# 7 Шаблонизация с помощью Handlebars

Если вы не используете шаблонизацию или вообще не знаете, что такое шаблонизация, то знайте: это и есть самое важное, что вы вынесете из этой книги. Если вы раньше работали с PHP, то можете удивиться, из-за чего весь этот ажиотаж: PHP — один из первых языков, который может быть назван языком шаблонизации. Практически все основные языки, приспособленные к веб-разработке, включают тот или иной вид поддержки шаблонизации. Но сейчас ситуация изменилась: *шаблонизатор* обычно расцеплен с языком. Один из примеров этого — Mustache, исключительно популярный языково-независимый шаблонизатор.

Так что же такое *шаблонизация*? Начнем с того, чем шаблонизация **не является**, и рассмотрим наиболее прямой и очевидный путь генерации одного языка из другого (конкретнее, мы будем генерировать HTML с помощью JavaScript):

```
document.write('<h1>Пожалуйста, не делайте так</h1>');
document.write('<p><span class="code">document.write</span> капризен,\n');
document.write('и его следует избегать в любом случае.</p>');
document.write('<p>Сегодняшняя дата: ' + new Date() + '.</p>');
```

Возможно, единственная причина, по которой это кажется очевидным, — то, что именно так всегда учили программированию:

```
10 PRINT "Hello world!"
```

В императивных языках мы привыкли говорить: «Сделай это, затем сделай то, а затем сделай что-то еще». Для некоторых вещей этот подход отлично работает. В нем нет ничего плохого, если у вас 500 строк JavaScript для выполнения сложного вычисления, результатом которого является одно число, и каждый шаг зависит от предыдущего. Но что, если дела обстоят с точностью до наоборот? У вас 500 строк HTML и три строки JavaScript. Имеет ли смысл писать `document.write` 500 раз? Отнюдь.

На самом деле все сводится к тому, что переключать контекст проблематично. Если вы пишете много кода JavaScript, вплетать в него HTML неудобно и это приводит к путанице. Противоположный способ не так уж плох: мы привыкли писать JavaScript в блоках `<script>`, но надеюсь, что вы видите разницу: тут все-таки

есть переключение контекста и вы или пишете HTML, или в блоке <script> пишете JavaScript. Использование же JavaScript для генерации HTML чревато проблемами.

- Вам придется постоянно думать о том, какие символы необходимо экранировать и как это сделать.
- Использование JavaScript для генерации HTML, который, в свою очередь, сам содержит JavaScript, очень быстро сводит вас с ума.
- Вы обычно лишаетесь приятной подсветки синтаксиса и прочих удобных возможностей, отражающих специфику языка, которыми обладает ваш редактор.
- Становится намного сложнее заметить плохо сформированный HTML.
- Сложно визуально анализировать код.
- Другим людям может оказаться труднее понимать ваш код.

Шаблонизация решает проблему, позволяя вам писать на целевом языке и при этом обеспечивая возможность вставлять динамические данные. Взгляните на предыдущий пример, переписанный в виде шаблона Mustache:

```
<h1>Намного лучше</h1>
<p>Никаких <span class="code">document.write</span> здесь!</p>
<p>Сегодняшняя дата {{today}}.</p>
```

Все, что нам осталось сделать, — обеспечить значение для {{today}};. Это и есть основа языков шаблонизации.

# Нет абсолютных правил, кроме этого[1]

Я не говорю, что вам **никогда** не следует писать HTML в JavaScript — только что следует избегать этого где только возможно. В частности, это несколько уместнее в коде клиентской части благодаря таким библиотекам, как jQuery. Например, следующее вызвало бы с моей стороны совсем немного комментариев:

```
$('#error').html('Случилось что-то <b>очень плохое!</b>');
```

Однако я бы намекнул, что пришло самое время применить шаблон, если оно постепенно видоизменится до вот такого:

```
$('#error').html('<div class="error"><h3>Ошибка</h3>' +
    '<p>Случилось что-то <b><a href="/error-detail/' + errorNumber
    +'"> очень плохое.</a></b> ' +
    '<a href="/try-again">Попробуйте снова<a>, или ' +
    '<a href="/contact">обратитесь в техподдержку</a>.</p></div>');
```

[1] Перефразируя моего друга Пола Инмана.

Дело в том, что я посоветовал бы вам как можно тщательнее обдумать, где провести границу между HTML в строках и использованием шаблонов. Лично я допустил бы перекос в сторону шаблонов и избегал генерации HTML с помощью JavaScript во всех случаях, за исключением простейших.

# Выбор шаблонизатора

Во вселенной Node у вас есть выбор из множества шаблонизаторов. Как же выбрать нужный? Это непростой вопрос, в значительной степени зависящий от того, что именно вам требуется. Вот некоторые критерии, которые следует принять во внимание.

- **Производительность.** Несомненно, вы хотите, чтобы шаблонизатор работал как можно быстрее. Не годится, чтобы он замедлял работу вашего сайта.
- **Клиент, сервер или и то и другое?** Большинство, хотя и не все, шаблонизаторов доступны на стороне как сервера, так и клиента. Если вам необходимо использовать шаблоны и тут и там (и вы будете это делать), я рекомендую выбрать шаблонизатор, равно производительный в обоих случаях.
- **Абстракция.** Хочется ли вам чего-то знакомого (вроде обычного HTML с добавлением фигурных скобок), или вы тайно ненавидите HTML и предпочли бы что-то без всех этих угловых скобок? Шаблонизация (особенно шаблонизация на стороне сервера) предоставляет вам возможность выбора.

Это только некоторые из наиболее важных критериев выбора шаблонизатора. Если вам хотелось бы более подробного обсуждения данного вопроса, я очень советую почитать в блоге Вины Басаварадж (http://bit.ly/templating_selection_criteria) о ее критериях выбора шаблонизатора для LinkedIn.

Для LinkedIn оказался выбран Dust, однако мой любимый шаблонизатор Handlebars также был в числе финалистов, и именно его мы будем использовать в этой книге.

Express позволяет использовать любой шаблонизатор, какой вы только пожелаете, так что, если Handlebars вас не устраивает, вы без проблем сможете его заменить. Если хотите узнать, какие существуют варианты, можете попробовать вот эту забавную и удобную утилиту выбора шаблонизатора: http://garann.github.io/template-chooser.

# Jade: другой подход

В то время как большинство шаблонизаторов использует подход с сильной ориентацией на HTML, Jade выделяется, абстрагируя от вас его подробности. Стоит также отметить, что Jade — детище Ти Джея Головайчука, того самого, кто подарил

нам Express. Ничего удивительного, что Jade отлично интегрируется с Express. Используемый Jade подход весьма благороден: в его основе лежит утверждение, что HTML — слишком перегруженный деталями и трудоемкий для написания вручную язык. Посмотрим, как выглядит шаблон Jade вместе с результирующим HTML (взято с домашней страницы Jade и слегка изменено для соответствия книжному формату):

```
doctype html
html(lang="ru")
  head
    title= pageTitle
    script.
      if (foo) {
        bar(1 + 5)
      }
  body

    h1 Jade
    #container
      if youAreUsingJade
        p Браво!
      else
        p Просто сделайте это!
      p.
        Jade сжатый
        и простой
        язык шаблонизации
        с сильным акцентом на
        производительности
        и многочисленных возможностях.
<!DOCTYPE html>
<html lang="ru">
<head>
<title>Демонстрация Jade</title>
<script>
  if (foo) {
    bar(1 + 5)
  }
</script>
<body>
<h1>Jade</h1>
<div id="container">
<p>Браво!</p>
<p>
  Jade сжатый и простой
```

```
    язык шаблонизации
    с сильным акцентом на
    производительности
    и многочисленных возможностях.
</p>
</body>
</html>
```

Jade, безусловно, требует намного меньше набора на клавиатуре: больше никаких угловых скобок и закрывающих тегов. Вместо этого Jade опирается на структурированное расположение текста и определенные общепринятые правила, облегчая вам выражение своих идей. У Jade есть и еще одно достоинство: теоретически, когда меняется сам язык HTML, вы можете просто перенастроить Jade на новую версию HTML, что обеспечивает вашему контенту «защиту от будущего».

Как бы я ни восхищался философией Jade и изяществом его выполнения, я обнаружил, что не хочу, чтобы подробности HTML абстрагировались от меня. HTML лежит в основе всего, что я как веб-разработчик делаю, и если цена этого — износ клавиш угловых скобок на моей клавиатуре — так тому и быть. Множество разработчиков клиентской части, с которыми я обсуждал это, думают аналогично, так что мир, возможно, просто еще не готов к Jade...

На этом мы расстаемся с Jade, больше вы его в этой книге не увидите. Однако, если абстракция вам нравится, у вас определенно не будет проблем при использовании Jade с Express, и существует масса ресурсов, которые вам в этом помогут.

# Основы Handlebars

Handlebars — расширение Mustache, еще одного распространенного шаблонизатора. Я рекомендую Handlebars из-за его удобной интеграции с JavaScript (как в клиентской, так и в прикладной части) и знакомого синтаксиса. По моему мнению, он обеспечивает все правильные компромиссы, и именно на нем мы сосредоточимся в данной книге. Однако концепции, которые мы будем обсуждать, легко применимы и к другим шаблонизаторам, так что вы будете вполне готовы к тому, чтобы попробовать другие шаблонизаторы, если Handlebars придется вам не по вкусу.

Ключ к пониманию шаблонизации — понимание концепции *контекста*. Когда вы визуализируете шаблон, вы передаете шаблонизатору объект, который называется *контекстным объектом*, что и обеспечивает работу подстановок.

Например, если мой контекстный объект `{ name: 'Лютик' }`, а шаблон — `<p>Привет, {{name}}!</p>`, то `{{name}}` будет заменено на `Лютик`. Что же произойдет, если

вы хотите передать HTML шаблону? Например, если вместо этого наш контекст будет `{ name: '<b>Лютик</b>' }`, использование предыдущего шаблона приведет к выдаче `<p>Привет, &lt;b&gt;Лютик&lt;b&gt;</p>`, что, вероятно, совсем не то, чего вы хотели. Для решения этой проблемы просто используйте три фигурные скобки вместо двух: `{{{name}}}`.

Несмотря на то что мы решили избегать использования формирования HTML с помощью JavaScript, возможность отключать экранирование HTML с помощью тройных фигурных скобок имеет интересные способы применения. Например, если вы создаете CMS с помощью WYSIWYG-редактора, вам, вероятно, захочется иметь возможность передавать HTML вашим представлениям. Кроме того, возможность визуализировать свойства из контекста без экранирования HTML важна для макетов и секций, о чем вы скоро узнаете.

На рис. 7.1 мы видим, как механизм Handlebars использует контекст (представленный овалом) в сочетании с шаблоном для визуализации HTML.

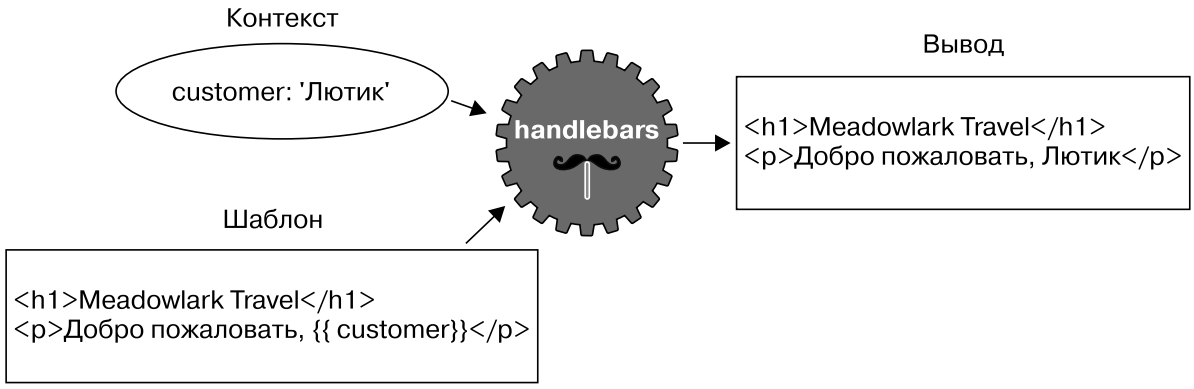


**Рис. 7.1.** Визуализация HTML с помощью Handlebars

# Комментарии

Комментарии в Handlebars выглядят вот так: `{{! Здесь находится комментарий }}`. Важно понимать различие между комментариями Handlebars и комментариями HTML. Рассмотрим следующий шаблон:

```
{{! Очень секретный комментарий }}
<!-- не очень секретный комментарий -->
```

Если это серверный шаблон, очень секретный комментарий никогда не будет отправлен браузеру, в то время как не очень секретный комментарий будет виден, если пользователь заглянет в исходный код HTML. Вам следует предпочесть комментарии Handlebars всем остальным, раскрывающим подробности реализации или что-либо еще, что вам не хотелось бы выставлять напоказ.

## Блоки

Все усложняется, когда мы начинаем рассматривать *блоки*. Блоки обеспечивают управление обменом данными, условное выполнение и расширяемость. Рассмотрим следующий контекстный объект:

```
{
    currency: {
        name: 'Доллары США',
        abbrev: 'USD',
    },
    tours: [
        { name: 'Река Худ', price: '$99.95' },
        { name: 'Орегон Коуст', price: '$159.95' },
    ],
    specialsUrl: '/january-specials',
    currencies: [ 'USD', 'GBP', 'BTC' ],
}
```

Теперь взглянем на шаблон, которому мы можем этот контекст передать:

```
<ul>
    {{#each tours}}
        {{! Я в новом блоке... и контекст изменился }}
        <li>
            {{name}} - {{price}}
            {{#if ../currencies}}
                ({{../../currency.abbrev}})
            {{/if}}
        </li>
    {{/each}}
</ul>
{{#unless currencies}}
    <p>Все цены в {{currency.name}}.</p>
{{/unless}}
{{#if specialsUrl}}
    {{! Я в новом блоке... но контекст вроде бы не изменился }}
    <p>Проверьте наши <a href="{{specialsUrl}}">специальные предложения!</p>
{{else}}
    <p>Просьба чаще пересматривать наши специальные предложения.</p>
{{/if}}
<p>
    {{#each currencies}}
        <a href="#" class="currency">{{.}}</a>
    {{else}}
```

```
        к сожалению, в настоящее время мы принимаем только {{currency.name}}.
    {{/each}}
</p>
```

В этом шаблоне происходит много чего, так что разобьем его на составные части. Он начинается с вспомогательного элемента `each`, обеспечивающего итерацию по массиву. Важно понимать, что между `{{#each tours}}` и `{{/each tours}}` меняется контекст. На первом проходе он меняется на `{ name: 'Река Худ', price: '$99.95' }`, а на втором проходе — на `{ name: 'Орегон Коуст', price, '$159.95' }`. Таким образом, внутри этого блока мы можем ссылаться на `{{name}}` и `{{price}}`. Однако если мы хотим обратиться к объекту `currency`, нам придется использовать `../`, чтобы получить доступ к *родительскому* контексту.

Если свойство контекста само по себе является объектом, мы можем обратиться к его свойствам как обычно, через точку например: `{{currency.name}}`.

Вспомогательный элемент `if` — особенный и слегка сбивающий с толку. В Handlebars **любой** блок будет менять контекст, так что внутри блока `if` — новый контекст, который оказывается копией родительского контекста. Другими словами, внутри блоков `if` или `else` — тот же контекст, родительский. Обычно это совершенно прозрачная деталь реализации, но ее необходимо иметь в виду при использовании блоков `if` внутри цикла `each`. В цикле `{{#each tours}}` мы можем обратиться к родительскому контексту с помощью `../`. Однако в блоке `{{#if ../currencies}}` мы вошли в новый контекст, так что для доступа к объекту `currencies` нам приходится использовать `../../`. Первый `../` доходит до уровня контекста `product`, а второй возвращается к самому внешнему контексту. Это вызывает страшную неразбериху, и простейший способ избежать этого — не использовать блоки `if` внутри блоков `each`.

Как у `if`, так и у `each` может быть (необязательный) блок `else` (в случае `each` блок `else` будет выполняться при отсутствии элементов в массиве). Мы также использовали вспомогательный элемент `unless`, по существу являющийся противоположностью вспомогательного элемента `if`: он выполняется только в том случае, когда аргумент ложен.

Последняя вещь, которую хотелось бы отметить относительно этого шаблона: использование `{{.}}` в блоке `{{#each currencies}}`. `{{.}}` просто ссылается на текущий контекст; в данном случае текущий контекст — просто строка в массиве, которую мы хотим вывести на экран.

Обращение к текущему контексту через одиночную точку имеет и другое применение: оно дает возможность различать вспомогательные элементы (которые мы вскоре изучим) и свойства текущего контекста. Например, если у вас есть вспомогательный элемент foo и свойство foo в текущем контексте, {{foo}} ссылается на вспомогательный элемент, а {{./foo}} — на свойство.

## Серверные шаблоны

Серверные шаблоны дают возможность визуализировать HTML **до** его отправки клиенту. В отличие от шаблонизации на стороне клиента, где шаблоны доступны любопытному пользователю, знающему, как смотреть исходный код HTML, ваши пользователи никогда не увидят серверные шаблоны или контекстные объекты, используемые для генерации окончательного HTML.

Помимо скрытия подробностей реализации, серверные шаблоны поддерживают *кэширование* шаблонов, играющее важную роль в обеспечении производительности. Шаблонизатор кэширует скомпилированные шаблоны (перекомпилируя и кэшируя заново только при изменении самого шаблона), что повышает производительность шаблонизированных представлений. По умолчанию кэширование представлений отключено в режиме разработки и включено в эксплуатационном режиме. Явным образом активировать кэширование представлений, если захотите, вы можете следующим образом: `app.set('view cache', true);`.

Непосредственно «из коробки» Express поддерживает Jade, EJS и JSHTML. Мы уже обсуждали Jade, и я не стану рекомендовать использовать EJS или JSHTML (на мой вкус, оба они недостаточно развиты в смысле синтаксиса). Итак, нужно добавить пакет Node, который обеспечит поддержку Handlebars для Express:

```
npm install --save express-handlebars
```

Затем привяжем его к Express:

```
var handlebars = require('express-handlebars')
    .create({ defaultLayout: 'main' });
app.engine('handlebars', handlebars.engine);
app.set('view engine', 'handlebars');
```

Пакет express-handlebars предполагает, что расширение шаблонов Handlebars будет .handlebars. Я уже привык к нему, но если это расширение слишком длинно для вас, можете изменить его на распространенное .hbs при создании экземпляра express-handlebars: require('express-handlebars').create({ extname: '.hbs' }).

## Представления и макеты

*Представление* обычно означает отдельную страницу вашего сайта (хотя оно может означать и загружаемую с помощью AJAX часть страницы, или электронное письмо, или что-то еще в том же роде). По умолчанию Express ищет представления в подкаталоге `views`. *Макет* — особая разновидность представления, по существу, шаблон для шаблонов. Макеты важны, поскольку у большинства (если не у всех) страниц вашего сайта будут практически одинаковые макеты. Например, у них должны быть элементы `<html>` и `<title>`, они обычно загружают одни и те же файлы

CSS и т. д. Вряд ли вы захотите дублировать этот код на каждой странице, вот тут-то и пригодятся макеты. Взглянем на остов макета:

```
<!doctype>
<html>
<head>
    <title>Meadowlark Travel</title>
    <link rel="stylesheet" href="/css/main.css">
</head>
<body>
    {{{body}}}
</body>
</html>
```

Обратите внимание на текст внутри тега `<body>`: `{{{body}}}`. Благодаря ему шаблонизатор знает, где визуализировать содержимое вашего представления. Важно использовать три фигурные скобки вместо двух, поскольку представление почти наверняка будет содержать HTML и мы не хотим, чтобы Handlebars пытался его экранировать. Замечу, что нет ограничений относительно размещения поля `{{{body}}}`. Например, если вы создаете быстро реагирующий макет в Bootstrap 3, то, вероятно, поместите представление внутри контейнера `<div>`. Многие обычные элементы страницы, такие как заголовки и нижние колонтитулы, также чаще всего находятся в макетах, не в представлениях. Вот пример:

```
<!-- ... -->
<body>
    <div class="container">
        <header><h1>Meadowlark Travel</h1></header>
        {{{body}}}
        <footer>&copy; {{copyrightYear}} Meadowlark Travel</footer>
    </div>
</body>
```

На рис. 7.2 мы видим, как шаблонизатор объединяет представление, макет и контекст. Самая важная вещь, которую проясняет эта диаграмма, — порядок выполнения действий. **Сначала**, до макета, *визуализируется представление*. На первый взгляд это может показаться нелогичным: раз представление визуализируется **внутри** макета, не должен ли макет визуализироваться первым? Хотя технически вполне возможно выполнить это, есть определенные преимущества в обратном порядке действий. В частности, он позволяет самому представлению осуществлять подгонку макета, что вполне может пригодиться, как мы увидим при обсуждении *секций*.

Благодаря определенному порядку выполнения действий вы можете передать в представление свойство body, и оно будет правильно визуализироваться в представлении. Однако при визуализации макета значение body будет перезаписано визуализируемым представлением.

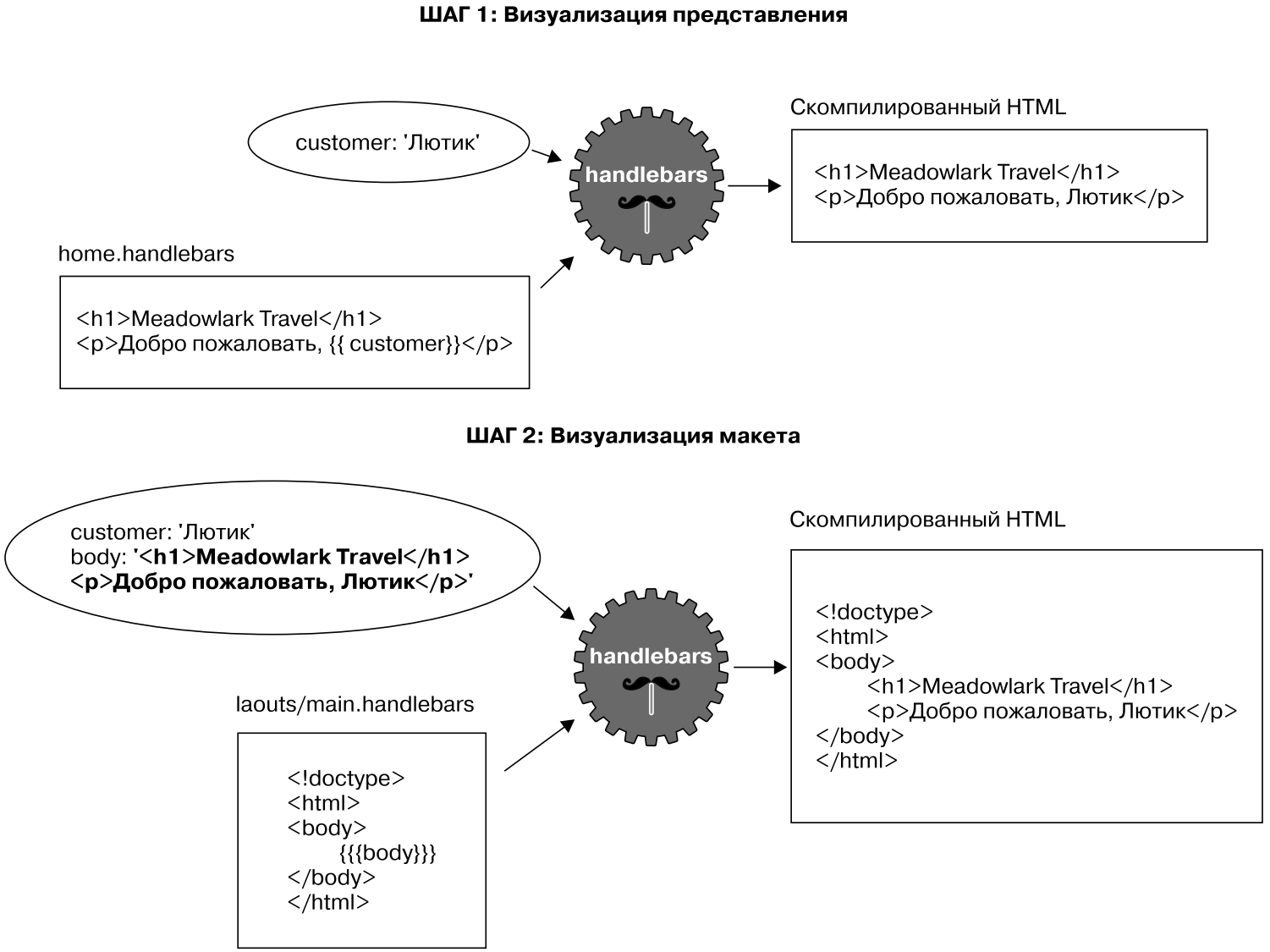


**Рис. 7.2.** Визуализация представления с помощью макета

## Использование (или неиспользование) макетов в Express

По всей вероятности, большинство, если не все ваши страницы станут использовать один и тот же макет, так что нет смысла указывать макет каждый раз при визуализации страницы. Как вы видели, при создании шаблонизатора мы задали имя макета по умолчанию:

```
var handlebars = require('express-handlebars')
    .create({ defaultLayout: 'main' });
```

По умолчанию Express ищет представления в подкаталоге `views`, а макеты — в подкаталоге `layouts`. Так что, если у вас есть представление `views/foo.handlebars`, можете его визуализировать следующим образом:

```
app.get('/foo', function(req, res){
    res.render('foo');
});
```

В качестве макета при этом будет использоваться `views/layouts/main.handlebars`. Если вообще не хотите применять макет (а значит, вам придется держать весь шаблонный код в представлении), можете указать в контекстном объекте `layout: null`:

```
app.get('/foo', function(req, res){
    res.render('foo', { layout: null });
});
```

А если хотите использовать другой шаблон, можете указать имя шаблона:

```
app.get('/foo', function(req, res){
    res.render('foo', { layout: 'microsite' });
});
```

При этом будет визуализироваться представление с макетом `views/layouts/microsite.handlebars`.

Помните, что чем больше у вас шаблонов, тем проще следует быть вашему макету HTML. В то же время это может оправдаться при наличии у вас страниц, сформированных по иному макету. Вам придется найти в этом вопросе правильное соотношение для своих проектов.

## Частичные шаблоны

Очень часто вам будут попадаться компоненты, которые вы захотите повторно использовать на различных страницах (в кругах разработчиков клиентской части их часто называют виджетами). Один из способов добиться этого с помощью шаблонов — использовать *частичные шаблоны* (называемые так потому, что они не визуализируют целое представление или целую страницу). Допустим, нам нужен

компонент Current Weather, отображающий текущие погодные условия в Портленде, Бенде и Манзаните[1]. Мы хотим сделать этот компонент пригодным для повторного применения, чтобы иметь возможность поместить его на любую нужную нам страницу. Поэтому будем использовать частичный шаблон. Вначале создадим файл частичного шаблона `views/partials/weather.handlebars`:

```
<div class="weatherWidget">
    {{#each partials.weatherContext.locations}}
        <div class="location">
            <h3>{{name}}</h3>
            <a href="{{forecastUrl}}">
                <img src="{{iconUrl}}" alt="{{weather}}">
                {{weather}}, {{temp}}
            </a>
        </div>
    {{/each}}
    <small>Источник: <a href="http://www.wunderground.com">Weather
        Underground</a></small>
</div>
```

Обратите внимание на то, что имя области видимости контекста начинается с `partials.weatherContext`: поскольку нам требуется возможность применять частичный шаблон на любой странице, передавать контекст для каждого представления неудобно, так что вместо этого мы используем объект `res.locals`, доступный для каждого представления. Но поскольку нам не хотелось бы пересекаться с контекстом, задаваемым отдельными представлениями, мы поместили весь частичный контекст в объект `partials`.

express-handlebars позволяет вам передавать частичные шаблоны как часть контекста. Например, если вы добавите partials.foo = "Шаблон!" в ваш контекст, вы сможете визуализировать этот частичный шаблон с помощью {{> foo}}. При таком использовании любые файлы представлений .handlebars будут переопределены, и именно поэтому мы выше добавили partials.weatherContext вместо partials.weather, которое переопределило бы views/partials/weather.handlebars.

В главе 19 мы увидим, как можно извлечь информацию о текущей погоде из общедоступного API Weather Underground. А пока просто будем использовать фиктивные данные. Создадим в файле приложения функцию для получения данных о погоде:

```
function getWeatherData(){
    return {
        locations: [
            {
                name: 'Портленд',
```

[1] Города в штате Орегон. — *Примеч. пер.*

```
            forecastUrl: 'http://www.wunderground.com/US/OR/Portland.html',
            iconUrl: 'http://icons-ak.wxug.com/i/c/k/cloudy.gif',
            weather: 'Сплошная облачность ',
            temp: '54.1 F (12.3 C)',
        },
        {
            name: 'Бенд',
            forecastUrl: 'http://www.wunderground.com/US/OR/Bend.html',
            iconUrl: 'http://icons-ak.wxug.com/i/c/k/partlycloudy.gif',
            weather: 'Малооблачно',
            temp: '55.0 F (12.8 C)',
        },
        {
            name: 'Манзанита',
            forecastUrl: 'http://www.wunderground.com/US/OR/Manzanita.html',
            iconUrl: 'http://icons-ak.wxug.com/i/c/k/rain.gif',
            weather: 'Небольшой дождь',
            temp: '55.0 F (12.8 C)',
        },
    ],
};
}
```

Теперь создадим промежуточное ПО для внедрения этих данных в объект `res.locals.partials` (подробнее поговорим о промежуточном ПО в главе 10):

```
app.use(function(req, res, next){
    if(!res.locals.partials) res.locals.partials = {};
    res.locals.partials.weatherContext = getWeatherData();
    next();
});
```

Теперь, когда все настроено, все, что нам осталось сделать, — использовать частичный шаблон в представлении. Например, чтобы поместить наш виджет на домашнюю страницу, отредактируем `views/home.handlebars`:

```
<h2>Добро пожаловать в Meadowlark Travel!</h2>
{{> weather}}
```

Синтаксис вида `{{> partial_name}}` описывает то, как вы включаете частичный шаблон в представление: `express-handlebars` будет знать, что нужно искать представление `partial_name.handlebars` (или `weather.handlebars` в нашем примере) в `views/partials`.

express-handlebars поддерживает подкаталоги, так что, если у вас много частичных шаблонов, можете их упорядочить. Например, если у вас есть частичные шаблоны социальных медиа, можете разместить их в каталоге views/partials/social и включить с помощью {{> social/facebook}}, {{> social/twitter}} и т. д.

## Секции

В основе одного метода, который я позаимствовал у созданного компанией Microsoft замечательного шаблонизатора Razor, лежит идея *секций*. Макеты отлично работают, если каждое из представлений спокойно помещается в отдельный элемент макета, но что произойдет, если представлению понадобится внедриться в другие части макета? Распространенный пример — представление, которому требуется добавить что-либо в элемент `<head>` или вставить использующий jQuery `<script>` (а значит, он должен оказаться после ссылки на jQuery, которая из соображений производительности иногда оказывается последним элементом макета).

Ни у Handlebars, ни у `express-handlebars` нет встроенного способа сделать это. К счастью, вспомогательные элементы Handlebars сильно облегчают эту задачу. При создании объекта Handlebars мы добавим вспомогательный элемент `section`:

```
var handlebars = require('express-handlebars').create({
    defaultLayout:'main',
    helpers: {
        section: function(name, options){
            if(!this._sections) this._sections = {};
            this._sections[name] = options.fn(this);
            return null;
        }
    }
});
```

Теперь мы можем применять в представлении вспомогательный элемент `section`. Добавим представление (`views/jquery-test.handlebars`) для включения чего-либо в `<head>` и сценарий, использующий jQuery:

```
{{#section 'head'}}
    <!-- Мы хотим, чтобы Google игнорировал эту страницу -->
    <meta name="robots" content="noindex">
{{/section}}
<h1>Тестовая страница</h1>
<p>Тестируем что-то, связанное с jQuery.</p>
{{#section 'jquery'}}
    <script>
        $('document').ready(function(){
            $('h1').html('jQuery работает');
        });
    </script>
{{/section}}
```

А теперь может разместить в нашем макете секции так же, как размещаем `{{{body}}}`:

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

```
<head>
    <title>Meadowlark Travel</title>
    {{{_sections.head}}}
</head>
<body>
    {{{body}}}
    <script src="http://code.jquery.com/jquery-2.0.2.min.js"></script>
    {{{_sections.jquery}}}
</body>
</html>
```

## Совершенствование шаблонов

В основе вашего сайта лежат шаблоны. Хорошая структура шаблонов сбережет время разработки, увеличит единообразие сайта и уменьшит число мест, в которых могут скрываться различные странности макетов. Но, чтобы получить этот эффект, вам придется провести некоторое время за тщательным проектированием шаблонов. Определить, сколько требуется шаблонов, — настоящее искусство: в общем случае чем меньше, тем лучше, однако остается проблема снижения числа возвратов, зависящая от однородности страниц. Шаблоны являются также первой линией защиты от проблем межбраузерной совместимости и правильности HTML. Они должны быть спроектированы с любовью и сопровождаться кем-то, кто хорошо разбирается в разработке клиентской части. Отличное место для начала этой деятельности, особенно если вы новичок, — HTML5 Boilerplate. В предыдущих примерах мы использовали минимальный шаблон HTML5, чтобы соответствовать книжному формату, но для нашего настоящего проекта будем применять HTML5 Boilerplate.

Другое популярное место, с которого стоит начать работу с шаблонами, — темы производства сторонних разработчиков. На таких сайтах, как Themeforest (http://themeforest.net/category/site-templates) и WrapBootstrap (https://wrapbootstrap.com/), можно найти сотни готовых для использования тем, которые вы можете применять в качестве отправной точки для своих шаблонов. Использование тем сторонних разработчиков начинается с переименования основного файла (обычно `index.html`) в `main.handlebars` (или дайте файлу макета другое название) и помещения всех ресурсов (CSS, JavaScript, изображений) в используемый вами для статических файлов каталог `public`. Затем нужно будет отредактировать файл шаблона и решить, куда вы хотите вставить выражение `{{{body}}}`. В зависимости от элементов шаблона вы можете захотеть переместить некоторые из них в частичные шаблоны. Отличный пример этого — «герой» (большой баннер, разработанный специально для привлечения внимания пользователя). Если «герой» должен отображаться на каждой странице (вероятно, не лучший вариант), вы, наверное, оставите его в файле шаблона. Если он показывается только на одной странице (обычно домашней),

то его лучше поместить только в это представление. Если он показывается на нескольких (но не на всех) страницах, то можно рассмотреть возможность его помещения в частичный шаблон. Выбор за вами, и в этом состоит искусство создания оригинального, привлекательного сайта.

## Handlebars на стороне клиента

Шаблонизация с помощью Handlebars на стороне клиента удобна, если вы хотите, чтобы у вас было динамическое содержимое. Конечно, вызовы AJAX могут возвращать фрагменты HTML, которые мы можем просто вставить в DOM как есть, но Handlebars на стороне клиента дает возможность получать результаты вызовов AJAX в виде данных в формате JSON и форматировать их для соответствия нашему сайту. Поэтому он особенно удобен для связи со сторонними API, которые будут возвращать не HTML, а JSON, отформатированный под ваш сайт.

Перед тем как начать применять Handlebars на стороне клиента, нам нужно загрузить Handlebars. Мы можем сделать это или вставив Handlebars в статическое содержимое, или с помощью уже доступного CDN. Будем использовать в `views/nursery-rhyme.handlebars` второй подход:

```
{{#section 'head'}}
    <script src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.3.0/↵
handlebars.min.js"></script>
{{/section}}
```

Теперь нам нужно место для размещения шаблонов. Для этого можно использовать уже существующий элемент нашего HTML, желательно скрытый. Вы можете выполнить это, поместив HTML в элементы `<script>` в `<head>`. На первый взгляд это кажется странным, но работает отлично:

```
{{#section 'head'}}
    <script src="//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.3.0/
       handlebars.min.js"></script>
    <script id="nurseryRhymeTemplate" type="text/x-handlebars-template">
        у Мэри был маленький <b>\{{animal}}</b>, его <b>\{{bodyPart}}</b>
        был <b>\{{adjective}}</b>, как <b>\{{noun}}</b>.
    </script>
{{/section}}
```

Обратите внимание на то, что нам пришлось экранировать хотя бы одну из фигурных скобок, в противном случае при обработке представлений на стороне сервера произошла бы попытка выполнить вместо них подстановки.

Перед использованием шаблона нужно его скомпилировать:

```
{{#section 'jquery'}}
    $(document).ready(function(){
```

```
        var nurseryRhymeTemplate = Handlebars.compile(
            $('#nurseryRhymeTemplate').html());
    });
{{/section}}
```

И нам нужно место для размещения визуализированного шаблона. Для тестовых целей добавим пару кнопок: одну для визуализации непосредственно из JavaScript, другую — для визуализации из вызова AJAX:

```
<div id="nurseryRhyme">Нажмите кнопку...</div>
<hr>
<button id="btnNurseryRhyme">Генерация детского стишка </button>
<button id="btnNurseryRhymeAjax"> Генерация детского стишка из AJAX</button>
```

И наконец, код визуализации шаблона:

```
{{#section 'jquery'}}
    <script>
        $(document).ready(function(){

            var nurseryRhymeTemplate = Handlebars.compile(
                $('#nurseryRhymeTemplate').html());

                var $nurseryRhyme = $('#nurseryRhyme');

                $('#btnNurseryRhyme').on('click', function(evt){
                    evt.preventDefault();
                    $nurseryRhyme.html(nurseryRhymeTemplate({
                        animal: 'василиск',
                        bodyPart: 'хвост',
                        adjective: 'острый',
                        noun: 'иголка'
                    }));
            });
            $('#btnNurseryRhymeAjax').on('click', function(evt){
                evt.preventDefault();
                $.ajax('/data/nursery-rhyme', {
                    success: function(data){
                        $nurseryRhyme.html(
                            nurseryRhymeTemplate(data))
                    }
                });
            });

        });
    </script>
{{/section}}
```

И обработчики маршрутов для страницы детского стишка и вызова AJAX:

```
app.get('/nursery-rhyme', function(req, res){
    res.render('nursery-rhyme');
});
app.get('/data/nursery-rhyme', function(req, res){
    res.json({
        animal: 'бельчонок',
        bodyPart: 'хвост',
        adjective: 'пушистый',
        noun: 'черт',
    });
});
```

По существу, `Handlebars.compile` принимает шаблон и возвращает функцию. Эта функция принимает контекстный объект и возвращает визуализированную строку. Так что сразу после компиляции шаблонов мы получим пригодные для повторного использования визуализаторы, к которым можно обращаться как к функциям.

# Резюме

Мы увидели, как шаблонизация может облегчить написание, чтение и сопровождение вашего кода. Благодаря шаблонам больше не нужно мучительно собирать HTML из строк JavaScript: мы можем писать HTML в любимом редакторе и делать его динамическим с помощью лаконичного и легкого для чтения языка шаблонизации.