 191;
сброс в исходное состояние 194;
события отправки 193
функции включения 291;
создание с помощью службы $compile 291
функции обратного вызова 103;
объединение 113;
регистрация 114

## Ш

шаблоны;
MVC 27;
безопасность 228;
динамическая загрузка 310;
комбинирование приемов предварительной загрузки 372;
предварительная загрузка 368
шаблоны AngularJS;
перевод строк 318
шаблоны на основе DOM 150

## Э

элементы выбора из списка 176

## A

Accept-Language, заголовок запроса 323
accordion, директива;
реализация 303
accordion-group, директива;
реализация 304
action, атрибут 193
alert, директива;
создание 286
anchorScroll, служба 197
angular.callbacks._k, функция 105
AngularJS;
MVC, шаблон 27;
двунаправленное связывание данных 27;
знакомство 22;
и jQuery 58;
инструменты 24;
константы 49;
контексты 29;
дополнительные подробности 30;
иерархии 30;
наследование 33;
контроллеры 29;
модели 30;
модули 42;
видимость служб 53;
жизненный цикл 51;
зависимости 52;
преимущества 56;
фаза выполнения 51;
фаза настройки 51;
представления 37;
пример 25;
провайдеры 50;

службы;
регистрация 46;
ускоренное введение 25
AngularJS, директивы 251;
виджеты 261;
использование шаблонов 263;
реализация 267;
в разметке HTML 253;
встроенные 252;
изолированные от родительского контекста 264;
контроллер директивы 270;
внедрение 270;
необязательный 271;
поиск родительского 271;
определение 257;
оформление кнопок 258;
порядок компиляции 253;
проверки 269;
тестирование 255
AngularJS, формы 169;
вложенные 189;
вывод сообщений об ошибках 186;
обзор 169;
отправка 192;
повторение вложенных форм 189;
проверка 184;
проверка полей повторяющихся форм 191;
сброс в исходное состояние 194;
события отправки 193
AngularJS, фреймворк 363;
$digest, цикл обработки событий 337;
внутренние механизмы 331;
иерархия контекстов 341;
и определение асинхронных модулей 377;
настройка производительности 343, 345;
обновление модели 332;
передача изменений из модели в DOM 333;
поддержка браузеров 379;
подключение прикладных сценариев 376;
синхронизация дерева DOM и модели 333, 335;
стабильность моделей 338, 340
authorization, служба 248;
создание 248

## B

Batarang, дополнение к браузеру Chrome 24
Batarang, расширение для Chrome 344
beforeEach, функция 87
beHungry, метод 111
Bootstrap CSS 68
button, директива 258

## C

cache, свойство 102
cancel(), метод 245
cancelAll(), метод 245
cancelLogin(redirectTo) , метод 236
canSave(), метод 187
CDN (Content Delivery Network сеть доставки содержимого) 26
clicked(), выражение 149
closeLoginDialog()? вспомогательный метод 237
compile, поле 257
compile, функция;
функции включения 292
constant, метод 49
controller, поле 257
controller, свойство;
в определениях директив 297
CORS 106;
преодоление ограничений политики общего происхождения 106
currency, фильтр 155, 316
currentUser, свойство 236

## D

date, фильтр 155, 315, 326
datepicker, директива 279
describe, функция 87
directive(), метод 257

## E

eat, метод 111
errorCallBack, метод 112
errorcb, аргумент 124

expect, функция 87
Express, фреймворк 67

## F

factory, метод 48
field, директива;
 настройка шаблона 310;
 создание 306
filter, фильтр 155
flush, метод 135
fullDate, формат 327

## G

GET /current-user, запрос 228
getCssClasses(), метод 187
getLoginReason(), метод 236
getName(), выражение 339
Grunt.js, инструмент сборки 72

## H

headers, свойство 101
href, атрибут 220, 325
HTML, разметка;
 в выражениях AngularJS 139, 140;
 и директивы 253
HTML5, стандарт;
 интерпретации адресов URL 203;
 на стороне клиента 203;
 на стороне сервера 204;
 настройка 203;
 интерфейс истории посещений 199
HTTP-ответы;
 обработка 103
https, модуль 230
HTTPS, протокол 230

## I

i18n, фильтр 318
if, директива;
 создание 294
IInternet Explorer, браузер 380
it, функция 87

## J

Jasmine, инструмент тестирования 72
JavaScript, код;
 минификация 365
jsFiddle, инструмент 25
json, фильтр 155
JSON_CALLBACK, параметр
 запроса 105
JSONP, запросы;
 выполнение с помощью службы
  $http 99;
 ограничения 105;
 преодоление ограничений политики
  общего происхождения 104

## K

Karma runner, инструмент
 тестирования 73

## L

limitTo, фильтр 155
link, поле 257
loadTemplate, функция 310
login(email, password) , метод 236
login-toolbar, директива 239
logout(redirectTo) , метод 236
lowercase, фильтр 155

## M

method, свойство 101
modelChangeCallback, параметр 338
MongoLab;
 знакомство 66;
 модель адресов URL 100;
 модель данных 100
Mustache, система поддержки
 шаблонов;
 URL 331
MVC, шаблон 27

## N

name, поле 257
new, ключевое слово 46
ng-app, атрибут 44
ng-bind, директива 139, 231, 375
ng-bind-html, директива 140, 232, 233
ng-bind-html-unsafe, директива 140, 232
ng-class, директива 187

ng-click, директива 220
ng-cloak, директива 374
ng-controller, директива 29, 212
ng-hide, семейство директив 141
ng-if, директива 142
ng-include, директива 142, 207, 218
ng-init, атрибут 29
ng-model, директива 270, 332, 336
ng-repeat, директива 143, 145, 260, 296;
легко создает множество привязок 359;
наблюдение за коллекцией 359;
проблемы производительности 358
ng-show, семейство директив 141
ng-switch, директива 141
ng-transclude, директива 288
ngBind, директива;
отображение значений моделей 139
ngChange, директива 149
ngClass, директива 148
ngClassEven, директива 148
ngClassOdd, директива 148
ngClick, директива 149, 194
ngDblClick, директива 149
ngDisabled, директива 187
ngFormController, контроллер 184;
использование атрибута name 185
ngKeydown, директива 149
ngKeypress, директива 149
ngKeyup, директива 149
ngLocale, модуль 315, 323
ngMaxLength, директива 174
ngMinLength, директива 174
ngModel, директива 138, 171
ngModel.$render(), метод 280
ngModelController, контроллер 182;
$error 272;
$valid 272;
конвейер передачи данных 182
ngMousedown, директива 149
ngMouseenter, директива 149
ngMouseleave, директива 149
ngMousemove, директива 149
ngMouseover, директива 149
ngMouseup, директива 149
ngMultiple, директива 181
ngOptions, директива 176
ngPattern, директива 174
ngSanitize, модуль 140, 232, 233
ngSubmit, директива 193
novalidate, атрибут 188
number, фильтр 155, 317

## O

Object.observe, стандарт 59
openLoginDialog(), вспомогательный метод 237
orderBy, фильтр 155, 159;
сортировка 159

## P

pagination, директива 261;
тестирование 262
pagination, фильтр 162
params, аргумент 124
params, свойство 101
Passport, компонент поддержки безопасности 67
payloadData, аргумент 124
Plunker, инструмент 25
POST /login, запрос 228
POST /logout, запрос 228
priority, поле 257
priority, свойство;
в определениях директив 296
Promise API;
и служба $http 120;
и служба $q 109;
обзор 109, 120
provider, метод 50
pushRetryFn(), метод 245

## Q

Q Promise API, библиотека 110

## R

reject, метод 112
remaining(), функция 39
replace, поле 257
replace, свойство;
в определениях директив 287
require, поле 257
requireAdminUser(), метод 249
requireAuthenticatedUser(), метод 248

resolve, метод 112
resolve, свойство 214, 247
respond, метод 135
REST (Representational State Transfer передача репрезентативного состояния), архитектурное решение 121
REST API 100
Restler, клиентская библиотека 67
restrict, поле 257
retry(), метод 245
retryAll(), метод 245
reverse, свойство 160
routes, объект 206

## S

Scenario Runner, решение интеграционного тестирования 93
scope, поле 257
scope.$apply, метод 335, 337
SCRUM, методология 62, 63
security, служба 235, 236
securityInterceptor, служба 242
securityRetryQueue, служба 244;
создание 244
select, директива 180
service, метод 47, 48
showLogin(), метод 236
simple-bind, директива 333
simple-model, директива 332
sortField, свойство 160
successcb, аргумент 124
switchLocaleUrl, функция 326

## T

template, поле 257
template, свойство 217
templates, модуль 371
templateUrl, поле 257
templateUrl, свойство 217, 369
terminal, свойство;
в определениях директив 308
timeout, свойство 101
toEqual, функция 87
transclude, поле 257
transclude, свойство;
в определениях директив 288
transcludeFn, параметр 292
transformRequest, свойство 102
transformResponse, свойство 102

## U

uppercase, фильтр 155
URL, адреса 197, 201;
в одностраничных веб-приложениях 197;
в эпоху до HTML5 198
url, свойство 101
user.password, поле модели 174
user.role, поле модели 175
Users.delete, метод 124
Users.get, метод 124
Users.query, метод 124
Users.save, метод 124

## V

value, метод 47
verifyNoOutstandingExpectation, метод 135
verifyNoOutstandingRequest, метод 135

## W

watchExpression, параметр 334
whenGET, метод 135

## X

X-XSRF-TOKEN, заголовок 235
XMLHttpRequest (XHR), объект 99
XSRF-TOKEN, параметр 235
XSS (Cross-Site Scripting межсайтовый скриптинг);
предотвращение 231

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» 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**.

Павел Козловский
Питер Бэкон Дарвин

# Разработка веб-приложений с использованием AngularJS

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 12.11.2013. Формат 60 90 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 24,13. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

С появлением HTML5 и CSS3 разработка клиентских веб-приложений на языке JavaScript приобрела особую популярность. Создатели фреймворка AngularJS предприняли революционный подход к решению вопроса превращения браузера в самую лучшую платформу для разработки веб-приложений.

Книга «Разработка веб-приложений с использованием AngularJS» проведет вас через основные этапы конструирования типичного одностраничного веб-приложения. В ней обсуждаются такие темы, как организация структуры приложения, взаимодействие с различными серверными технологиями, безопасность, производительность и развертывание.

После представления фреймворка AngularJS и обзора перспектив клиентских веб-приложений, эта книга шаг за шагом проведет вас через создание достаточно сложного приложения. В каждой главе рассматривается одна из основных тем, с которыми вам придется столкнуться при разработке приложений, более сложных, чем простые демонстрационные примеры.

## Кому адресована эта книга

Эта книга будет наиболее полезна веб-разработчикам, желающим оценить или решившим применить фреймворк AngularJS для создания своих приложений. Предполагается, что читатель имеет некоторое знакомство с AngularJS, хотя бы на уровне понимания простейших примеров. Мы надеемся также, что вы обладаете знанием HTML, CSS и JavaScript.

**В этой книге вы:**

- Познакомитесь с особенностями поддержки шаблонов в AngularJS и отличиями от других фреймворков.
- Научитесь запрашивать и изменять данные с использованием различных серверных технологий и овладеете искусством использования Promise API.
- Узнаете, как быстро создавать сложные формы, используя преимущества двунаправленного связывания данных.
- Познакомитесь с приемами организации навигации в своих веб-приложениях, опираясь на History API в HTML5.
- Научитесь управлять зависимостями с использованием систем поддержки модулей и внедрения зависимостей в AngularJS.
- Узнаете, как закрывать свои приложения от неавторизованного доступа.
- Познакомитесь с возможностями модульного тестирования кода на JavaScript с использованием фреймворка Jasmine BDD.
- Научитесь создавать виджеты, валидаторы и собственные директивы, а также увидите, как можно управлять компилятором AngularJS.

[PACKT] open source* PUBLISHING community experience distilled

ДМК ПРЕСС издательство
www.дмк.рф

Internet-магазин: www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)725-5409, books@alians-kniga.ru

ISBN 978-5-97060-064-1