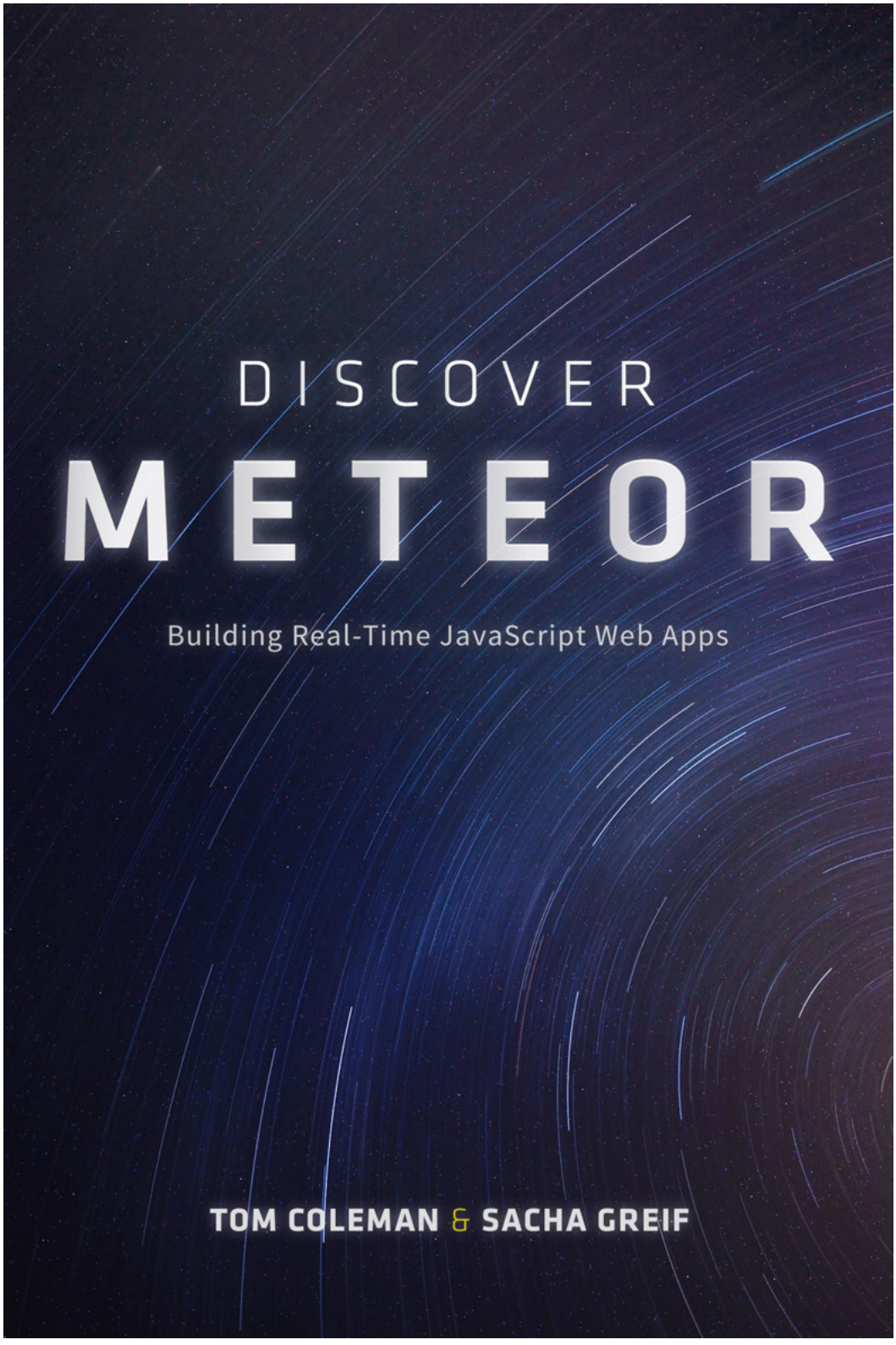

DISCOVER
METEOR
Building Real-Time JavaScript Web Apps
TOM COLEMAN & SACHA GREIF

www.discovermeteor.com

Давайте проведем небольшой мысленный эксперимент. Представьте, что вы открыли одну и ту же папку, в двух разных окнах на вашем компьютере.

Теперь удалите файл в одном из этих окон. Файл исчез и из другого окна, ведь так?

Вам вовсе не нужно проделывать эти шаги в реальности, чтобы понять, что он также исчез. Когда мы меняем что-либо в нашей локальной файловой системе, изменения применяются везде без необходимости перезагружать окно. Это просто происходит.

Теперь давайте подумаем о том, как тот же сценарий будет развиваться в Интернете. Представим, что вы открыли панель администратора WordPress сайта в двух окнах браузера, а затем добавили пост в одном из них. В отличие от похожей ситуации на вашем компьютере, другое окно не будет отображать изменения, пока вы его не обновите, независимо от того, сколько времени вы прождете.

На протяжении многих лет мы мирились с тем, что общаться с веб-сайтом можно лишь средствами отдельных, коротких запросов.

Но Meteor является представителем новой волны веб-фреймворков, которые бросают вызов устоявшемуся несовершенному порядку, внедряя современные концепции real-time web* и reactive programming*.

Meteor - это платформа для создания так называемых real-time web apps - современных веб-приложений. По сути, Meteor - это слой между интерфейсом вашего приложения и его базой данных, который следит за их синхронизацией.

Поскольку фреймворк построен на основе Node.js, то JavaScript используется как на клиенте, так и на сервере. И более того, Meteor позволяет использовать один и тот же код и на клиенте, и на сервере!

В результате всего этого мы получаем очень мощную, и при этом простую в использовании платформу, так как большинство стандартных рутин и трудностей создания веб-приложений уже реализованы из коробки.

Итак, зачем же вам тратить свое время на изучение Meteor, а не выбрать какой-нибудь другой фреймворк? Даже если мы сейчас оставим в стороне все остальные преимущества этой платформы, то главным останется то, что Meteor удивительно прост в освоении!

В отличие от других фреймворков, он позволяет вам создать собственное real-time веб-приложение и выложить его в Интернете в течение всего лишь нескольких часов. И плюс, если вы когда-нибудь занимались front-end разработкой, то вы уже знакомы с JavaScript и не нужно изучать

новый язык.

Meteor может идеально подходить для ваших нужд, а может и не совсем. Но так как на то, чтобы изучить этот фреймворк вам потребуется всего несколько вечеров или уик-энд, то почему бы не попробовать и не выяснить самому?

Последние пару лет мы работали над множеством Meteor-приложений, от веб-приложений до мобильных, коммерческих проектов и проектов с открытым кодом.

Мы усвоили тонны полезной информации, разрабатывая эти приложения, и часто было очень непросто находить ответы на возникающие по ходу вопросы. Нам приходилось искать эти ответы по кускам в разных источниках, а во многих случаях даже придумывать свои решения. Цель написания данной книги - создание простой пошаговой инструкции на основе всего, что мы усвоили, которая постепенно проведет вас через все этапы создания полноценного Meteor-приложения с нуля!

Приложение, которое мы с вами будем создавать - это слегка упрощенная версия сайтов социальных новостей как **Hacker News** или **Reddit**, и назовем его Microscope (по аналогии с его старшим братом, Meteor-приложения с открытым исходным кодом **Telescope**). Шаг за шагом, в процессе его создания, мы затронем все основные темы, касающиеся разработки приложений на основе Meteor, таких как работа с учетными записями пользователя, коллекции данных, маршрутизация и многое другое.

Одна из задач при написании этой книги состояла в том чтобы сделать ее доступной и легкой для понимания. Так, что даже не имея опыта работы с Meteor, Node.js, MVC фреймворками и серверной разработки, вы должны без проблем следовать материалу книги.

С другой стороны, мы ожидаем что вы знакомы с основными концепциями и синтаксисом JavaScript. Если вы хотя бы писали код на jQuery или играли с консолью разработчика браузера, то вы скорее всего справитесь.

Если вы сомневаетесь в своих познаниях JavaScript, то мы рекомендуем ознакомиться с **JavaScript минимум для Meteor** перед прочтением этой книги.

На случай, если вам интересно, кто же мы такие и почему вы должны нам доверять, то вот немного информации о нас обоих:

Tom Coleman - сотрудник **Percolate Studio**, агентства веб-разработки с особым вниманием к качеству и удобству своих продуктов. Он является одним из разработчиков **Atmosphere**, а также участник многих других Meteor-проектов с открытым исходным кодом (таких как **Iron Router**).

Sasha Greif сотрудничал с такими стартапами как **Hipmunk** и **RubyMotion** в качестве веб-дизайнера. Он создатель проектов **Telescope** и **Sidebar** (основанном на Telescope), а также основатель **Folyo**.

Мы старались, чтобы эта книга была полезной как тех, кто только познакомился с Meteor, так и для опытных программистов, поэтому мы разделили ее главы на 2 категории: обычные (с номерами от 1 до 14) и дополнительные (номера оканчиваются на .5).

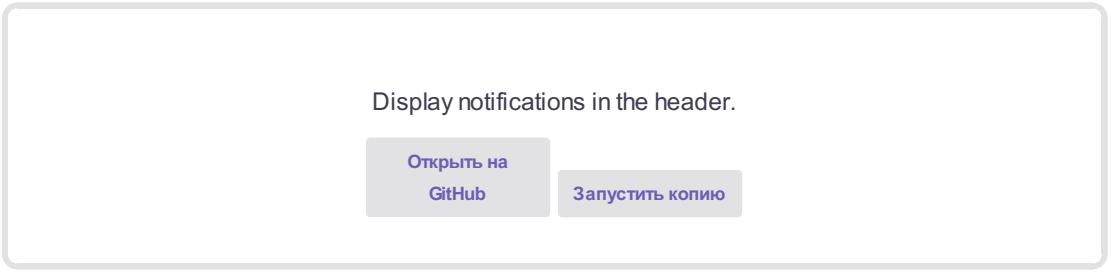
В обычных главах мы будем объяснять вам самые важные, базовые вещи процесса создания приложения, не углубляясь слишком глубоко в детали. Так вы быстрее войдете в курс дела и сможете начать применять свои навыки.

В дополнительных же главах мы наоборот будем уводить вас глубже в тонкости фреймворка, и поможем лучше понять, что же происходит за кулисами процессов.

Так что, если вы только начинаете знакомство с Meteor, то лучше просто пропускайте дополнительные главы при первом прочтении. Вы всегда сможете к ним вернуться после изучения основ.

Нет ничего хуже чем изучать книгу по программированию, и вдруг понять, что код, который вы пишите отличается от примеров и не работает так, как нужно.

Для этого мы создали **репозиторий на GitHub**. Также после изменений кода в книге мы добавляем прямую ссылку на коммиты в этом репозитории. Плюс ко всему, с каждым коммитом сопряжена своя рабочая копия приложения на текущем этапе разработки, так что вы всегда можете ее сравнить со своей локальной версией. Вот пример того, как это выглядит:



Но это не значит, что вы должны банально прыгать от одного `git checkout` к другому. Процесс обучения пройдет в несколько раз эффективнее, если вы будете вручную набирать код приложения и по-настоящему вникать в него.

Если вы захотите глубже изучить какой-нибудь отдельный аспект Meteor, то лучшим местом, чтобы начать будет **официальная документация**.

Также мы рекомендуем **официальный Stack Overflow** и #meteor **IRC канал**, если вам нужна помощь в режиме реального времени.

Хотя знакомство с системой контроля версий Git некритично для изучения этой книги, мы все же настоятельно рекомендуем ознакомиться с данным инструментом.

Чтобы быстро разобраться с Git, можете приобрести книгу Nick Farina **Git Is Simpler Than You Think**.

Также, если вы новичок Git и пользователь системы Mac, то мы рекомендуем приложение **GitHub for Mac**, которое позволяет вам управлять репозиториями без использования командной строки.

- Если вы хотите с нами связаться, то пишите нам на **hello@discovermeteor.com**.
- Если вы найдете ошибку в переводе, то дайте нам знать, **сообщив о баге**.
- Если вы обнаружили проблему в коде самого Microscope, то можете сообщить о баге **здесь**.
- Наконец, задать любой другой вопрос вы можете прямо в боковой панели этого приложения.

Первое впечатление крайне важно, и установка Meteor должна пройти без особых сложностей. В большинстве случаев, весь процесс занимает не более пяти минут.

Для начала мы можем установить Meteor набрав в терминале:

```
curl https://install.meteor.com | sh
```

Это установит глобальную команду meteor, после чего вы сможете начать использовать фреймворк.

> *не*
>
> Если вы не можете (или не хотите) устанавливать Meteor локально, рекомендуем вам **Nitrous.io**.
>
> Nitrous.io - это сервис, который позволяет запускать приложения и редактировать их код прямо в браузере. Мы написали **небольшое руководство по этой теме**
>
> Вы можете просто пройти эту инструкцию до пункта «Устанавливаем Meteor» включительно, а затем снова вернуться к этой книге начиная с секции «Создаем простое приложение» текущей главы.

Теперь, когда мы установили Meteor, давайте создадим приложение. Чтобы это сделать, мы используем команду `meteor`:

```
meteor create microscope
```

Эта команда загрузит Meteor, развернет и создаст базовый, готовый к работе проект. Когда Meteor закончит этот процесс, вы увидите, что директория `microscope/` содержит следующее:

```
.meteor
microscope.css
microscope.html
microscope.js
```

Приложение, которое Meteor создал за вас - это шаблонное приложение, демонстрирующее несколько простых подходов.

Хотя наше приложение пока мало что делает, мы все же можем запустить его. Чтобы это сделать, вернитесь в окно терминала и введите:

```
cd microscope
meteor
```

Теперь перейдите в браузере по адресу `http://localhost:3000/` (или `http://0.0.0.0:3000/`) и вы должны увидеть что-то вроде этого:

Meteor's Hello World.

Created basic microscope project.

Открыть на GitHub | Запустить копию

Поздравляем! Вы только что создали и запустили свое первое Meteor-приложение. Кстати, чтобы прервать его работу откройте вкладку терминала, в котором оно исполняется и наберите `ctrl+c`.

Заметьте, что если вы используете Git, сейчас самое время для инициализации вашего репозитория командой `git init`.

> В одно время Meteor использовал внешний менеджер пакетов, который назывался Meteorite. Начиная с версии Meteor 0.9.0, Meteorite больше не нуждается, так как его функции были ассимилированы в самом Meteor.
>
> Так что если вы обнаружите какие-либо отсылки к командам Meteorite `mrt` в этой книге, или на просторах Интернета и в материалах про Meteor, вы можете смело заменять их на привычное `meteor`.

Давайте воспользуемся системой пакетов Meteor для включения **Bootstrap** в наш проект.

Это ничем не отличается от добавления Bootstrap обычным способом, когда мы вручную добавляем CSS и JavaScript файлы в проект. За исключением того, что мы воспользуемся пакетом члена сообщества Meteor **Andrew Mao** (где “mizzao” в `mizzao:bootstrap-3` имя пользователя автора этого пакета) чтобы иметь актуальную версию фреймворка.

Мы также добавим пакет **Underscore**. Underscore это очень полезная вспомогательная библиотека JavaScript, когда дело доходит до управления структурами данных в JavaScript.

На момент написания статьи пакет `underscore` все еще является частью «официальных» пакетов Meteor, вот почему она не имеет автора:

```
meteor add mizzao:bootstrap-3
meteor add underscore
```

Обратите внимание, что мы добавляем Bootstrap **3**. Некоторые скринкасты этой книги были сделаны со старой версией Microscope, которая работала под Bootstrap **2** и поэтому они могут выглядеть несколько иначе.

Added bootstrap and underscore packages.

Открыть на GitHub | Запустить копию

Как только вы добавите пакет Bootstrap, вы увидите изменения в нашем минималистичном приложении:

В отличие от «традиционного» метода добавления внешних ресурсов нам не нужно связывать CSS или JavaScript файлы, потому что Meteor сделает это за нас! Это лишь одно из многих преимуществ пакетов Meteor.

Когда мы говорим о модулях в контексте Meteor, необходимо учитывать некоторую специфику. Фреймворк использует несколько базовых типов модулей (packages):

- Ядро самого Meteor состоит из **базовых модулей**. Они автоматически включены в каждое Meteor-приложение, и вы пока можете не думать о них вовсе.
- Стандартные модули известны как «**изомодули**» или изоморфные модули (то есть они могут работать как на клиенте, так и на сервере). **Официальные модули** такие, как `accounts-ui` или `appcache` поддерживаются рабочей группой Meteor и **поставляются вместе с Meteor**.
- **Сторонние модули** - это те же изомодули, разработанные другими пользователями и загруженные на сервер пакетов Meteor. Вы можете просматривать их на **Atmosphere** или используя команду `meteor search`.
- **Локальные модули**, модули которые вы создаете сами и хранятся в директории `/packages`.
- **NPM-модули** (Node.js Packaged Modules) - это модули Node.js. Из коробки они не работают с Meteor, но могут быть использованы предыдущими типами модулей.

Прежде чем приступить непосредственно к разработке, мы должны правильно организовать наш проект. Для того чтобы создать нужную файловую структуру, спервa откройте директорию ‘/microscope’ и удалите ‘microscope.html’, ‘microscope.js’ и ‘microscope.css’.

Далее, создадим четыре основных директорий внутри `/microscope`: `/client`, `/server`, `/public` и `/lib`.

Также создадим пустые файлы `main.html` и `main.js` внутри только что созданной директории `/client`. Не волнуйтесь, что сейчас это поломает приложение, мы вернемся к этим файлам в следующей главе.

Необходимо отметить, что некоторые из этих директорий имеют специальное назначение. Когда дело касается файловой структуры, у Meteor есть ряд правил:

- Код в директории `/server` исполняется только на сервере.
- Код в директории `/client` исполняется только на клиенте.
- Все остальные файлы исполняются и на клиенте, и на сервере.
- Статичные файлы (картинки, шрифты и т.п.) - в директории `/public`.

Также полезно знать, как Meteor решает в каком порядке загрузить ваши файлы:

- Файлы в директории `/lib` загружаются самыми *первыми*
- Все файлы с именем `main.*` загружаются самыми *последними*
- Всё остальное загружается в алфавитном порядке по имени файла.

Заметьте также, что несмотря на эти правила, вы не обязаны использовать предлагаемую структуру, если вы этого не хотите. Предложенная выше структура - всего лишь один из способов организации файлов, а не строгий закон.

Если хотите получить больше информации по этому вопросу, то рекомендуем почитать **официальную документацию**.

> Если вы имели опыт работы с такими фреймворками, как Ruby on Rails, вы можете задаться вопросом, придерживается ли Meteor архитектуры MVC (Model View Controller)?
>
> Короткий ответ - нет! В отличие от Rails, Meteor не заставляет придерживаться какой-либо определенной архитектуры, так что в этой книге мы просто структурируем приложение так, как считаем оптимальным, не особо заботясь о том, какой аббревиатурой это обозвать.

Ладно, мы соврали. Нам не нужна директория `public/`, но только лишь потому, что Microscope не использует статичных файлов. Но, так как большинство других Meteor-приложений включают в себя хотя бы пару картинок, мы посчитали нужным коснуться и этой директории.

Кстати, вы должно быть заметили спрятанную директорию `.meteor`. Там находится код самого Meteor, и что-то там менять - обычно очень плохая идея. На деле вам вряд ли когда-нибудь понадобится туда даже заглядывать. Единственное исключение - это файлы `.meteor/packages` и `.meteor/release`, которые отвечают соответственно за ваши модули и используемую версию Meteor. Когда вы добавляете модули или меняете версию Meteor, то иногда бывает полезно проверить эти файлы.

> Что мы скажем по поводу извечного холивара `my_variable` или `myVariable` - неважно, каким из этих стилей вы пользуетесь, если вы используете его везде.
>
> В этой книге мы используем CamelCase, так как это стандартно для JavaScript (в конце концов, он именно JavaScript, а не java_script!).
>
> Единственное исключение - это имена файлов (`my_file.js`), и CSS-классы (`.my-class`). Причина в том, что символ подчёркивания больше распространён в файловой системе, а CSS в самом своем синтаксисе использует дефисы (`font-family` , `text-align` и т.д.).

Эта книга не о CSS. Чтобы не тормозить вас заботой о стилях и не отвлекаться на него в дальнейшем, мы решили добавить весь CSS в самом начале.

Meteor автоматически минифицирует CSS, так что, в отличие от остальных статичных файлов, ему лучше находиться в директории `/client` , а не `/public` . Давайте создадим директорию `/client/stylesheets/` и положим туда вот этот файл `style.css` :

```
.grid-block, .main, .post, .comments li, .comment-form {
  background: #fff;
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  -ms-border-radius: 3px;
  -o-border-radius: 3px;
  border-radius: 3px;
  padding: 10px;
  margin-bottom: 10px;
  -webkit-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);
  -moz-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);
  box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15); }

body {
  background: #eee;
  color: #666666; }

.navbar {
  margin-bottom: 10px; }
  /* line 32, ../sass/style.scss */
  .navbar .navbar-inner {
    -webkit-border-radius: 0px 0px 3px 3px;
    -moz-border-radius: 0px 0px 3px 3px;
    -ms-border-radius: 0px 0px 3px 3px;
    -o-border-radius: 0px 0px 3px 3px;
    border-radius: 0px 0px 3px 3px; }

#spinner {
  height: 300px; }

.post {
  /* For modern browsers */
  /* For IE 6/7 (trigger hasLayout) */
  *zoom: 1;
  position: relative;
  opacity: 1; }
  .post:before, .post:after {
    content: "";
    display: table; }
  .post:after {
    clear: both; }
  .post.invisible {
    opacity: 0; }
  .post.instant {
    -webkit-transition: none;
    -moz-transition: none;
    -o-transition: none;
    transition: none; }
  .post.animate{
    -webkit-transition: all 300ms 0ms ease-in;
    -moz-transition: all 300ms 0ms ease-in;
    -o-transition: all 300ms 0ms ease-in;
    transition: all 300ms 0ms ease-in; }
  .post .upvote {
    display: block;
    margin: 7px 12px 0 0;
    float: left; }
  .post .post-content {
    float: left; }
    .post .post-content h3 {
      margin: 0;
      line-height: 1.4;
      font-size: 18px; }
      .post .post-content h3 a {
        display: inline-block;
        margin-right: 5px; }
      .post .post-content h3 span {
        font-weight: normal;
        font-size: 14px;
        display: inline-block;
```

```
        color: #aaaaaa; }
    .post .post-content p {
      margin: 0; }
  .post .discuss {
    display: block;
    float: right;
    margin-top: 7px; }

.comments {
  list-style-type: none;
  margin: 0; }
  .comments li h4 {
    font-size: 16px;
    margin: 0; }
    .comments li h4 .date {
      font-size: 12px;
      font-weight: normal; }
    .comments li h4 a {
      font-size: 12px; }
  .comments li p:last-child {
    margin-bottom: 0; }

.dropdown-menu span {
  display: block;
  padding: 3px 20px;
  clear: both;
  line-height: 20px;
  color: #bbb;
  white-space: nowrap; }

.load-more {
  display: block;
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  -ms-border-radius: 3px;
  -o-border-radius: 3px;
  border-radius: 3px;
  background: rgba(0, 0, 0, 0.05);
  text-align: center;
  height: 60px;
  line-height: 60px;
  margin-bottom: 10px; }
  .load-more:hover {
    text-decoration: none;
    background: rgba(0, 0, 0, 0.1); }

.posts .spinner-container{
  position: relative;
  height: 100px;
}

.jumbotron{
  text-align: center;
}
.jumbotron h2{
  font-size: 60px;
  font-weight: 100;
}

@-webkit-keyframes fadeOut {
  0% {opacity: 0;}
  10% {opacity: 1;}
  90% {opacity: 1;}
  100% {opacity: 0;}
}

@keyframes fadeOut {
  0% {opacity: 0;}
  10% {opacity: 1;}
  90% {opacity: 1;}
  100% {opacity: 0;}
```

```
}

.errors{
  position: fixed;
  z-index: 10000;
  padding: 10px;
  top: 0px;
  left: 0px;
  right: 0px;
  bottom: 0px;
  pointer-events: none;
}
.alert {
          animation: fadeOut 2700ms ease-in 0s 1 forwards;
  -webkit-animation: fadeOut 2700ms ease-in 0s 1 forwards;
     -moz-animation: fadeOut 2700ms ease-in 0s 1 forwards;
  width: 250px;
  float: right;
  clear: both;
  margin-bottom: 5px;
  pointer-events: auto;
}
```

client/stylesheets/style.css

Re-arranged file structure.

Открыть на GitHub | Запустить копию

В этой книге мы будем использовать чистый JavaScript, но если вы предпочитаете CoffeeScript, то просто добавьте пакет CoffeeScript и начинайте писать на нем:

```
meteor add coffeescript
```

Некоторые предпочитают спокойно работать над проектом, пока не доведут его до совершенства, другие же напротив, стараются показать его всему миру как можно раньше.

Если вы относитесь к первой категории и сейчас предпочтете работать локально, то просто пропустите эту главу. Если же вы хотите сразу научиться публиковать Meteor-приложения, то эта глава для вас.

Мы научимся публиковать приложения несколькими способами. Пробуйте их на любой стадии разработки, работаете вы над Microscope, либо над любым другим Meteor-приложением. Давайте приступим!

> Вы сейчас читаете именно дополнительную главу. В этих главах мы будем более детально разбирать общие темы Meteor, независимо от остальной книги.
>
> Так что если вы хотите просто продолжить разрабатывать Microscope, можете пока пропустить эту главу и вернуться к ней позже.

Опубликовать приложение на поддомен meteor.com (напр. ‘http://myapp.meteor.com’) - это самый простой способ. С него мы и начнем. Этот способ наиболее уместен, если вы хотите показать приложение другим людям на ранних стадиях, либо чтобы быстро развернуть тестовый сервер.

Публикация на meteor.com - очень простой процесс. Просто откройте терминал, войдите в директорию вашего приложения и наберите:

```
$ meteor deploy myapp.meteor.com
```

Само собой, замените ‘myapp’ на имя вашего приложения, предпочтительно свободное. Если такое имя уже используется, то Meteor может спросить у вас пароль. В этом случае просто отмените операцию набрав ‘ctrl+c’ и попробуйте снова, уже с другим именем.

Если все пройдет хорошо, то уже через несколько секунд вы сможете увидеть ваше приложение по адресу ‘http://myapp.meteor.com’.

По умолчанию нет никаких запретов по поддоменам ‘meteor.com’. Каждый может использовать любое имя, которое пожелает и перезаписать существующее приложение с таким же именем. Так что вы скорее всего захотите защитить свой домен паролем, добавив ‘-p’, как показано ниже:

```
$ meteor deploy myapp.meteor.com -p
```

Далее Meteor попросит вас ввести пароль, и после будет его спрашивать при каждой новой публикации на этот домен.

В **официальной документации** вы сможете найти больше информации по данному вопросу, например, как напрямую общаться с базой данных на вашем хосте или настроить собственное доменное имя для вашего приложения.

Modulus - отличное решение для публикации Node.js приложений. Это одна из немногих PaaS (platform-as-a-service) платформ, которая официально поддерживает Meteor, и уже есть люди, использующие ее в качестве production-решения для Meteor-приложений.

> Команда Modulus выложила в открытый доступ инструмент под названием **demeteorizer**, который конвертирует Meteor-приложение в стандартное Node.js приложение.

Начните с **создания аккаунта**. Чтобы опубликовать наше приложение на Modulus, сперва необходимо установить инструмент Modulus для командной строки:

```
$ npm install -g modulus
```

Далее необходимо залогиниться:

```
$ modulus login
```

Теперь давайте создадим наш проект Modulus (обратите внимание на то, что вы можете создать проект с помощью интерфейса на сайте Modulus)

```
$ modulus project create
```

Следующим шагом будет создание базы данных MongoDB для нашего приложения. Мы можем создать базу данных с помощью самого **Modulus**, **MongoHQ**, либо любого другого облачного провайдера.

Как только мы создали нашу базу данных, мы можем получить ее MONGO_URL (перейдите Dashboard > Databases > Select your database > Administration), и затем настроить наше приложение следующим образом:

```
$ modulus env set MONGO_URL "mongodb://<user>:<pass>@mongo.onmodulus.net:27017/<database_name>"
```

Теперь пришло время опубликовать приложение. Просто наберите:

```
$ modulus deploy
```

Все, наше приложение опубликовано на Modulus. Ознакомьтесь с **документацией** для получения большей информации о том, как получить доступ к логам, доменам и SSL.

Хотя новые облачные решения появляются чуть ли не каждый день, они зачастую имеют свои проблемы и ограничения. Так что на сегодняшний день лучшим production-решением для вашего Meteor-приложения является публикация на собственный сервер. Единственное, публикация на собственный сервер не будет столь простой, особенно если вы ждете действительно production-качества.

Meteor Up (или сокращенно mup) - это очередная попытка поправить ситуацию с помощью инструмента для командной строки, который берет на себя хлопоты по публикации. Давайте посмотрим, как опубликовать Microscope с помощью Meteor Up.

Прежде всего, нам потребуется сервер для публикации. Мы рекомендуем либо **Digital Ocean**, тарифы которого начинаются с 5$/мес, либо **AWS**, который бесплатно предоставляет т.н. micro instances (вы вскоре столкнетесь с необходимостью увеличения мощностей, но для начала работы с Meteor Up бесплатных будет вполне достаточно).

Какой бы сервис вы ни выбрали, вам нужно будет в итоге получить 3 вещи: IP-адрес вашего сервера, логин (обычно root или ubuntu) и пароль. Держите их под рукой, вскоре они нам понадобятся!

Для начала нам нужно установить Meteor Up с помощью ‘npm’:

```
$ npm install -g mup
```

Далее нужно создать отдельную директорию, которая будет содержать конфигурацию Meteor Up для текущей публикации. Мы используем отдельную директорию по двум причинам: во-первых, лучше избегать любых приватных ключей в Git репозитории, особенно открытом.

Во-вторых, используя несколько отдельных директорий, мы сможем параллельно использовать несколько конфигураций Meteor Up. Они могут пригодиться например, для публикации приложения на production и staging серверы.

Итак, давайте создадим новую директорию и инициализируем там Meteor Up проект:

```
$ mkdir ~/microscope-deploy
$ cd ~/microscope-deploy
$ mup init
```

> Отличный способ быть уверенным в том, что вы и ваша команда используете одни и те же настройки - это создать директорию конфигурации Meteor up в Dropbox или похожем сервисе.

Во время инициализации нового проекта, Meteor Up создаст два файла: ‘mup.json’ и ‘settings.json’.

‘mup.json’ будет содержать настройки для публикации, а ‘settings.json’, в свою очередь - все настройки, касающиеся нашего приложения (OAuth токены, токены аналитики и т.д.).

Следующим шагом будет конфигурация ‘mup.json’. Вот файл, по умолчанию созданный командой ‘mup init’, в котором все что вам нужно - это заполнить поля:

```
{
  //server authentication info
  "servers": [{
    "host": "hostname",
    "username": "root",
    "password": "password"
    //or pem file (ssh based authentication)
    //"pem": "~/.ssh/id_rsa"
  }],

  //install MongoDB in the server
  "setupMongo": true,

  //location of app (local directory)
  "app": "/path/to/the/app",

  //configure environmental
  "env": {
    "ROOT_URL": "http://supersite.com"
  }
}
```

mup.json

Давайте разберемся в этих настройках.

Server Authentication

Как вы могли заметить, Meteor Up поддерживает аутентификацию с помощью пароля и private key

(PEM), что делает возможным использование практически любой облачной платформы.

Важно: если вы предпочтете использовать аутентификацию с помощью пароля, сперва необходимо установить shpass (**прочтите здесь**).

Конфигурация MongoDB

Следующим шагом будет настройка MongoDB для нашего приложения. Мы рекомендуем использовать **MongoHQ**, либо любую другую облачную платформу, которая предоставляет профессиональную поддержку и высококлассные инструменты управления.

Если вы решили использовать MongoHQ, поставьте ‘false’ для ‘setupMongo’ и добавьте переменную окружения ‘MONGO_URL’ в блок ‘env’ вашего файла ‘mup.json’. Если же вы решили хостить MongoDB с помощью Meteor Up, то просто поставьте ‘true’ для setupMongo’, и Meteor Up сделает остальное за вас.

Путь к Meteor-приложению

Так как конфигурация Meteor Up находится в отдельной директории, нам нужно будет указать путь к нашему приложению с помощью свойства ‘app’. Просто поместите в значение этого свойства полный путь к директории с вашим приложением на локальной машине. Вы можете получить этот путь, набрав в терминале команду ‘pwd’, находясь в корневой директории вашего приложения.

Переменные окружения

Вы можете указать все ваши переменные окружения (напр. `ROOT_URL`, `MAIL_URL`, `MONGO_URL`) в блоке ‘env’.

Прежде чем мы опубликуем приложение, мы должны настроить сервер для обслуживания нашего Meteor-приложения. Волшебство Meteor Up заключается в том, что он инкапсулирует весь этот сложный процесс в одну команду!

```
$ mup setup
```

Это займет некоторое время, в зависимости от мощности вашего сервера и скорости соединения. После успешной настройки, мы, наконец, можем опубликовать наше приложение:

```
$ mup deploy
```

Эта команда соберет наше приложение и опубликует его на сервер, который мы только что настроили.

Логи - достаточно важная вещь, и Meteor Up предоставляет нам простой способ их вывести, эмулируя команду ‘tail -f’. Просто наберите:

```
$ mup logs -f
```

На этом заканчиваем наш обзор возможностей Meteor Up. Для большей информации, рекомендуем посетить **репозиторий Meteor Up**

Этих трех способов публикации Meteor-приложений достаточно для большинства возможных случаев. Конечно, многие из вас захотят иметь полный контроль и настроить Meteor сервер с нуля. Но это тема другого дня… или книги.

Чтобы облегчить разработку в Meteor, мы будем применять outside-in подход. Другими словами, в начале мы создадим обычный HTML/JavaScript шаблон, а затем подключим его к нашему приложению.

Это означает, что в этой главе мы коснемся только того, что происходит в директории `/client`.

Создадим новый файл `main.html` в директории `/client` и добавим в файл следующий код:

```
<head>
  <title>Microscope</title>
</head>
<body>
  <div class="container">
    <header class="navbar navbar-default" role="navigation">
      <div class="navbar-header">
        <a class="navbar-brand" href="/">Microscope</a>
      </div>
    </header>
    <div id="main" class="row-fluid">
      {{> postsList}}
    </div>
  </div>
</body>
```

client/main.html

Это будет наш главный шаблон приложения. Как вы заметили, это обычный HTML, кроме тега `{{> postsList}}`. Он является точкой входа шаблона `postsList`, который мы создадим позже. Теперь создадим еще пару шаблонов.

По своей сути, социально-новостной сайт состоит из сообщений, организованных в виде списка, именно так мы организуем наши шаблоны.

Теперь создадим директорию `/views` внутри `/client`. Туда мы положим все наши шаблоны. Также создадим директорию `/posts` внутри `/views`. Там будут находится шаблоны, которые связаны с новостями.

Meteor силен в поиске файлов. Не имеет значения, куда мы положили код внутри директории `/client`, Meteor найдет его и обработает должным образом. Это означает, что нет необходимости вручную прописывать пути к JavaScript и CSS файлам.

Другими словами, вы вполне можете хранить все ваши файлы в одном каталоге или же весь код в одном файле. Но так как Meteor в любом случае соберет все файлы в один, будет гораздо удобнее разбить все на части и держать код в чистоте, в виде хорошей и понятной структуры.

Итак, создадим наш второй шаблон. В директории `client/views/posts`, создадим файл `posts_list.html` со следующим содержанием:

```
<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}
  </div>
</template>
```

client/views/posts/posts_list.html

Также создадим файл `post_item.html` со следующим содержанием:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
    </div>
  </div>
</template>
```

client/views/posts/post_item.html

Обратите внимание на атрибут `name="postsList"` тега `<template>`. Это имя используется для того, чтобы Meteor понимал, какой и где шаблон используется. В этом случае шаблон будет внутри основного (его мы создали ранее), вместо тега `{{> postsList}}`.

Настало время освоить шаблонизатор, который используется в Meteor - **Handlebars**. Handlebars это простой HTML с добавлением трех вещей: *partials*, *expressions* и *block helpers*.

Partials - это конструкция вида `{{> имяШаблона}}`, с помощью которой мы обращаемся к Meteor и сообщаем ему, чтобы он заменил ее шаблоном с тем же именем (в нашем случае это `postItem`).

Expressions, такие как `{{title}}`, или вызывают свойство текущего объекта, или же возвращают значение, которое определено в текущем менеджере шаблонов (подробнее об этом позже).

Block helpers - это специальные теги, такие как `{{#each}}…{{/each}}` или `{{#if}}…{{/if}}`, с помощью которых можно создавать логические конструкции прямо в шаблоне.

> Вы можете зайти на **официальный сайт Handlebars** или на **этот удобный учебник** если хотите узнать больше о Handlebars.

Вооружившись этими знаниями, мы можем легко понять, что здесь происходит.

Во-первых, шаблон `postsList` мы выводим в цикле с помощью конструкции `{{#each}}…{{/each}}`. Для каждой итерации мы подключаем шаблон `postItem`.

Так откуда берется объект `posts`? Хороший вопрос. На самом деле это template helper, мы определим его позже, после того, как посмотрим на template managers.

Шаблон `postItem` довольно прост. Он использует только три выражения: `{{url}}` и `{{title}}`, которые возвращают свойства документа, и `{{domain}}`, который вызывает template helper.

Мы уже не раз упоминали выражение “template helpers” в этой главе, не объясняя, что он делает. Но для того, чтобы исправить это, мы должны сначала поговорить о менеджерах.

До сих пор мы имели дело с Handlebars, который является обычным HTML с дополнительными тегами. В отличие от других языков, таких как PHP (или даже обычных HTML страниц, которые могут содержать JavaScript), Meteor разделяет шаблоны от логики, и эти шаблоны ничего не делают сами по себе.

Для того, чтобы вдохнуть жизнь в шаблон, нам нужен **manager**. Вы можете думать о менеджере как о поваре, который получает сырье (ваши данные) и готовит их, прежде чем передать готовое блюдо официанту (шаблону), который принесет его вам.

Другими словами, в то время как роль шаблона ограничивается отображением или циклом с переменными, менеджер делает тяжелую работу по присвоению значения каждой переменной.

> Когда мы спросили у разработчиков Meteor, что они подразумевают под фразой `template managers`, то кто-то из них сказал, что это контроллеры, а кто-то сказал: “Это те файлы, куда я положил свой JavaScript код”.
>
> Менеджеры на самом деле не являются контроллерами (по крайней мере, не в смысле контроллеров в MVC) и “Те файлы, куда я положил свой JavaScript код” тоже не является истиной, поэтому мы отклонили оба ответа.
>
> Так как нам нужно было определиться с названием, мы придумали термин “менеджер” - подходящее слово, у которого нет множества других значений в веб-разработке.

Для простоты, мы будем называть файлы менеджеров шаблонов как и сами шаблоны, за исключением их расширения **.js**. Итак, создадим файл `posts_list.js` в директории `/client/views/posts` прямо сейчас и начнем создание нашего первого менеджера:

```
var postsData = [
  {
    title: 'Introducing Telescope',
    url: 'http://sachagreif.com/introducing-telescope/'
  },
  {
    title: 'Meteor',
    url: 'http://meteor.com'
  },
  {
    title: 'The Meteor Book',
    url: 'http://themeteorbook.com'
  }
];
Template.postsList.helpers({
  posts: postsData
});
```

client/views/posts/posts_list.js

Если вы все сделали правильно, теперь вы должны видеть что-то похожее в вашем браузере:

Our first templates with static data

Мы делаем две вещи. Во-первых, устанавливаем макет прототипа данных в виде массива, обозначенного как `postsData`. Эти данные, как правило, берутся из базы данных, но так как мы пока не знаем, как работать с базой (потерпите до следующей главы), то мы временно будем использовать статические данные.

Во-вторых, мы используем функцию `Template.myTemplate.helpers()` для определения шаблона хелпера `posts`, который просто возвращает наш массив `postsData`.

Определение хелпера `posts` означает, что он теперь доступен для использования в нашем шаблоне:

```
<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}
  </div>
</template>
```

client/views/posts/posts_list.html

Таким образом, наш шаблон может перебирать массив `postsData` и подставлять каждый элемента массива в цикле вместо тега `postItem` в шаблоне.

Added basic posts list template and static data.

Открыть на GitHub | Запустить копию

`domain`

Теперь создадим менеджер `post_item.js`, в котором будет храниться логика шаблона `postItem`:

```
Template.postItem.helpers({
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  }
});
```

client/views/posts/post_item.js

На этот раз наш хелпер `domain` является не массивом, а анонимной функцией. Этот паттерн является более распространенным (и более полезным) по сравнению с нашим предыдущим примером, где мы использовали массив вместо базы данных.

Displaying domains for each links.

Хелпер `domain` берет URL и возвращает домен, используя немного магии JavaScript. Но где же брать url?

Ответ на вопрос мы можем найти, вернувшись немного назад, к файлу шаблона `posts_list.html`. Блок `{{#each}}` не только перебирает наш массив, а еще и **устанавливает значение** `this` **внутри блока**.

Это означает, что между тэгами `{{#each}}`, каждый пост во время итерации обозначается как `this`, и это распространяется на все пути, которые подключены внутри менеджера

(`post_item.js`).

Теперь мы понимаем, почему `this.url` возвращает текущий URL поста (элемента массива postsData). Более того, если мы используем `{{title}}` и `{{url}}` внутри нашего шаблона `post_item.html` , Meteor понимает, что мы имеем в виду, написав `this.title` и `this.url` , и возвращает нам корректные значения.

Setup a `domain` helper on the `postItem`.

Открыть на GitHub

Запустить копию

Хотя это не является специфичным для Meteor, но вот краткое объяснение того, что происходило выше: это всего лишь немного “магии JavaScript”. Во-первых, мы создаем пустой элемент HTML (`a`) и храним его в памяти.

Затем мы устанавливаем ему атрибут `href` , равный нашему URL (как вы заметили ранее, он является объектом хелпера и вызывается через `this`).

Наконец, мы воспользуемся свойством `hostname` элемента `a` , чтобы получить ссылку только на домен, без остальной части URL.

Если вы сделали все без ошибок, то вы должны увидеть в браузере список сообщений. Этот список является всего лишь статическими данными. Мы пока не используем все прекрасные возможности real-time, которые предоставляет нам Meteor. Мы покажем вам, как использовать эти возможности в следующей главе!

Вы могли заметить, что нет необходимости обновлять страницу в браузере после каждого внесения правок в код.

Это потому что Meteor отслеживает все файлы в директории проекта и автоматически обновляет страницу, если обнаружит изменения в каком-либо файле.

Горячее обновление кода в Meteor устроено довольно умно, и может даже сохранять состояние вашего приложения между двумя обновлениями!

GitHub это социальный репозиторий для проектов с открытым исходным кодом, использующих **Git** для контроля версий исходного кода. Главная задача GitHub - сделать процесс разработки простым и увлекательным, в особенности когда над проектом одновременно работает несколько человек. Использование GitHub также позволяет научиться многим вещам. В этой главе мы пробежимся по некоторым фишкам GitHub, чтобы вам было легче следить за развитием событий книги.

Эта глава подразумевает, что вы малознакомы или совсем не знакомы с Git и GitHub. Если вы уже общаетесь с ними на “ты”, можете смело пропустить эту главу.

Базовым кирпичиком git репозитория является *коммит* (*commit*). Его можно представить как фотоснимок всего вашего кода в отдельный момент времени. Код со временем меняется, но если вы делаете коммиты, то к любому из них можно вернуться, отмотав время назад с помощью волшебства git.

Вместо того чтобы просто выдать вам законченный код от Microscope, мы делали снимки кода на каждом шаге написания этой книги. Все снимки доступны онлайн на GitHub.

Например, вот так выглядит **последний коммит предыдущей главы**:

A Git commit as shown on GitHub.

Вы видите “diff” (от слова “difference” - “разница”) файла `post_item.js`, или иными словами изменения, которые произошли с этим файлом в результате последнего коммита. В нашем случае мы создали файл `post_item.js` с чистого листа, поэтому всё его содержимое подсвечено зелёным.

Давайте посмотрим на другой пример из **будущей главы**:

Modifying code.

В этот раз зелёным подсвечены только те строчки кода, которые были изменены.

Иногда строчки кода не только добавляются или изменяются, но и **удаляются**:

Deleting code.

Мы только что познакомились с одной из фишек GitHub - легко просматривать код и находить в нем изменения.

По умолчанию git коммит показывает только изменения в коде, но иногда вам может захотеться посмотреть на файлы, которые *не* изменились. Просто чтобы удостовериться что их содержимое именно такое, каким вы ожидаете его увидеть.

GitHub поможет нам и в этом. Когда вы находитесь на странице коммита, нажмите **Browse code**:

The Browse code button.

Теперь у вас есть доступ к репозиторию в том виде, в котором он был *на момент этого коммита*:

The repository at commit 3-2.

На первый взгляд может оказаться не совсем ясно что мы смотрим на коммит. Но если мы сравним репозиторий с главной *мастер* веткой то сразу же будет видно что файлы выглядят по-другому:

The repository at commit 14-2.

Мы только что узнали как посмотреть весь код коммита на GitHub'е. Но что делать, если вам нужно посмотреть код одного из коммитов локально? Например, вам нужно откатить приложение во времени до определенного коммита, запустить его локально и посмотреть как оно себя поведет.

Для этого мы совершим наши первые шаги (по-крайней мере в этой книге) с командной утилитой `git`. Для начала удостоверьтесь что Git **установлен на вашей машине**. Затем **клонируйте** (то есть, загрузите локальную копию) репозитория Microscope следующей командой:

```
$ git clone git@github.com:DiscoverMeteor/Microscope.git github_microscope
```

Обратите внимание на `github_microscope` в конце команды - это название локальной папки, куда репозиторий будет клонирован. Если у вас уже есть папка `microscope`, просто выберите любое другое имя (оно не должно совпадать с именем Github репозитория).

Перейдем в созданную папку с помощью команды `cd`. Отсюда мы сможем начать использование утилиты `git`:

```
$ cd github_microscope
```

Теперь когда мы клонировали репозиторий, мы загрузили *весь* код приложения. Это означает что у

нас код финального приложения в момент самого последнего коммита.

К счастью, мы можем откатиться во времени на момент определенного коммита, не влияя при этом на все другие коммиты. Давайте попробуем:

```
$ git checkout chapter3-1
Note: checking out 'chapter3-1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at a004b56... Added basic posts list template and static data.
```

Git сообщает, что мы в режиме под названием “detached HEAD”. Это означает что с точки зрения Git мы можем смотреть на прошлые коммиты, но мы не можем их изменять. Это можно сравнить с волшебником, смотрящим на прошлое сквозь хрустальный шар.

(Стоит обратить внимание что у Git есть команды, которые позволяют *изменять* прошлые коммиты. Это будет примерно как если путешественник во времени отправляется в прошлое и наступает на бабочку. Мы оставим это команды за рамками этой главы.)

Причина по которой мы можем обратиться напрямую к `chapter3-1` является то, что мы отметили все коммиты Microscope правильными закладками с именами глав. Если бы этого не было сделано, вам пришлось бы узнать **хеш** коммита - уникальную опознавательную строку.

GitHub опять делает нашу жизнь чуть проще. Вы можете найти хеш коммита в нижнем правом углу синего заголовка, как показано на скриншоте:

Finding a commit hash.

Давайте попробуем команду `git checkout` с хешем вместо тега:

```
$ git checkout c7af59e425cd4e17c20cf99e51c8cd78f82c9932
Previous HEAD position was a004b56... Added basic posts list template and static data.
HEAD is now at c7af59e... Augmented the postsList route to take a limit
```

Наконец, что если нам нужно прекратить смотреть в хрустальный шар и вернуться к реальности? Для этого нужно сообщить Git, что мы желаем открыть ветку **master**:

```
$ git checkout master
```

Заметьте, что вы также можете запустить свое приложение командой `meteor` на любой стадии

процесса, даже когда вы находитесь в состоянии “detached HEAD”. Вам может понадобиться сначала запустить быструю команду `meteor update` если Meteor ругается на недостающие пакеты, так как код пакетов не включен в Git репозиторий Microscope.

Вот еще один случай: вы просматриваете файл и замечаете код, который раньше никогда не видели. Дело в том, что вы не можете вспомнить *когда* этот код был добавлен. Вы могли бы просматривать коммиты один за одним, пока не нашелся бы верный. Но есть способ проще. Встречайте кнопку History на GitHub'е.

Для начала откройте один из файлов в репозитории на GitHub. Затем найдите кнопку “History”:


GitHub's History button.

Теперь вам доступен лист всех коммитов, которые затрагивали этот файл:


Displaying a file's history.

Напоследок давайте обратим внимание на кнопку “Blame”:


GitHub's Blame button.

Эта страница показывает нам, кто изменил каждую линию файла, а также коммит, частью которого это изменение является. Другими словами - кого винить в том, что код больше не работает.


GitHub's Blame view.

Git довольно сложный инструмент - равно как и GitHub - и мы даже не пытаемся охватить все в рамках этой главы. На самом деле мы едва затронули все то, что возможно достичь с помощью этих двух инструментов. Но мы надеемся, что даже то немногое что мы успели рассмотреть, поможет вам при дальнейшем чтении этой книги.

В первой главе мы говорили о ключевой особенности Meteor - автоматической синхронизации данных между клиентом и сервером.

В этой главе мы рассмотрим подробнее, как это работает, а также разберем функционирование ключевой технологии Meteor, которая делает это возможным, - Meteor **Коллекции** (Collections).

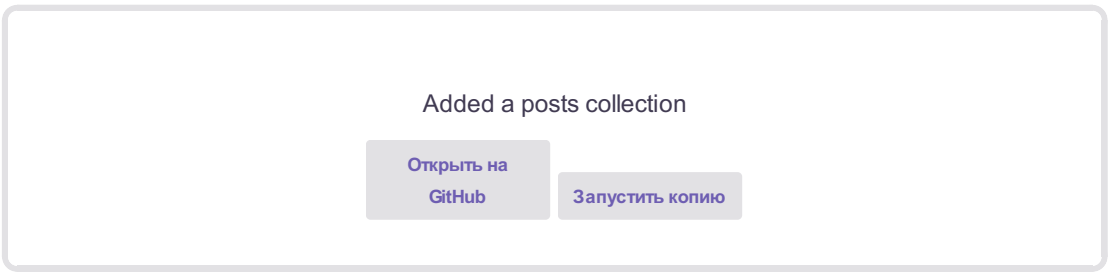
Коллекция в Meteor - это особая структура данных, ответственная за постоянное хранение данных на сервере в базе данных MongoDB и обеспечивающая синхронизацию в реальном времени этих данных с браузерами всех подключенных пользователей.

Мы хотим, чтобы наши посты были общими для всех пользователей и сохранялись на сервере, поэтому мы начнем с того, что создадим коллекцию под названием `Posts` для хранения постов.

Коллекции являются ключевым элементом любого приложения, и чтобы гарантировать их загрузку в первую очередь, мы положим их в директорию `lib`. Создайте поддиректорию ‘collections‘ внутри `lib`; а в ней создайте файл ‘posts.js` со следующим содержанием:

```
Posts = new Mongo.Collection('posts');
```

lib/collections/posts.js



В Meteor зона видимости переменной, определенной с помощью `var`, ограничена текущим файлом. Так как мы хотим, чтобы коллекция `Posts` была доступна для всего нашего приложения, мы *не будем* использовать это ключевое слово.

У веб-приложений есть три основных способа хранения данных, каждый из которых выполняет свою роль:

- **Память браузера:** такие типы данных, как переменные JavaScript хранятся в памяти браузера, что означает, что они *непостоянные*: они являются локальными по отношению к текущей вкладке браузера и исчезнут, как только вы ее закроете.

- **Браузерное хранилище:** браузеры также могут хранить данные более долгий срок используя куки (cookies) или **Локальное хранилище** (Local Storage). Несмотря на то, что эти данные сохраняются от сессии к сессии браузера, они являются *локальными* для текущего пользователя (хотя и доступны во всех вкладках), и их не так легко использовать совместно с другими пользователями.
- **Серверная база данных**: старая добрая база данных - это лучшее место для постоянного хранения данных, доступных для использования многими пользователями (MongoDB является БД по умолчанию для Meteor приложений).

Meteor использует все перечисленные методы и иногда синхронизирует данные между ними (как мы вскоре увидим). Но база данных остается “каноническим” источником данных, в котором хранится мастер-копия наших данных.

Код вне директорий `/server` и `/client` будет исполняться как на сервере, так и на клиенте, так что наша коллекция `Posts` будет доступна в *обоих* средах; однако поведение коллекции на сервере и на клиенте может отличаться.

На сервере коллекции поддерживают связь с MongoDB, производя чтение и запись любых изменений. В этом смысле коллекции можно сравнить с обычной библиотекой для работы с БД.

Тогда как коллекция на клиенте - это *выборка* из канонической коллекции на сервере. Коллекции на клиенте постоянно и незаметно (чаще всего) синхронизируются с коллекциями на сервере в реальном времени.

В этой главе мы начнем использовать **консоль браузера** (browser console); ее не нужно путать с **командной строкой ОС** (terminal), **командной строкой Meteor** (Meteor shell) или **командной строкой Mongo** (Mongo shell). Ниже краткое описание каждой из них.

Командная строка ОС

- Вызывается из оперционной системы.
- Результат команды `console.log()` **на сервере** отображается здесь.
- Обозначение: `$`.
- Также называют Shell или Bash.

Консоль браузера

- Открывается в браузере и исполняет JavaScript.
- Результат команды `console.log()` **на клиенте** отображается здесь.
- Обозначение: `❯`.
- Также называют консоль JavaScript, консоль разработчика, консоль Devtools.

Командная строка Meteor

- Вызывается из командной строки ОС при помощи команды `meteor shell`.
- Дает прямой доступ к серверному коду вашего приложения
- Обозначение: `>`.

Командная строка Mongo

- Открывается в терминале командами `meteor mongo`.
- Позволяет напрямую проводить операции с базой данных.
- Обозначение: `>`.
- Также называют консоль Mongo.

Заметьте, что вам не нужно вводить символ краткого обозначения (`$`, `❯`, или `>`) как часть команды. И как вы можете заметить, каждая строка, не начинающаяся с краткого обозначения, - это вывод результата предыдущей команды.

На сервере коллекции работают в качестве API для нашей базы MongoDB. Это позволяет нам выполнить команды вроде `Posts.insert()` или `Posts.update()` на сервере, которые произведут изменения в коллекции `posts` непосредственно в MongoDB.

Чтобы взглянуть поближе на нашу базу данных, откройте еще одно окно командной строки (в то время как сам процесс `meteor` исполняется в первом окне) и перейдите в директорию нашего приложения. Затем введите команду `meteor mongo` для запуска консоли Mongo, в которой мы сможем выполнять стандартные команды MongoDB (как обычно, выйти из консоли Mongo можно нажав `ctrl+c`). Для примера, давайте добавим новый пост:

```
meteor mongo

> db.posts.insert({title: "A new post"});

> db.posts.find();
{ "_id": ObjectId(".."), "title" : "A new post"};
```

Командная строка Mongo

> Если вы опубликовали ваше приложение на myApp.meteor.com, вы можете получить доступ к консоли Mongo вашего приложения при помощи команды `meteor mongo myApp`.
>
> И раз уж мы говорим про опубликованное приложение, вывести логи с сервера можно набрав `meteor logs myApp`.

Синтаксис Mongo многим знаком, так как он использует JavaScript. Мы не будем дальше работать с нашей БД при помощи консоли Mongo, но иногда уместно туда зайти, чтобы проверить, в каком состоянии сейчас находится MongoDB.

Гораздо интереснее обстоят дела с коллекциями на клиенте. Когда вы пишете `Posts = new Mongo.Collection('posts');` на клиенте, вы создаете *локальную браузерную кэш-копию* настоящей коллекции Mongo. Когда мы говорим, что коллекция на клиенте - это «кэш», мы подразумеваем то, что она содержит *выборку* части данных из БД и предоставляет *очень быстрый* доступ к этим данным.

Важно понимать, что это одна из фундаментальных особенностей Meteor. В общем и целом, коллекция на клиенте состоит из *частичной выборки* из документов коллекции Mongo на сервере (ведь мы не хотим отправлять на клиент *всю* базу данных целиком).

Также важно то, что коллекции на клиенте хранятся *в памяти браузера*, а это значит, что мы можем получить к ним доступ практически мгновенно. Больше никаких медленных запросов на сервер, чтобы получить данные из БД, так как вызывая, скажем, метод `Posts.find()` на клиенте, мы работаем с уже предварительно загруженными данными.

> Версия Mongo на клиенте в Meteor называется MiniMongo. Пока технология еще не доведена до совершенства, и есть некоторые функции MongoDB, которые не будут работать в MiniMongo. Несмотря на это, все функции, которые мы затрагиваем в этой книге, работают как в MongoDB, так и в MiniMongo.

Важнейшая часть всего этого процесса - сам способ, при помощи которого коллекция на клиенте синхронизирует свои данные с одноименной коллекцией на сервере (в нашем случае `'posts'`).

Вместо того чтобы объяснять все в деталях, давайте просто посмотрим, что происходит.

Начнем с того, что откроем два окна браузера, и в каждом из них откроем консоль JavaScript. Далее, запускаем консоль Mongo в командной строке.

Сейчас мы должны увидеть единственный документ, который мы создали ранее, во всех трех открытых консолях (обратите внимание, что *пользовательский интерфейс* нашего приложения все еще показывает три предыдущих поста. На время мы их проигнорируем).

```
> db.posts.find();
{title: "A new post", _id: ObjectId("..")};
```

Командная строка Mongo

```
❯ Posts.findOne();
{title: "A new post", _id: LocalCollection._ObjectID};
```

Консоль первого окна браузера

Теперь в одном из окон браузера давайте создадим новый пост, набрав команду:

```
❯ Posts.find().count();
1
❯ Posts.insert({title: "A second post"});
'xxx'
❯ Posts.find().count();
2
```

Консоль первого окна браузера

Пост появился в локальной коллекции на клиенте. Давайте проверим MongoDB:

```
❯ db.posts.find();
{title: "A new post", _id: ObjectId("..")};
{title: "A second post", _id: 'yyy'};
```

Командная строка Mongo

Как вы видите, пост также появился и в MongoDB, при этом мы не написали ни строчки кода для этого (ну, строго говоря, мы все же написали *одну* строчку: `new Mongo.Collection('posts')`). Но это еще не все!

Введите в консоли другого окна браузера:

```
❯ Posts.find().count();
2
```

Консоль второго окна браузера

Этот пост доступен и там! Даже несмотря на то, что мы не обновляли это окно и уж тем более не писали никакого кода для того, чтобы он там появился. Все случилось само собой, как по волшебству; и к тому же мгновенно. Далее мы поймем, каким образом все это осуществилось.

А произошло следующее: коллекция на клиенте сообщила коллекции на сервере, что у нее появился новый пост, а коллекция на сервере добавила этот пост непосредственно в базу данных Mongo; и разослала его всем остальным коллекциям `post` на открытых в данный момент клиентах.

Получать документы через браузерную консоль - не особо полезное занятие. Скоро мы научимся, как связывать данные с шаблонами, и превратим наш простой HTML прототип в полностью функциональное приложение, обновляющееся в реальном времени.

Просто отображать коллекции через браузерную консоль это не совсем то, что нам нужно. То, что мы действительно хотим - это отображать сами данные и изменения в этих данных прямо на экране. Этим мы превратим наше приложение из простой *статичной страницы* в настоящее *веб-приложение* с динамическими, постоянно меняющимися данными, и обновляющееся в реальном времени.

Первое, что мы сделаем - это создадим некоторые записи в нашей БД. Для этого создадим особый файл, который загрузит специально подготовленные нами данные в коллекцию `Posts`, когда сервер впервые запустится.

Для начала давайте убедимся, что наша база данных пуста. Для этого используем команду `meteor reset`, которая сотрет нашу БД и перезапустит проект. Само собой, нужно быть очень осторожными с этой командой, как только вы начнете работать над реальным проектом.

Остановите Meteor сервер, нажав в командной строке `ctrl-c`, и затем введите команду:

```
meteor reset
```

Эта команда полностью очистит нашу базу данных. Эта полезная команда в процессе разработки, когда велика вероятность того, что база данных придет в непригодное состояние.

Давайте опять запустим наше Meteor приложение:

```
meteor
```

Теперь, когда наша база пуста, мы можем добавить в наше приложение следующий код, который при запуске сервера загрузит три поста в коллекцию `Posts` в том в случае, если она пуста:

```
if (Posts.find().count() === 0) {
  Posts.insert({
    title: 'Introducing Telescope',
    url: 'http://sachagreif.com/introducing-telescope/'
  });

  Posts.insert({
    title: 'Meteor',
    url: 'http://meteor.com'
  });

  Posts.insert({
    title: 'The Meteor Book',
    url: 'http://themeteorbook.com'
  });
}
```

server/fixtures.js

Added data to the posts collection.

Открыть на GitHub | Запустить копию

Мы поместили этот файл в директорию `server/`, так что он никогда не будет загружен в браузер пользователя. Код исполнится сразу же после того, как сервер будет запущен, и добавит три простых поста в нашу коллекцию `Posts` при помощи метода `insert`.

Теперь снова запустите сервер командой `meteor` и эти три поста будут загружены в базу данных.

Теперь, открыв браузерную консоль, мы увидим, что все три поста загружены также и в MiniMongo:

```
> Posts.find().fetch();
```

Консоль браузера

Чтобы отразить эти данные на странице, мы воспользуемся помощью нашего друга под названием “метод шаблона” (template helper).

В Главе 3 мы видели, каким образом Meteor позволяет привязывать *контекст данных* к нашим шаблонам Spacebars, чтобы построить HTML представление простых структур данных. Мы можем аналогичным образом привязать данные из нашей коллекции. Мы просто заменим наш статичный JavaScript объект `postsData` на динамическую коллекцию.

Для этого можете смело удалить код с ‘postsData’. Вот как теперь должен выглядеть файл `posts_list.js`:

```
Template.postsList.helpers({
  posts: function() {
    return Posts.find();
  }
});
```

client/templates/posts/posts_list.js

Wired collection into `postsList` template.

Открыть на GitHub | Запустить копию

В Meteor метод `find()` возвращает *курсор*(cursor), который является **реактивным источником данных** (reactive data source). Когда мы хотим получить его содержимое, мы можем использовать на нем метод `fetch()`, который трансформирует содержимое курсора в массив.

Внутри приложения Meteor достаточно умен, чтобы знать, как перебирать содержимое курсоров не трансформируя их в массивы. Поэтому вы не часто сможете встретить `fetch()` в коде Meteor приложения (по этой же причине мы не использовали его в нашем коде выше).

Теперь вместо того, чтобы получать список постов в виде статичного массива, мы возвращаем курсор, указывающий на наш метод ‘posts’ (правда, на экране ничего особо не изменится, т.к. мы по прежнему используем одни и те же данные):

Использование динамических данных

Наш метод `{{#each}}` перебрал все документы в коллекции Posts и вывел их на экран. Коллекция на сервере получила посты из MongoDB, передала их в коллекцию на клиенте и далее наш Spacebars метод передал эти данные в шаблон.

Давайте пойдем еще дальше, добавив еще один пост через консоль:

```
❯ Posts.insert({
  title: 'Meteor Docs',
  author: 'Tom Coleman',
  url: 'http://docs.meteor.com'
});
```

Консоль браузера

Теперь вернемся в браузер и увидим следующее:

Добавление постов через консоль

Вы только что впервые увидели реактивность в действии. Когда мы сказали Spacebars методу перебрать данные внутри курсора `Posts.find()`, он также начал следить за состоянием этого курсора; в случае его изменения он будет обновлять наш HTML, отображая на экране актуальные данные.

В нашем случае, самый простой способ внести изменения - добавить еще один `<div class="post">...</div>`. Если вы хотите убедиться, что Meteor действительно сделал именно это, то откройте закладку DOM Inspector в окне инструменты разработчика вашего браузера и выберите `<div>`, относящийся к любому посту.

Далее добавьте с помощью консоли браузера еще один пост. Когда вы вернетесь обратно на экран DOM Inspector, вы увидите еще один `<div>`, относящийся к новому посту, но при этом у вас останется выбранным *тот же самый* первоначальный `<div>`. Это удобный способ проверить, какой элемент был обновлен, а какой остался нетронутым.

До этого момента, у нас был подлючен пакет `autopublish`, который не предназначен для готовых приложений. Как можно понять из его названия, этот пакет просто говорит приложению, что каждая серверная коллекция должна быть полностью доступна каждому подключенному клиенту (опубликована). Это совсем не то, чего бы нам хотелось, так что давайте его отключим.

Откройте новое окно командной строки и введите:

```
$ meteor remove autopublish
```

Это произведет мгновенный эффект. Если вы сейчас откроете браузер, то увидите, что все наши посты исчезли! Причина этому - мы использовали пакет `autopublish` для того, чтобы полностью копировать данные из нашей БД в коллекцию на клиенте.

Рано или поздно нам придется проконтролировать, что мы передаем клиенту только те посты, которые пользователь должен видеть (при этом принимая во внимание такие вещи, как нумерация страниц (pagination)). Но сейчас мы настроим коллекцию `Posts` таким образом, чтобы она публиковалась целиком.

Для этого мы создадим функцию `publish()`, которая возвращает курсор со ссылкой на все посты (публикация):

```
Meteor.publish('posts', function() {
  return Posts.find();
});
```

server/publications.js

На клиенте, в свою очередь, мы должны *подписаться* на эту публикацию. Просто добавьте следующую строку в файл `main.js`:

```
Meteor.subscribe('posts');
```

client/main.js

Removed `autopublish` and set up a basic publication.

Открыть на GitHub | Запустить копию

Если мы снова проверим браузер, то увидим, что наши посты вернулись. Ура!

Чего же мы в итоге добились? Ну, хотя у нас пока и нет пользовательского интерфейса, наше приложение уже полностью функционально. Мы можем опубликовать его в интернете и начать добавлять новые посты (используя консоль браузера), которые будут появляться в браузерах других пользователей по всему миру.

Система публикаций и подписок - это одна из главных концепций Meteor, и с первого раза иногда бывает сложно понять ее полностью.

Это приводит ко многим заблуждениям, таким как уверенность в том, что Meteor - небезопасен, или что Meteor-приложения не в состоянии справляться с большим количеством данных.

Главная причина того, что людям эти вещи сперва казались непонятными - это именно то «волшебство», что Meteor делает за нас. Хотя это самое волшебство на деле крайне полезно, за ним не видно процессов, происходящих за кулисами (на то оно и волшебство). Так что давайте попробуем заглянуть по-глубже и понять, что же на самом деле происходит.

Сперва давайте вернемся в старый добрый 2011 год, когда Meteor еще не существовал. Представим, что вы решили создать простое приложение на Ruby on Rails. Когда пользователь заходит на сайт, клиент (его браузер) посылает запрос на сервер, где живет ваше приложение.

Первое, что в этом случае должно сделать приложение - это понять, какие данные пользователь должен увидеть. Это может быть, например, 12-ая страница поиска, или страница профиля некой Маши, или последние 20 твитов Боба и т.д. Это похоже на то, как продавец книжного магазина ищет на полках книгу, которую вы попросили.

Как только нужная информация найдена, следующее, что должно сделать приложение - это предоставить эту информацию в виде HTML (или JSON, в случае API).

Если опять проводить параллели с книжным магазином, то наш воображаемый продавец на данном этапе оборачивает книгу, которую вы купили и кладет ее в пакет. Это и называется «View» в популярной архитектуре «Model-View-Controller».

Наконец, приложение отправляет готовый HTML в браузер пользователя. Работа приложения выполнена, и теперь оно может, откинувшись, глотнуть пива, пока не придет следующий запрос от браузера.

Теперь давайте посмотрим, что же такое особенное делает Meteor, по сравнению с предыдущим подходом. Как мы уже поняли, ключевая инновация Meteor состоит в том, что, в то время, как приложение Ruby on Rails живет только на сервере, он имеет еще и клиентскую часть, которая исполняется в браузере.

Отправляем подмножество данных клиенту.

Это как если бы в нашем предыдущем примере, продавец не просто бы нашел вам нужную книгу, но также пришел бы к вам домой, чтобы почитать ее вам на ночь (признаем, звучит жутковато).

Такая архитектура позволяет Meteor делать много классных вещей, главная из которых - то, что по сути мы имеем базу данных и на сервере, и на клиенте. В общих чертах, Meteor просто берет часть вашей базы данных и копирует ее на клиент.

Такой подход имеет 2 больших отличия: во-первых, вместо того, чтобы отправлять уже готовый HTML, Meteor отправляет собственно сами данные, и дает коду на клиенте возможность самому распоряжаться этими данными как он пожелает (data on a wire). Во-вторых, операции с этими данными будут происходить практически мгновенно, так как вам не нужно будет постоянно слать запросы на сервер (latency compensation).

База данных приложения может иметь десятки тысяч записей, некоторые из которых могут быть приватными или крайне важными. Разумеется, мы не будем отправлять на клиент всю нашу базу данных, т.к. в большинстве случаев это плохо отразится на безопасности и производительности.

Так что нам нужен способ, с помощью которого мы сможем отправлять на клиент только необходимые данные. Этот способ - публикации (publications).

Давайте вернемся к Microscope. Вот все наши посты, которые есть в БД:

Все посты в нашей базе данных.

Хотя этот функционал мы еще не реализовали, давайте все же представим, что некоторые посты были отмечены как оскорбительные. Хотя мы и не будем их удалять из самой БД, они все же не должны показываться пользователям (те не нужно отправлять их на клиент).

Первое, что мы должны сделать - это сообщить Meteor, какие данные мы хотим отправить на клиент. В данном случае - что мы хотим отправить только те посты, что не отмечены как оскорбительные.

Исключаем помеченные посты.

Вот серверный код, который отвечает за это:

```
// on the server
Meteor.publish('posts', function() {
  return Posts.find({flagged: false});
});
```

Этот код обезопасит нас от того, что клиент будут иметь доступ к отмеченным постам. Это именно тот способ, которым можно сделать Meteor-приложения безопасными: просто убедитесь, что вы публикуете только те данные, к которым вы хотите предоставить доступ на клиенте.

Базово можно рассматривать публикации и подписки как трубу, через которую данные из коллекции на сервере перетекают в коллекцию на клиенте.

Протокол, который собственно и является такой трубой, называется DDP (Distributed Data Protocol). Чтобы узнать больше о нем, вы можете посмотреть **вот эту речь** одного из основателей Meteor Matt DeBergalis, либо **этот скринкаст** за авторством Chris Mather, который более детально раскроет вам основные концепции этого протокола.

Даже несмотря на то, что мы хотим отправить непомеченные посты клиенту, мы все же не можем отправлять их тысячами сразу. Нам нужен способ, с помощью которого клиент может указать, какая часть данных ему нужна в данный момент. Этот способ и есть подписки.

Любые данные, на которые вы подписываетесь, будут отражены на клиенте благодаря MiniMongo.

К примеру, давайте предположим, что мы просматриваем профиль некоего Bob Smith, и хотим, чтобы отображались только его посты.

Подписываясь на посты Bob'a мы реплицируем их на клиенте.

Сперва, мы немного изменим нашу публикацию так, чтобы она принимала параметр:

```
// on the server
Meteor.publish('posts', function(author) {
  return Posts.find({flagged: false, author: author});
});
```

Далее мы укажем этот параметр уже в подписке:

```
// on the client
Meteor.subscribe('posts', 'bob-smith');
```

Вот как вы можете сделать Meteor-приложение гибким на клиенте: вместо того, чтобы подписываться на все доступные данные, вы получаете только те данные, которые вам сейчас нужны. Таким образом, вы избегаете перегрузки памяти браузера, независимо от размеров вашей базы данных на клиенте.

Теперь посты Bob’a, которые мы получили, распределены по всем категориям (например: «JavaScript», «Ruby» и «Python»). Может мы и хотим загрузить все его посты в память браузера,

прямо сейчас мы хотим отображать только те, что принадлежат категории «JavaScript». Как раз вот здесь и нужна выборка:

Выделяем подмножество документов на клиенте

Также как и на сервере, мы будем использовать для этого метод `Posts.find()`.

```
// on the client
Template.posts.helpers({
  posts: function(){
    return Posts.find({author: 'bob-smith', category: 'JavaScript'});
  }
});
```

Теперь, когда мы имеем четкое представление роли публикаций и подписок в Meteor-приложении, давайте пойдем еще дальше и посмотрим несколько способов реализации этого механизма.

Если вы создаете Meteor-приложение с нуля (используя meteor create), оно будет автоматически включать в себя пакет `autopublish`. Для начала, давайте разберем, что же именно он делает.

Главная назначение модуля `autopublish` - это сделать максимально простым и удобным начало процесса разработки вашего Meteor-приложения. Он автоматически отражает все серверные данные на клиенте, позаботившись за вас о публикациях и подписках.

Autopublish

Как это работает? Если у вас есть коллекция под названием `posts` на сервере, то пакет `autopublish` автоматически отправит каждый пост, который он найдет в этой коллекции MongoDB в одноименную коллекцию на клиенте (если она конечно существует).

Если вы используете `autopublish`, то вам не нужно думать о публикациях: данные доступны везде, все просто. Конечно же, мы не будем копировать всю нашу базу данных в кэш браузеров всех пользователей, т.к. в конечном итоге, это приведет к огромным проблемам.

По-этому, использование `autopublish` имеет смысл только в начале процесса разработки, пока вы еще не думаете о публикациях.

Как только вы уберете пакет `autopublish`, вы увидете, что все данные исчезли с клиента. Самый простой способ вернуть их - это просто воспроизвести то же, что делает `autopublish` - опубликовать коллекцию целиком. Например:

```
Meteor.publish('allPosts', function(){
  return Posts.find();
});
```

Публикация полной коллекции

Да, все еще публикуем коллекции целиком, но по крайней мере мы можем контролировать, какие коллекции мы публикуем, а какие нет. В данном случае, мы публикуем коллекцию `Posts`, но не публикуем `Comments`.

Следующий уровень контроля - публикация только части коллекции. К примеру, только постов, принадлежащих конкретному автору:

```
Meteor.publish('somePosts', function(){
  return Posts.find({'author':'Tom'});
});
```

Частичная публикация коллекции

Если вы читали документацию Meteor **по этой теме**, вы должно быть знаете об использовании методов `added()` или `ready()` для того, чтобы задавать атрибуты записей на клиенте и изо всех сил пытались понять, почему же во всех Meteor-приложениях, которые вы видели, они никогда не используются.

Причина в том, что Meteor предоставляет очень полезный метод: `_publishCursor()`. Вы также его еще не встречали, верно? Но это именно тот метод, который Meteor использует, когда вы возвращаете курсор публикации (например: `Posts.find({'author': 'Tom'})`).

Когда Meteor видит, что публикация `somePosts` возвращает курсор, он вызывает метод `_publishCursor()`, чтобы (как вы и догадались) опубликовать курсор автоматически.

Вот что делает этот метод:

- Проверяет имя коллекции на сервере.
- Достает все соответствующие документы из курсора и отправляет их в одноименную коллекцию на клиенте. (Он использует метод `.added()` для этого).
- Каждый раз, когда документ добавлен, удален или изменен, он сообщает об этих изменениях коллекции на клиенте, используя метод `.observe()` на курсоре и `.added()`, `.changed()` или `.removed()`.

В данном примере мы можем быть уверенны в том, что пользователь получает только те посты, в которых он заинтересован (те, что написал Tom), и они доступны в кэше его браузера.

Теперь мы знаем, как можно публиковать только часть всех наших постов, но мы можем пойти еще дальше! Давайте посмотрим, как публиковать лишь отдельные свойства документов.

Как и в прошлый раз, мы используем метод `.find()` чтобы вернуть курсор, но на этот раз мы исключим некоторые свойства:

```
Meteor.publish('allPosts', function(){
  return Posts.find({}, {fields: {
    date: false
  }});
});
```

Частичная публикация свойств (полей)

Конечно, мы можем совмещать обе техники. Например, если мы хотим получить все посты от Tom'а, но при этом оставляя в стороне даты, мы напишем:

```
Meteor.publish('allPosts', function(){
  return Posts.find({'author':'Tom'}, {fields: {
    date: false
  }});
});
```

Итак, мы прошли путь от публикации всех свойств всех документов всех коллекций (используя `autopublish`) к публикации части свойств части документов части коллекций.

Мы поняли чего можно добиться, используя публикации Meteor, и этих техник вполне достаточно в большинстве случаев.

Иногда вы вынуждены идти еще дальше, объединяя или совмещая публикации. Мы это затронем в следующей главе!

Теперь, когда мы имеем список постов (которые в конце концов отправят пользователи), нам нужна отдельная страница для каждого поста, где наши пользователи смогут обсудить его.

Мы бы хотели, чтобы эти страницы были доступны через *постоянную ссылку* (*permalink*) – URL вида `http://myapp.com/posts/xyz` (где `xyz` – идентификатор `_id` в MongoDB), которая уникальна для каждого поста.

Для этого нам понадобится какая-нибудь *маршрутизация (routing)*, чтобы адресная строка в браузере правильно соответствовала отображаемому контенту.

Iron Router – это пакет маршрутизации, который был задуман специально для Meteor приложений.

Он не только помогает в маршрутизации (настройке путей), но может и позаботиться о фильтрации (сопоставлении действий и некоторых путей), и даже управлять подписками (контролировать, какой путь имеет доступ к этим данным). (Примечание: Iron Router частично был разработан Tom Coleman – соавтором книги *Discover Meteor*.)

Во-первых, установим пакет из Atmosphere:

```
$ meteor add iron:router
```

Terminal

Эта команда скачает и установит пакет Iron Router для использования вашим приложением. Заметьте, что вам иногда нужно будет перезапустить Meteor-приложение (с помощью `ctrl+c` убить процесс, затем `meteor` чтобы запустить его снова) перед тем, как пакет может быть использован.

В этой главе мы коснёмся большинства особенностей маршрутизации. Если вы уже имеете некоторый опыт с фреймворками, такими как Rails, вам будет знакомо большинство из этих концепций. На случай, если это не так, здесь приводится краткий словарик для быстрого ознакомления:

- **Маршруты (Routes)**: Основной строительный блок маршрутизации. Это в основном набор инструкций, которые говорят куда идти и что делать при встрече с данным URL-ом.
- **Пути (Paths)**: URL внутри вашего приложения. Он может быть статичным (`/terms_of_service`) или динамичным (`/posts/xyz`), и даже включать параметры запроса (`/search?keyword=meteor`).
- **Сегменты (Segments)**: Различные части пути, разделенные прямым слэшем (`/`).
- **Обработчики (Hooks)**: Действия, которые вы захотите произвести перед, после, или даже во время процесса маршрутизации. Типичным примером может быть проверка прав пользователя перед отображением страницы.
- **Фильтры (Filters)**: Простые обработчики, которые вы глобально определяете для одного или нескольких маршрутов.
- **Шаблоны маршрутов (Route Templates)**: Каждому маршруту нужно указать шаблон. Если вы его не укажете, маршрутизатор будет искать шаблон с таким же именем, как у маршрута по умолчанию.
- **Макеты (Layouts)**: Вы можете думать о макетах, как о цифровых фоторамках. Они содержат в себе весь html-код, который оборачивается текущим шаблоном, и будут оставаться ими же, даже если шаблон изменится.
- **Контроллеры (Controllers)**: Иногда вы будете понимать, что многие ваши маршруты повторно используют одни и те же параметры. Вместо дублирования кода вы можете наследовать такие маршруты от одного *маршрутного контроллера (routing controller)*, который будет содержать всю логику маршрутизации.

Для более подробной информации об Iron Router смотрите **полную документацию на GitHub**.

До сих пор мы делали сборку нашего макета, используя жёстко заданные вставки шаблонов (такие как `{{>postsList}}`). Таким образом, контент нашего приложения может меняться, но основная структура страницы всегда одинакова: заголовок со списком постов ниже.

Iron Router позволяет нам уйти от этой замшелости взятием на себя отрисовки содержимого html-тега `<body>` . Поэтому мы не будем определять содержимое `<body>` сами, как мы делали это с обычной html-страницей. Вместо этого мы укажем маршрут к специальному макету, который содержит метод шаблона `{{> yield}}` .

Метод `{{> yield}}` определит специальную динамическую зону, которая автоматически будет отрисовывать шаблон, соответствующий текущему маршруту (договоримся, что с этого места такие специальные шаблоны мы будем называть «маршрутными шаблонами» – «route templates»):

Layouts and templates.

Начнем с создания нашего макета и добавления метода `{{> yield}}`. Прежде всего удалим наш html-тег `<body>` из `main.html` и переместим его контент в шаблон `layout.html`.

Итак, наш похудевший `main.html` теперь выглядит так:

```
<head>
  <title>Microscope</title>
</head>
```

client/main.html

В то же время вновь созданный `layout.html` теперь будет содержать внешний макет приложения:

```
<template name="layout">
  <div class="container">
    <header class="navbar">
      <div class="navbar-inner">
        <a class="brand" href="/">Microscope</a>
      </div>
    </header>
    <div id="main" class="row-fluid">
      {{> yield}}
    </div>
  </div>
</template>
```

client/views/application/layout.html

Как вы можете заметить, мы заменили включение шаблона `postsList` вызовом метода `yield`. Вы заметите, что после этого изменения мы ничего не увидим на экране. Это потому что мы еще не сказали маршрутизатору, что делать с URL `/`, поэтому он просто выдаёт пустой шаблон.

Для начала мы можем вернуть наше старое поведение, сделав соответствие корневого URL `/` шаблону `postsList`. В корне нашего проекта создадим каталог `/lib`, а внутри него - файл `router.js`:

```
Router.configure({
  layoutTemplate: 'layout'
});

Router.map(function() {
  this.route('postsList', {path: '/'});
});
```

lib/router.js

Мы сделали две важные вещи. Во-первых, указали маршрутизатору, какой макет использовать по умолчанию. Во-вторых, определили новый маршрут `postsList`, соответствующий пути `/`.

`/lib`

Всё, что находится в папке `/lib`, гарантированно загрузится первым - прежде, чем всё остальное в вашем приложении (за возможным исключением умных пакетов). Это делает папку `/lib` отличным место для любого вспомогательного кода, который должен быть доступен всё время.

Небольшое предупреждение: заметим, что поскольку папки `/lib` нет внутри папок `/client` или `/server` это означает, что её контент будет доступен для обоих окружений.

Проясним здесь некоторые моменты. Мы назвали наш маршрут `postsList`, но мы также дали имя `postsList` и *шаблону*. Что же происходит в таком случае?

По умолчанию Iron Router ищет шаблон с таким же именем как и маршрут. По факту он будет даже искать *путь (path)*, основанный на имени маршрута, – это означает, что мы, не определив свой путь (который мы указываем опцией `path` в нашем описании маршрута), сделали наш шаблон по умолчанию доступным по URL `/postsList`.

Вам может быть интересно, почему вообще маршрутам нужно давать имена. Именованные маршруты позволяют использовать несколько особенностей Iron Router, чтобы облегчить создание ссылок внутри приложения. Один из самых полезных методов в Handlebars – это `{{pathFor}}`, который возвращает компонент URL путь любого маршрута.

Мы хотим, чтобы наша ссылка на главную страницу указывала на список постов. Вместо того чтобы определить статический URL `/`, мы можем использовать метод в Handlebars. Конечный результат будет такой же, но это даст нам больше гибкости, так как этот метод будет всегда выводить правильный URL, даже если мы изменим путь маршрута в маршрутизаторе.

```
<header class="navbar">
  <div class="navbar-inner">
    <a class="brand" href="{{pathFor 'postsList'}}">Microscope</a>
  </div>
</header>

//...
```

client/views/application/layout.html

Very basic routing.

Открыть на GitHub | Запустить копию

Если провести развертывание текущей версии приложения (или запустите экземпляр, используя ссылку выше), можно заметить, что при загрузке страницы список постов некоторое время отображается пустым. Это происходит, потому что пока подписчик `posts` не закончит забирать данные с сервера, постов для отображения на странице попросту нет.

Было бы намного лучше в таких случаях обеспечить пользователю визуальную обратную связь с происходящим.

К счастью, Iron Router даёт простой способ сделать это – мы можем воспользоваться подписчиком `waitOn`:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.map(function() {
  this.route('postsList', {path: '/'});
});

Router.onBeforeAction('loading');
```

lib/router.js

Разберемся с этим кодом. Во-первых, мы изменили блок `Router.configure()`, обеспечив маршрутизатор именем загрузочного шаблона (который мы скоро создадим) для перенаправления на него, пока наше приложение ожидает данные.

Во-вторых, мы добавили функцию `waitOn`, которая возвращает нашу `posts`-подписку. Это означает, что подписка `posts` гарантированно подгрузится перед отправкой пользователя по запрошенному маршруту.

Заметим, что поскольку мы определяем нашу функцию `waitOn` глобально на уровне маршрутизации, она сработает, когда пользователь впервые зайдёт в ваше приложение. После этого данные уже будут загружены в память браузера, и маршрутизатору не нужно будет ждать их снова.

Так как мы сейчас позволили маршрутизатору обрабатывать нашу подписку, её можно безопасно удалить из `main.js` (который теперь будет пустым).

Как правило, это хорошая идея – ожидать ваши подписки – не только для взаимодействия с пользователем, но и потому что так вы можете быть уверены, что данные всегда будут доступны в шаблоне. Это устраняет необходимость иметь дело с шаблонами, начинающими отрисовку перед тем, как их данные будут доступны.

Также мы добавим фильтр `onBeforeAction`, чтобы запустить встроенный `loading` триггер Iron Router'а и показать шаблон загрузки, пока мы ждем.

Финальным кусочком головоломки будет сам шаблон процесса загрузки. Для создания прекрасного анимированного индикатора загрузки мы воспользуемся пакетом `sacha:spin`. Добавим его командой `meteor add sacha:spin` и создадим шаблон `loading`:

```
<template name="loading">
  {{>spinner}}
</template>
```

client/views/includes/loading.html

Заметьте, что `{{>spinner}}` частично содержится в `spin` пакете. Даже если эта часть приходит “извне” нашего приложения, мы можем вставлять его также, как и любой другой шаблон.

Wait on the post subscription.

Открыть на GitHub Запустить копию

Реактивность – это базовая часть Meteor, и хотя мы еще по настоящему не касались её, наш шаблон загрузки даёт первый взгляд на эту концепцию.

Перенаправление на шаблон загрузки, пока данные загружаются, это, конечно же, хорошо. Но как маршрутизатор узнал, что нужно перенаправить пользователя *обратно* на правильную страницу, как только данные были получены?

На данный момент просто скажем, что здесь однозначно работает реактивность. Но не беспокойтесь, вы скоро изучите её.

Теперь, когда мы увидели как указывать маршрут к шаблону `postsList`, давайте построим маршрут для отображения подробного вида одного поста.

Есть только одна загвоздка: мы не можем определить свой отдельный маршрут для каждого поста – ведь их будет огромное множество. Вместо этого нам стоит создать один *динамический* маршрут, и заставить его показывать любой пост.

Для начала мы создадим новый шаблон, который просто отрисовывает такой же шаблон поста, что мы использовали ранее в списке постов.

```
<template name="postPage">
  {{> postItem}}
</template>
```

client/views/posts/post_page.html

Мы добавим больше элементов (таких, как комментарии) к этому шаблону позже, но на данный момент он будет служить простой оболочкой для нашей вставки `{{> postItem}}`.

Теперь создадим другой именованный маршрут – на этот раз отображение URL-ов вида `/posts/<ID>` к шаблону `postPage`:

```
Router.map(function() {
  this.route('postsList', {path: '/'});

  this.route('postPage', {
    path: '/posts/:_id'
  });
});
```

lib/router.js

Специальный синтаксис `:_id` сообщает маршрутизатору две вещи: 1) совпадает любой маршрут формы `/posts/xyz/`, где “xyz” может принимать любое значение; 2) положить всё что найдено на месте “xyz” внутрь свойства `_id` массива `params` маршрутизатора.

Обратите внимание, что мы используем `_id` здесь только ради удобства. Маршрутизатор не имеет возможности узнать, отправляем ли мы актуальный `_id` или просто некоторую случайную строку символов.

Теперь мы имеем маршрут к правильному шаблону, но нам чего-то не хватает. Маршрутизатор знает `_id` поста, который мы бы хотели показать, но шаблон не имеет о нём никакого представления.

К счастью, маршрутизатор имеет неплохое встроенное решение. Он позволяет указать **контекст данных (data context)** для шаблона. Вы можете думать о данных контекста как о начинке в торте, который сделан из шаблонов и макетов. Проще говоря, это то, чем вы заполняете шаблон:

The data context.

В нашем случае мы получим правильный контекст данных, отыскав наш пост по `_id`, который мы получили из URL:

```
Router.map(function() {
  this.route('postsList', {path: '/'});

  this.route('postPage', {
    path: '/posts/:_id',
    data: function() { return Posts.findOne(this.params._id); }
  });
});
```

lib/router.js

Итак, каждый раз, когда пользователь обращается по этому маршруту, мы находим соответствующий пост и отправляем его в шаблон. Напомним, что `findOne` возвращает единственный пост, совпавший с запросом, и ему можно передать только один аргумент `id` для сокращения записи `{_id: id}`.

Внутри функции `data` для маршрута, `this` соответствует текущему совпавшему маршруту, и мы можем использовать `this.params` для доступа к именованным частям маршрута (которые мы обозначили с помощью префиксов `:` внутри нашего `path`).

Передавая контекст данных шаблону, вы можете контролировать значение `this` внутри методов шаблона.

Это обычно делается неявно в итераторе `{{#each}}`, который автоматически устанавливает контекст данных каждой итерации к текущему элементу итерации:

```
{{#each widgets}}
  {{> widgetItem}}
{{/each}}
```

Но мы можем также сделать это явно, используя `{{#with}}`, который просто говорит: “возьми этот объект, и примени следующий шаблон к нему”. Например, мы можем написать:

```
{{#with myWidget}}
  {{> widgetPage}}
{{/with}}
```

Оказывается, вы можете добиться того же результата отправкой контекста в качестве *аргумента* в месте вызова шаблона. Таким образом, предыдущий блок кода может быть переписан так:

```
{{> widgetPage myWidget}}
```

Наконец, мы должны убедиться, что правильно указываем на индивидуальный пост. Снова мы

могли бы сделать как-то так `<a href="/posts/{{_id}}">`, но вместо этого используем метод шаблона для маршрутизации – он более надёжен.

Мы поименовали маршрут к посту как `postPage`, таким образом, мы можем использовать метод шаблона `{{pathFor 'postPage'}}`:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  </div>
</template>
```

client/views/posts/post_item.html

Routing to a single post page.

Открыть на GitHub | Запустить копию

Но подождите, как именно маршрутизатор узнает, где взять часть `xyz` в `/posts/xyz`? В конце концов, мы не передаем ему любой `_id`.

Оказывается, Iron Router достаточно умён, чтобы понять это самому. Мы говорим маршрутизатору использовать маршрут `postPage`, и маршрутизатор узнаёт, что этот маршрут требует некий параметр `_id` (ведь он прописан нами в определении маршрута `path`).

Таким образом, маршрутизатор ищет этот `_id` в наиболее логичном из возможных мест: в контексте данных метода `{{pathFor 'postPage'}}`, другими словами в `this`. Выходит наш `this` будет соответствовать посту, который (сюрприз!) действительно обладает свойством `_id`.

В качестве альтернативы вы также можете явно указать маршрутизатору, где бы вы хотели найти свойство `_id`, отправив второй аргумент методу шаблона (т.е. `{{pathFor 'postPage' someOtherPost}}`). На практике этот паттерн можно использовать, например, для получения предыдущих или следующих постов в списке.

Чтобы убедиться, что всё работает правильно, перейдите в браузере к списку постов и кликните по какой-нибудь ссылке ‘Discuss’. Вы должны увидеть что-то подобное этому:

A single post page.

Стоит понимать, что изменения адреса URL происходят с помощью технологии **HTML5 pushState**.

Маршрутизатор умеет отлавливать клики на локальных линках, одновременно не давая браузеру покинуть приложение. Вместо этого маршрутизатор просто вносит изменения в состояние приложения.

Если все работает корректно, страница должна измениться мгновенно. По факту, иногда всё меняется так быстро, что может понадобиться какая-нибудь страница перехода. Это выходит за рамки этой главы, но тем не менее эта тема интересна.

Meteor - это реактивный фреймворк. Если в вашем коде или в любых других данных появились какие-либо изменения, то они применятся немедленно, без необходимости что-либо перезагружать или обновлять.

Мы уже видели этот механизм в действии, когда наши шаблоны обновлялись сразу после изменения данных и route.

Мы постараемся подробнее разобрать и понять, как это работает в следующих главах, но сейчас, мы бы хотели подробнее остановиться на некоторых основных “реактивных” функциях, которые используются повсеместно.

На данный момент текущее состояние приложения Microscope полностью отражается в адресе URL (и в базе данных).

Но бывают случаи, когда необходимо сохранить какое-либо состояние, которое относится только к конкретному пользователю приложения (например, сворачивание и разворачивание списков, диалогов и т.д). Сессия будет удобным способом решения этой проблемы.

Сессия является глобальным и быстрым хранилищем данных. Под глобальным, подразумевается глобальный singleton объект: есть одна сессия, и она доступна отовсюду. Использование глобальных переменных считается дурным тоном, но сессия сама по себе рассматривается как глобальное хранилище данных и используется в различных частях приложения.

Сессия доступна отовсюду как `Session`. Чтобы установить какое-либо значение в сессии, необходимо сделать следующее:

```
❯ Session.set('pageTitle', 'A different title');
```

Browser console

Прочитать данные сессии можно, написав `Session.get('mySessionProperty');`. Это очень быстрое и удобное хранилище данных. Если вы напишете это в хелпер, то заметите, что вывод хелпера моментально меняется после изменения переменной сессии.

Чтобы проверить это, добавим следующий код в текст шаблона “layout”:

```
<header class="navbar">
  <div class="navbar-inner">
    <a class="brand" href="{{pathFor 'postsList'}}">{{pageTitle}}</a>
  </div>
</header>
```

client/views/application/layout.html

```
Template.layout.helpers({
  pageTitle: function() { return Session.get('pageTitle'); }
});
```

client/views/application/layout.js

Meteor автоматически обновится (как мы уже знаем, это называется “горячее обновление кода” или HCR) сохраняя переменные сессии, так что теперь мы увидим новый заголовок “A different title” в панели навигации. Если ничего не произошло, то просто попробуйте набрать предыдущую команду `Session.set()` снова.

Более того, если мы изменим значение еще раз (снова набрав `Session.set()` в консоли браузера), то мы увидим новый заголовок:

```
❯ Session.set('pageTitle', 'A brand new title');
```

Browser console

Сессия доступна глобально, так что изменения могут быть внесены в любом месте приложения. Это дает нам больше возможностей, но также может и загнать в ловушку.

> Если вы записываете в переменную сессии (`Session.set()`) одно и тоже значение повторно, Meteor'у хватит ума, чтобы не запускать реактивные обновления и избежать ненужных вызовов функций.

Мы уже встречали пример реактивного источника данных и видели его в действии внутри хелпера шаблона. Хоть некоторые места в Meteor (такие, как хелперы шаблонов) и являются реактивными, львиная доля всего приложения - все еще простой не реактивный JavaScript.

Давайте предположим, что где-нибудь в нашем приложении у нас есть следующий фрагмент кода:

```
helloWorld = function() {
  alert(Session.get('message'));
}
```

Не смотря на то, что мы обращаемся к переменной сессии, *контекст*, в котором к ней обращались, не реактивен: мы не будем видеть новые `alert`'ы каждый раз после изменения переменной.

Тут на помощь приходит **Autorun**. Как подразумевает название, код внутри блока `autorun` автоматически выполнится и будет выполнятся каждый раз, когда реактивный источник данных, использованный внутри, будет изменен.

Попробуйте выполнить в браузерной консоли следующее:

```
❯ Deps.autorun( function() { console.log('Value is: ' + Session.get('pageTitle')); } )
;
Value is: A brand new title
```

Browser console

Как вы могли ожидать, блок кода, переданный в `autorun`, сработал один раз и вывел в консоль значение переменной. Теперь давайте попробуем изменить заголовок:

```
❯ Session.set('pageTitle', 'Yet another value');
Value is: Yet another value
```

Browser console

Магия! Как только значение переменной сессии изменилось, `autorun` узнал об этом и выполнил весь свой код снова, выведя новое значение в консоль.

Итак, возвращаясь к предыдущему примеру, если мы хотим показывать новый alert при каждом изменении переменной сессии, нам нужно лишь обернуть наш код в блок `autorun`.

```
Deps.autorun(function() {
  alert(Session.get('message'));
});
```

Как мы только что видели, autorun'ы могут быть очень полезными для отслеживания реактивных источников данных и обязательного реагирования на них.

Пока мы разрабатывали наш Microscope, мы использовали одно из преимуществ Meteor, сберегающих время: горячее обновление кода (HCR). Всякий раз, когда мы сохраняем один из файлов исходников, Meteor замечает изменения и незаметно перезапускает работающий сервер

Meteor, оповещая каждого клиента о необходимости обновить страницу.

Это очень похоже на автоматическое обновление страниц, но есть одно важное отличие.

Чтобы понять его, начнем изменять переменную сессии, которую мы использовали:

```
❯ Session.set('pageTitle', 'A brand new title');
❯ Session.get('pageTitle');
'A brand new title'
```

Browser console

Если мы обновим страницу в браузере вручную, наши переменные сессии будут потеряны (точнее, они будут созданы заново). Но если мы спровоцируем выполнение горячего обновления кода (например, сохранив один из файлов исходников), страница будет обновлена, а переменные сессии останутся прежними. Попробуйте!

```
❯ Session.get('pageTitle');
'A brand new title'
```

Browser console

Итак, если мы используем переменные сессии для хранения того, что делает пользователь, HRC будет полностью прозрачной для него, так как сохранит значения всех переменных сессии. Это позволяет нам загружать новую production-версию нашего Meteor-приложения, будучи уверенными в минимальных нарушениях работы клиента наших пользователей.

Рассмотрим это чуть подробней. Если мы будем хранить все состояния в URL и в сессии, то мы сможем прозрачно менять *запущенный код* каждого клиента с минимальными потерями.

Давайте теперь проверим, что получится, если мы обновим страницу вручную:

```
❯ Session.get('pageTitle');
null
```

Browser console

Когда мы перезагрузили страницу, мы потеряли сессию. При HCR Meteor сохраняет сессию в local storage вашего браузера и загружает ее после обновления. Однако, при явной перезагрузке имеет смысл иное поведение: если пользователь обновляет страницу, как если бы он снова перешел по такому же URL, то данные должны быть сброшены к начальному состоянию, которое увидит любой пользователь, посетивший этот URL.

Самое важное в этой главе:

1. Всегда храните состояние пользователя в сессии или в URL, тогда HCR будет проходить с минимальными потерями.

2. Любое состояние, которое вы ходите разделить между пользователями, храните *внутри URL*.

У нас получилось создать и вывести статичные данные, а также связать все в простое приложение.

Мы даже увидели как наш UI отражает любое изменение данных, и как новые или измененные данные мгновенно появляются в UI. Не смотря на это наш сайт все еще хромает - мы не можем вводить данные. У нас даже полностью отсутствует такое понятие как пользователи!

Что же делать?

В большинстве сред веб разработки добавление пользовательских аккаунтов это нетривиальная задача. Почти каждое веб приложение предполагает наличие пользователей, но не смотря на это добавление подобной функциональности вовсе не так просто. Еще хуже - как только вам приходится иметь дело с OAuth или любым другим протоколом авторизации, вещи быстро выходят из под контроля.

К счастью, Meteor позаботился и об этом. Благодаря тому, что пакеты Meteor могут иметь один и тот же код на сервере (Javascript) и клиенте (Javascript, HTML и CSS), мы получаем систему пользовательских аккаунтов почти даром.

Мы могли бы использовать стандартный компонент Meteor для аккаунтов (добавив его с помощью `meteor add accounts-ui`), но так как мы начали пользовать Bootstrap в нашем приложении, мы подключим пакет `accounts-ui-bootstrap-dropdown`. Не волнуйтесь, единственная разница между этими двумя пакетами это стили внешнего вида. Откройте командную строку и вызовите:

```
$ meteor add mrt:accounts-ui-bootstrap-dropdown
$ meteor add accounts-password
```

Терминал

Эти две команды дают нам возможность использовать специальные шаблоны аккаунтов. Мы можем подключить их на нашем сайте с помощью хелпера `{{loginButtons}}`. Подсказка: вы можете контролировать с какой стороны появится выпадающее меню авторизации пользователя с помощью атрибута `align`. Например: `{{loginButtons align="right"}}`

Добавим кнопки в заголовок страницы. Так как содержимое заголовка растет, вынесем его в отдельный шаблон в папке `client/views/includes/`. Мы также добавим тегов и классов из Bootstrap, чтобы все выглядело красиво:

```
<template name="layout">
  <div class="container">
    {{>header}}
    <div id="main" class="row-fluid">
      {{>yield}}
    </div>
  </div>
</template>
```

client/views/application/layout.html

```
<template name="header">
  <header class="navbar">
    <div class="navbar-inner">
      <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </a>
      <a class="brand" href="{{pathFor 'postsList'}}">Microscope</a>
      <div class="nav-collapse collapse">
        <ul class="nav pull-right">
          <li>{{>loginButtons}}</li>
        </ul>
      </div>
    </div>
  </header>
</template>
```

client/views/includes/header.html

Теперь, если открыть наше приложение в браузере, можно заметить кнопки `Login` в верхнем правом углу сайта.

Пользовательские аккаунты Meteor

Все базовые действия пользователя теперь доступны - можно регистрироваться, авторизироваться, запрашивать смену пароля, и все остальное, что может понадобиться простому сайту с аккаунтами.

Можно указать системе аккаунтов, что пользователи должны авторизироваться при помощи логина, а не почтового адреса. Для этого потребуется добавить блок `Accounts.ui` в новый `config.js` файл в папке `client/helpers/`:

```
Accounts.ui.config({
  passwordSignupFields: 'USERNAME_ONLY'
});
```

client/helpers/config.js

Добавлены аккаунты и шаблон с кнопками

Открыть на GitHub | Запустить копию

Попробуйте зарегистрироваться с помощью кнопки `Sign up`. После этого надпись на кнопке сменится на ваше имя пользователя. Это означает, что вы успешно зарегистрировались и для вас была создана учетная запись. Но где сохранена эта запись?

Добавив пакет `accounts`, Meteor создал специальную коллекцию `Meteor.users`. Чтобы ее просмотреть, откройте консоль браузера и введите:

```
❯ Meteor.users.findOne();
```

Консоль браузера

Консоль должна вернуть объект, указывающий на объект вашего пользователя. Обратите внимание, там будет ваше имя пользователя, а также уникальный идентификатор `_id`. Вы можете также получить текущего пользователя с помощью команды `Meteor.user()`.

Теперь выйдите и зайдите как другой пользователь. `Meteor.user()` вернет второго пользователя. Попробуем посчитать, сколько у нас пользователей всего:

```
❯ Meteor.users.find().count();
1
```

Консоль браузера

Консоль вернет нам 1. Разве их не должно быть 2? Неужели первый пользователь был удален? Если вы попробуете залогиниться как первый пользователь, вы заметите что все на месте.

Давайте посмотрим, что происходит в главной базе данных Mongo. Запустите терминал Mongo командой `meteor mongo`, и посчитайте количество пользователей:

```
> db.users.count()
2
```

Консоль Mongo

Оба пользователя присутствуют. Тогда почему мы видим только одного пользователя в консоли браузера?

Если вы помните, в главе 4 мы отключили параметр `autopublish`, и тем самым коллекции перестали автоматически синхронизировать данные между сервером и локальными версиями коллекций. Чтобы образовался канал данных, нам надо было создать публикацию и подписку на нее.

Но мы не создавали ничего подобного для публикации новых пользователей. Каким же образом пользователям удается зарегистрироваться?

Оказывается пакет пользовательских аккаунтов содержит в себе опцию `auto-publish` - данные текущего пользователя передаются на сервер не смотря ни на что. Если бы этого не происходило, пользователь не смог бы авторизироваться на сайте.

Обратите внимание - пакет аккаунтов публикует только данные *текущего* пользователя. Это объясняет, почему пользователь не может увидеть данные других пользователей.

Данные пользователя публикуются только когда он авторизирован, и только одного пользователя на одну авторизацию. Когда ни один пользователь не залогинен, ничего не публикуется.

Вдобавок, документы нашего пользователя в коллекции содержат разные данные на сервере и клиенте. В Mongo у пользователя гораздо больше данных. Чтобы их просмотреть, вернитесь в терминал Mongo и запустите команду:

```
> db.users.findOne()
{
  "createdAt" : 1365649830922,
  "_id" : "kYdBd9hr3fWPGPcii",
  "services" : {
    "password" : {
      "srp" : {
        "identity" : "qyFCnw4MmRbmGyBdN",
        "salt" : "YcBjRa7ArXn5tdCdE",
        "verifier" : "df2c001edadf4e475e703fa8cd093abd4b63afccbca48fad1d2a0986ff2bcfba9
20d3f122d358c4af0c287f8eaf9690a2c7e376d701ab2fe1acd53a5bc3e843905d5dcaf2f1c47c25bf5dd87
764d1f58c8c01e4539872a9765d2b27c700dcdedadf5ac82521467356d3f91dbeaf9848158987c6d359c542
3e6b9cabf34fa0b45"
      }
    },
    "resume" : {
      "loginTokens" : [
        {
          "token" : "BMHipQqjfLoPz7gru",
          "when" : 1365649830922
        }
      ]
    }
  },
  "username" : "tmeasday"
}
```

Mongo консоль

Если же вы посмотрите на объект пользователя в браузере, вы заметите что он содержит гораздо меньше данных:

```
❯ Meteor.users.findOne();
Object {_id: "kYdBd9hr3fWPGPcii", username: "tmeasday"}
```

Консоль браузера

Этот пример наглядно демонстрирует как локальная коллекция может быть *безопасной версией* настоящей базы данных. Авторизированный пользователь видит ровно столько информации, сколько необходимо - в данном случае, для логина. Это очень полезный подход, как вы еще увидите в будущем.

Это вовсе не означает что вы не можете сделать больше пользовательских данных публичными. Стоит обратить внимание на **документацию Meteor** чтобы узнать, как опубликовать больше полей из коллекции `Meteor.users`.

Если коллекции - это центральный компонент Meteor, то **реактивность** - оболочка, благодаря которой этот компонент становится полезным.

Коллекции радикально преобразуют способ работы приложения с изменением данных. У нас больше нет необходимости проверять изменения вручную (например, через AJAX вызов), а затем применять их к HTML, - изменения данных сами могут поступать в любое время, и Meteor будет плавно применять их к пользовательскому интерфейсу.

Просто задумайтесь об этом ненадолго: за кулисами Meteor способен изменить *любую* часть вашего пользовательского интерфейса при обновлении лежащей в его основе коллекции.

Императивным способом сделать это будет использование `.observe()`, - функции курсора, которая вызывает коллбеки (callbacks - функции-обработчики события) при изменении соответствующих данному курсору документов. Далее с помощью этих коллбеков можно преобразовать DOM (отрисованный HTML нашей web-страницы). В итоге код будет выглядеть как-то так:

```
Posts.find().observe({
  added: function(post) {
    // при исполнении коллбека 'added' добавляем HTML элемент
    $('ul').append('<li id="' + post._id + '">' + post.title + '</li>');
  },
  changed: function(post) {
    // при исполнении коллбека 'changed' преобразуем текст HTML элемента
    $('ul li#' + post._id).text(post.title);
  },
  removed: function(post) {
    // при исполнении коллбека 'removed' удаляем HTML элемент
    $('ul li#' + post._id).remove();
  }
});
```

Вы, возможно, уже поняли, что такой код очень быстро станет запутанным. Представьте, что вам нужно будет иметь дело с изменениями для *каждого атрибута* поста и заменять сложный HTML внутри соответствующего `<li>` элемента. Не говоря уже обо всех замысловатых крайних случаях, которые могут возникнуть, когда мы начнем полагаться на разные источники информации, способные изменяться в реальном времени.

следует `observe()`

Использование описанного выше образца иногда необходимо, особенно в случае работы со сторонними виджетами. Например, давайте представим, что мы хотим в реальном времени добавлять или убирать отметки на карте в зависимости от данных в коллекции (скажем, чтобы отображать местоположение пользователей, находящихся в системе).

В таких случаях нужно будет использовать коллбеки `observe()`, чтобы карта "разговаривала" с Meteor коллекцией и знала, как реагировать на изменения данных. К примеру, вы будете полагаться на коллбеки `added` и `removed` для вызова собственных методов API карты `dropPin()` или `removePin()`.

Meteor предоставляет нам лучший способ - реактивность, которая, по существу, является примером применения декларативного подхода. Декларативность позволяет нам один раз определить отношение между объектами и знать, что они будут синхронизированы, вместо того, чтобы указывать поведение для каждого возможного изменения.

Это очень мощная концепция, потому что функционирующая в реальном времени система имеет много входных данных, способных изменяться в непредсказуемый момент. В Meteor декларативным образом определяется то, как нужно отрисовывать HTML в зависимости от состояния интересующих нас реактивных источников данных. Благодаря этому Meteor может позаботиться о мониторинге таких источников и взять на себя запутанную работу по поддержанию пользовательского интерфейса в актуальном состоянии.

А если покороче, то вместо того, чтобы думать о коллбеках `observe`, благодаря Meteor мы можем писать:

```
<template name="postsList">
  <ul>
    {{#each posts}}
      <li>{{title}}</li>
    {{/each}}
  </ul>
</template>
```

Затем получаем список постов, используя:

```
Template.postsList.helpers({
  posts: function() {
    return Posts.find();
  }
});
```

За кулисами Meteor настраивает коллбеки `observe()` за нас и перерисовывает соответствующие секции HTML при изменении реактивных данных.

Хотя Meteor является реактивным фреймворком, работающим в реальном времени, не *весь* код внутри приложения реактивный. Если бы это было так, то все приложение перезапускалось бы каждый раз, когда что-нибудь изменяется. Вместо этого реактивность ограничена отдельными областями кода, которые называют **вычислениями**(computations).

Другими словами, вычисление - это блок кода, который исполняется каждый раз, когда изменяется один из реактивных источников данных, от которых он зависит. Если у вас есть реактивный источник данных (например, переменная сессии) и вы хотите на него реактивно отзываться, вам нужно установить для него вычисление.

Заметьте, что обычно вам не нужно это делать напрямую, потому что Meteor сам добавляет для каждого отрисованного шаблона и хелпера их собственное вычисление (значит, вы можете быть уверены в том, что шаблоны будут реактивно отображать свои источники данных).

Каждый реактивный источник данных отслеживает все использующие его вычисления, чтобы сообщать им об изменении своего значения. Для этого он вызывает на вычислении функцию `invalidate()`.

Вычисления, как правило, установлены таким образом, чтобы просто пересматривать свое содержимое в случае инвалидации, - это как раз то, что происходит с вычислениями шаблонов (хотя они используют дополнительные трюки, чтобы перерисовывать страницу более эффективно). Несмотря на то, что вы можете при необходимости усилить контроль над тем, что именно ваши вычисления делают при инвалидации, на практике вы почти всегда будете использовать этот механизм.

Теперь когда мы понимаем теорию, которая стоит за вычислениями, было бы неплохо создать одно из них. Мы можем использовать функцию `Tracker.autorun`, чтобы заключить блок кода внутри вычисления и сделать его реактивным:

```
Meteor.startup(function() {
  Tracker.autorun(function() {
    console.log('There are ' + Posts.find().count() + ' posts');
  });
});
```

Обратите внимание, что нам нужно поместить блок `Tracker` внутри `Meteor.startup()`, чтобы убедиться, что он исполняется только после того, как Meteor закончил загружать коллекцию `Posts`.

За кулисами `autorun` затем создает вычисление и настраивает его, чтобы пересматривать содержимое при изменении источников данных, от которых он зависит. Мы установили простое вычисление, которое печатает в консоль количество постов. Так как `Posts.find()` - реактивный источник данных, он позаботится о том, чтобы сообщать вычислению о необходимости пересмотра содержимого каждый раз, когда количество постов изменяется.

```
> Posts.insert({title: 'New Post'});
There are 4 posts.
```

Итак, после прочтения этой главы мы можем писать код, который использует реактивные данные очень естественным образом, зная, что система зависимостей позаботится о его перезапуске в нужный момент.

Мы уже знаем как создавать новые посты через консоль командой `Posts.insert`. Но было бы жестоко заставлять наших пользователей открывать консоль и печатать команды обращения к базе данных.

Нам нужно создать дружелюбный пользовательский интерфейс для добавления новых постов.

Для начала создадим маршрут (route) к нашей новой странице:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
```

lib/router.js

Теперь, когда у нас есть маршрут новой страницы, давайте добавим на нее ссылку в заголовок:

```
<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" da
ta-target="#navigation">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="{{pathFor 'postsList'}}">Microscope</a>
      </div>
      <div class="collapse navbar-collapse" id="navigation">
        <ul class="nav navbar-nav">
          <li><a href="{{pathFor 'postSubmit'}}">Submit Post</a></li>
        </ul>
        <ul class="nav navbar-nav navbar-right">
          {{> loginButtons}}
        </ul>
      </div>
    </div>
  </nav>
</template>
```

client/templates/includes/header.html

Создание маршрута означает, что если пользователь решит открыть адрес `/submit` в браузере, Meteor отрендерит шаблон `postSubmit`. Давайте создадим этот шаблон:

```
<template name="postSubmit">
  <form class="main form">
    <div class="form-group">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
          <input name="url" id="url" type="text" value="" placeholder="Your URL" class
="form-control"/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
          <input name="title" id="title" type="text" value="" placeholder="Name your p
ost" class="form-control"/>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary"/>
  </form>
</template>
```

client/templates/posts/post_submit.html

Внимание: куча новой разметки. Но вся она родом из Twitter Bootstrap. Да, нам нужна только форма нового поста. Но дополнительные теги и классы сделают все гораздо симпатичнее. Теперь наша новая страница будет выглядеть примерно так:


Форма для нового поста

Нам не нужно волноваться насчет параметра `action` для этой формы, так как мы перехватим

событие и отправим данные с помощью JavaScript. Также не стоит волноваться насчет варианта когда JavaScript отключен в браузере - ведь тогда приложение Meteor просто не будет работать.

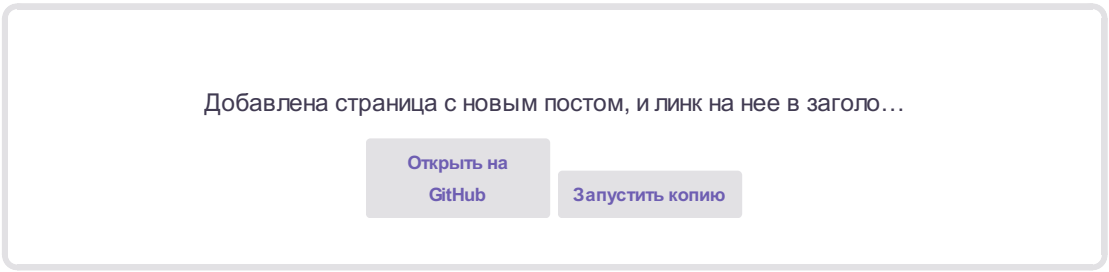
Давайте создадим обработчик событий для кнопки `Submit`. Проще всего использовать событие `submit` (чем, например, ловить событие `click` на кнопке), ведь тогда мы охватим все возможные сценарии событий (например, нажатие кнопки `Enter` в форме).

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    post._id = Posts.insert(post);
    Router.go('postPage', post);
  }
});
```

client/templates/posts/post_submit.js



Эта функция использует **jQuery** чтобы собрать данные со всех полей формы и создать объект нового поста. Мы также добавили вызов `preventDefault` для события `event`, чтобы браузер не попытался отправить форму традиционным способом.

Наконец, мы можем перенаправить пользователя на страницу с новым постом. Вызов функции `insert()` у коллекции вернет свежий `_id` объекта, который только что был добавлен в базу данных. Этот параметр мы добавим в вызов `Router.go()` - он будет добавлен в адресную строку.

Теперь пользователь может отправить новый пост кнопкой `Submit` - пост будет создан, а пользователь будет перенаправлен на страницу обсуждения этого поста.

Все это замечательно, но было бы неплохо ограничить доступ к созданию новых постов только для зарегистрированных пользователей. Мы могли бы спрятать форму на странице, но не смотря на это любой разбирающийся в браузерах человек мог бы открыть консоль и создать пост оттуда.

К счастью, защита данных зашита прямо в коллекции Meteor. Просто она по-умолчанию отключена

для новых проектов. Это позволяет легко начать новое приложение не тратя время на скучные вещи.

Нашему приложению больше не нужны эти костыли, поэтому настало время от них избавиться. Давайте удалим пакет `insecure`:

```
meteor remove insecure
```

Терминал

Возможно вы заметили что форма добавления постов сломалась. Это потому, что без пакета `insecure` редактирование коллекций на клиенте запрещено.

Нам нужно или специально сообщить Meteor'у что клиентам можно создавать новые посты, или делать вставку постов на сервере.

Для начала мы быстро починим отправку формы, разрешив создание постов на клиенте. Вы увидите позже что мы остановимся на немного другом подходе, но сейчас мы можем все починить следующим образом:

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  insert: function(userId, doc) {
    // разрешить постить только если пользователь залогинен
    return !! userId;
  }
});
```

lib/collections/posts.js

Удалили пакет insecure, разрешили определенные записи в п…

Открыть на GitHub | Запустить копию

Вызов `Posts.allow` сообщает Meteor'у что “в этих обстоятельствах клиентам разрешено модифицировать коллекцию `Posts`”. В данном случае мы говорим что “клиентам можно создавать новые посты до тех пор пока в них есть `userId`”.

Параметр `userId` пользователя, который создает пост, будет передан вызовам `allow` и `deny` (или функция вернет `null` если пользователь не залогинен), что очень полезно. Так как пользовательские аккаунты привязаны к ядру Meteor, мы можем рассчитывать на то что `userId` всегда верен.

Мы ограничили создание постов только для авторизированных пользователей. Попробуйте выйти из аккаунта и создать пост - вы скорее всего увидите следующее в консоли:

```
Insert failed: Access denied - Вставка провалилась: Отказано в доступе
```

Отлично. Осталась еще пара вещей:

- Неавторизованным пользователям все ещё доступна форма создания новых постов
- Пост никак не привязан к пользователю (и у нас нет кода на сервере для этого)
- Можно создать множество постов с одним и тем же URL.

Давайте это исправим.

Если пользователь не залогинен, ему не стоит показывать форму для новых постов. Верным местом для такого ограничения будет роутер. Для этого мы создадим *router hook*.

Hook умеет вмешиваться в процесс маршрутизации - когда мы перенаправляем пользователей согласно адресу на определенные функции нашего приложения. Hook похож на охранника проверяющего документы прежде чем пропустить дальше (или не пропустить).

Нам нужно проверить залогинен ли пользователь. Если нет - отрендерить шаблон `accessDenied` вместо привычного `postSubmit`, а также остановить маршрут и не дать ему больше ничего сделать. Давайте перепишем наш router.js:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

var requireLogin = function() {
  if (! Meteor.user()) {
    this.render('accessDenied');
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

lib/router.js

Теперь создадим шаблон для страницы “Доступ запрещен”:

```
<template name="accessDenied">
  <div class="access-denied jumbotron">
    <h2>Access Denied</h2>
    <p>You can't get here! Please log in.</p>
  </div>
</template>
```

client/templates/includes/access_denied.html

Доступ запрещен к странице с новыми постами если юзер не …

Открыть на GitHub | Запустить копию

Если вы попробуете открыть адрес http://localhost:3000/submit/ и при этом не будете залогинены, вы увидите нашу новую страницу:

Routing hooks хороши ещё тем, что они тоже *реактивны*. Это значит, что нам не нужно заботиться о навешивании обратных вызовов функций (callbacks) на авторизацию: если пользователь залогинится, шаблон страницы Роутера автоматически изменится с `accessDenied` на `postSubmit` - нам не нужно дополнительно писать код для этого (и между прочим, это сработает даже между разными вкладками браузера).

Авторизируйтесь и попробуйте обновить страницу. Возможно вы успеете заметить страницу “Доступ запрещен” на краткое мгновение перед тем, как появится форма нового поста. Это все потому, что Meteor начинает рендерить шаблоны как можно раньше, еще до того, как приложение успело побеседовать с сервером и спросить насчет существования текущего пользователя (который пока что сохранен в local storage браузера).

Чтобы избежать этой проблемы - а это распространенная проблема, с которой вы еще столкнетесь когда будете разбирать пикантные детали задержек передачи данных между клиентом и сервером - просто покажем экран загрузки на тот краткий момент, пока мы выясняем, если ли у текущего пользователя право создавать новые посты.

В данный момент мы еще не знаем, напечатал ли пользователь корректно свой логин и пароль. И мы не можем показать шаблон `accessDenied` или `postSubmit` до тех пор, пока это не выяснится.

Перепишем наш hook чтобы использовать шаблон загрузки страницы пока `Meteor.loggingIn()` возвращает true:

```
//...

var requireLogin = function() {
  if (! Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

lib/router.js

Показываем экран загрузки пока проверяется логин

Открыть на GitHub | Запустить копию

Самый простой способ предотвратить попытки пользователей зайти на страницу по ошибке - это просто спрятать линк когда они не залогинены. Легко:

```
//...

<ul class="nav navbar-nav">
  {{#if currentUser}}<li><a href="{{pathFor 'postSubmit'}}">Submit Post</a></li>{{/if}}
</ul>

//...
```

client/templates/includes/header.html

Показывать ссылку на создание поста только когда пользова…

Открыть на GitHub | Запустить копию

Хелпер `currentUser` доступен для нас через пакет `accounts` и является в шаблонах handlebars тем же самым, что и вызов `Meteor.user()`. Так как он реактивен, линк будет появляться и исчезать на странице когда статус логин пользователя будет меняться.

Мы закрыли доступ к странице с новыми постами для неавторизированных пользователей, а также запретили создание новых постов через консоль браузера. Осталось еще несколько моментов:

- Дата и время создания для каждого поста
- Добавить проверку уникальности URL в каждом посте. Один и тот же URL нельзя запостить дважды.
- Добавить детали автора поста (ID, имя пользователя, и все такое)

Первая мысль вероятно будет что мы можем все это воплотить в нашем обработчике события `submit`. На практике это вызвало бы массу проблем:

- Для даты и времени поста нам пришлось бы рассчитывать на дату-время компьютера пользователя, которая вполне может оказаться неверной.
- Браузер не сможет знать про *все* посты когда-либо отправленные на сайт. Только текущие посты будут сохранены в локальной базе данных браузера (чуть позже мы разберем как это работает). Так что мы никак не сможем проверить уникальность поля URL в новом посте.
- Наконец, даже если мы и *могли бы* добавить детали автора на клиенте, мы никак не смогли бы проверить их верность, что открыло бы дыру в безопасности для людей, использующих консоль браузера.

Из-за всех этих причин нам стоит делать обработчики событий простыми, а если мы собираемся совершать более продвинутые операции с добавлением и редактированием коллекций, стоит использовать **Метод**.

Метод Meteor - это функция на сервере, которую можно *вызвать* со стороны клиента. Пока что мы плохо с ними знакомы - хотя на самом деле, за кулисами, методы `insert`, `update`, `remove` наших коллекций все являются Методами. Давайте узнаем, как создать наш собственный Метод.

Вернемся к файлу `post_submit.js`. Вместо того чтобы добавлять новый пост напрямую в коллекцию `Posts`, мы вызовем Метод под названием `postInsert`:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // отобразить ошибку пользователю и прервать
      if (error)
        return alert(error.reason);

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

Функция `Meteor.call` вызовет Метод по имени своего первого аргумента. Вы можете добавить аргументы к этому вызову (в данном случае объект `post`, созданный из формы), и еще добавить callback-функцию, которая будет вызвана когда Метод на сервере закончит обработку.

Callback-функции в Методах должны обладать двумя аргументами: `error` и `result` (для ошибки и результата соответственно). Если, по какой-либо причине, в `error` было что-то передано, мы известим об этом пользователя (с использованием `return` для выхода из функции). Если же всё сработало как надо (в `error` не было ничего передано) мы перенаправим пользователя на страницу обсуждения вновь созданного поста.

Мы воспользуемся возможностью и добавим некоторую безопасность нашему методу с помощью пакета `audit-argument-checks`.

Этот пакет даёт возможность проверить любой JavaScript объект с помощью предустановленного паттерна. В нашем случае, мы используем его чтобы убедиться что пользователь, вызывающий Метод, залогинен (проверив что `Meteor.userId()` является `String`) и что объект `postAttributes`, который передаётся Методу как аргумент, содержит строки `title` и `url` (иначе пользователи смогли бы вставлять любые данные в нашу БД).

Теперь объявим Метод `postInsert` в файле `collections/posts.js`. Мы уберем блок `allow()` из `posts.js`, так как Методы Meteor игнорируют их в любом случае.

Перед тем, как вставить запись в базу данных и вернуть `_id` в виде объекта клиенту (иначе говоря, callback-функции на клиенте), мы дополним объект `postAttributes` тремя дополнительными полями: `_id` пользователя и его `username`, а также полем `submitted`, которое будет содержать временной код создания поста.

```
Posts = new Mongo.Collection('posts');

Meteor.methods({
  postInsert: function(postAttributes) {
    check(Meteor.userId(), String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});
```

collections/posts.js

Заметьте что `_.extend()` метод взят из библиотеки **Underscore**, и просто позволяет вам дополнить один объект свойствами другого.

Используем Метод для создания нового поста

Открыть на GitHub | Запустить копию

Обратите внимание что Методы выполняются на сервере, и Meteor предполагает что им можно доверять. Таким образом, Методы Meteor'а игнорируют любые allow/deny проверки.

Если вы хотите вызывать код перед любой операцией `insert`, `update`, или `remove` *даже на сервере*, мы предлагаем вам ознакомиться с пакетом **collection-hooks**.

Мы сделаем ещё одну проверку перед тем, как закончить с нашим Методом. Если пост с таким же URL уже был добавлен, мы вместо добавления дубликата просто перенаправим пользователя на этот ранее созданный пост.

```
Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});
```

collections/posts.js

Мы ищем в нашей базе данных любые посты, с таким же URL. Если таковой найден, то мы возвратим его `_id` вместе с флагом `postExists: true` чтобы уведомить клиента об исключительной ситуации.

А так как для возврата используется `return`, Метод на этом закончит выполнение и инструкция `insert` не будет вызвана, мы тем самым элегантно предотвращаем создание дубликатов.

Всё что осталось - это использовать новый флаг `postExists` в нашем хелпере на клиенте, чтобы отобразить предупреждение.

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

Enforce post URL uniqueness.

Открыть на GitHub Запустить копию

Теперь, когда у каждого поста есть время-дата, мы можем упорядочить все посты по этому атрибуту. Для этого воспользуемся оператором Mongo `sort`, который ожидает в качестве аргумента объект, состоящий из названий полей, по которым нужно сортировать, и индикаторов направления сортировки.

```
Template.postsList.helpers({
  posts: function() {
    return Posts.find({}, {sort: {submitted: -1}});
  }
});
```

client/templates/posts/posts_list.js

Сортируем объекты по времени создания

Открыть на GitHub Запустить копию

Всё это заняло у нас некоторое время - но теперь у нас есть полноценный интерфейс для создания контента!

Вдобавок к созданию контента пользователи захотят редактировать уже существующие посты, а также удалять их. Об этом и будет следующая глава.

В прошлой главе мы представили новый концепт из мира Meteor: **Методы**.

Без компенсации задержки передачи данных

Метод Meteor это способ организованно выполнить серию команд на сервере. В нашем примере мы использовали Метод чтобы добавить к новым постам имя автора, id автора, а также текущее время и дату на сервере.

Однако, если Meteor выполнял бы Методы самым примитивным способом, у нас появились бы проблемы. Представьте себе следующую цепочку событий (временные интервалы здесь просто случайные цифры в целях иллюстрации):

- *+0ms:* Пользователь жмет кнопку Submit и браузер вызывает Метод.
- *+200ms:* Сервер вносит изменения в базу данных Mongo.
- *+500ms:* Клиент получает измененные данные и обновляет интерфейс чтобы их отобразить

Если бы Meteor работал именно так, у нас была бы задержка между действием пользователя и отражением этого действия приложением (задержка будет более или менее ощутимая в зависимости как далеко вы находитесь от сервера). В современном веб приложении подобное запаздывание совершенно недопустимо!

С компенсацией задержки передачи данных

Чтобы избежать подобных проблем Meteor использует концепцию под названием **Компенсация Задержки** (**Latency Compensation**). Когда мы создали Метод `post`, мы добавили его в файл в папке `collections/`. Это означает что он доступен как серверу, так и *клиенту* - и оба могут запустить Метод одновременно!

Когда вы вызываете Метод, клиент посылает запрос на сервер. Но он также одновременно *симулирует* вызов Метода на коллекции клиента. Наша цепочка событий превращается в следующую:

- *+0ms:* Пользователь нажимают на кнопку Submit. Браузер вызывает Метод на сервере.
- *+0ms:* Клиент симулирует действие Метода на своих локальных коллекциях, и тут же отражает его результат в пользовательском интерфейсе.
- *+200ms:* Сервер вносит изменения в базу данных Mongo.
- *+500ms:* Клиент получает ответ от сервера с результатом операции. Клиент отменяет симулированные изменения и заменяет их настоящими, пришедшими с сервера (которые, как правило, совпадают с симулированными). Пользовательский интерфейс обновляется чтобы отразить изменения, если они есть.

В результате пользователь видит изменения мгновенно. Когда приложение получит ответ от сервера несколько мгновений спустя, интерфейс приложения может поменяться (а может и остаться прежним) чтобы отразить истинные изменения. Чтобы эти изменения оставались минимальными, нам нужно как можно лучше симулировать реальные документы.

Добавим небольшое изменение в Метод `post` чтобы понаблюдать за действиями клиента и сервера. Для этого мы напишем слегка продвинутый код с помощью npm пакета `futures` - он позволит нам замедлить создание объектов в Методе.

Мы воспользуемся свойством `isSimulation` чтобы узнать у Meteor если метод был вызван как `stub`. **Stub** это та самая симуляция Метода на клиенте, которую Meteor запускает параллельно с настоящим Методом на сервере.

Мы узнаем у Meteor выполняется ли код на клиенте. Если да - добавим строку `(client)` в конец заголовка нового поста. Если нет, добавим строку `(server)`:

```
Meteor.methods({
  post: function(postAttributes) {
    // […]

    // выбираем нужные поля для публикации
    var post = _.extend(_.pick(postAttributes, 'url', 'message'), {
      title: postAttributes.title + (this.isSimulation ? '(client)' : '(server)'),
      userId: user._id,
      author: user.username,
      submitted: new Date().getTime()
    });

    // ждем 5 секунд
    if (! this.isSimulation) {
      var Future = Npm.require('fibers/future');
      var future = new Future();
      Meteor.setTimeout(function() {
        future.return();
      }, 5 * 1000);
      future.wait();
    }

    var postId = Posts.insert(post);

    return postId;
  }
});
```

collections/posts.js

Внимание: если вы задумались что означает `this` в `this.isSimulation` - это так называемый **Method invocation object** (объект вызова Метода) который дает доступ к разным полезным переменным.

Подробный разбор пакета **Futures** выходит за рамки нашей книги. Но если вкратце - мы просто сообщили Meteor выждать 5 секунд перед тем как добавлять новый объект в коллекцию на сервере.

Мы также совершим перенаправление пользователя на страницу со списком постов:

```
Template.postSubmit.events({
  'submit form': function(event) {
    event.preventDefault();

    var post = {
      url: $(event.target).find('[name=url]').val(),
      title: $(event.target).find('[name=title]').val(),
      message: $(event.target).find('[name=message]').val()
    }

    Meteor.call('post', post, function(error, id) {
      if (error)
        return alert(error.reason);
    });
    Router.go('postsList');
  }
});
```

client/views/posts/post_submit.js

Демонстрируем порядок появления постов с помощью метода s…

Открыть на GitHub | Запустить копию

Если мы создадим пост, мы наглядно увидим компенсацию задержки в действии. Сначала пост появится со строкой `(client)` в заголовке (первый пост в списке, с ссылкой на GitHub):

Наш пост сохранен в коллекции клиента

Затем, 5 секунд спустя, он заменяется настоящим документом с сервера:

Наш пост после ответа сервера

После всего этого можно подумать, что Методы довольно сложны. На самом деле они могут быть очень просты. Мы уже увидели три очень простых Метода для редактирования коллекций - `insert`, `update` и `remove`.

Когда вы создаете новую коллекцию `posts`, вы также создаете три Метода: `posts/insert`, `posts/update` и `posts/delete`. Другими словами, когда вы вызываете `Posts.insert()` у коллекции клиента, вы вызываете Метод с компенсацией задержки, который делает две вещи:

1. Проверяет есть ли у него возможность редактировать коллекцию вызывая функции `allow` и `deny`.

2. Редактирует локальную коллекцию.

Если вы еще не потеряли нить повествования, вероятно вы только что заметили что наш Метод `post` вызывает другой Метод (`posts/insert`) когда мы создаем наш пост. Как это работает?

Когда запускается симуляция (версия Метода на клиенте), мы также запускаем симуляцию метода `insert` (таким образом добавляя новый объект в коллекцию на клиенте). Но мы *не вызываем* настоящий, серверный `insert` - мы ожидаем что *серверная* версия метода `post` сделает это.

Последовательно, когда серверная версия метода `post` вызывает `insert` , нам не нужно беспокоиться насчет симуляции, и объект успешно создается в главной базе данных.

Теперь, когда мы можем создавать посты, следующим шагом будет их редактирование и удаление. Хотя UI код для этого довольно прост, сейчас самое время будет поговорить о том, как Meteor работает с правами доступа.

Давайте взглянем на роутер. Мы добавим путь для страницы редактирования поста и зададим контекст:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.map(function() {
  this.route('postsList', {path: '/'});

  this.route('postPage', {
    path: '/posts/:_id',
    data: function() { return Posts.findOne(this.params._id); }
  });

  this.route('postEdit', {
    path: '/posts/:_id/edit',
    data: function() { return Posts.findOne(this.params._id); }
  });

  this.route('postSubmit', {
    path: '/submit'
  });
});

var requireLogin = function(pause) {
  if (! Meteor.user()) {
    if (Meteor.loggingIn())
      this.render('loading')
    else
      this.render('accessDenied');

    pause();
  }
}

Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

lib/router.js

Теперь можем сосредоточится на шаблоне. Наш `postEdit` шаблон приобретёт стандартный вид:

```
<template name="postEdit">
  <form class="main">
    <div class="control-group">
        <label class="control-label" for="url">URL</label>
        <div class="controls">
            <input name="url" type="text" value="{{url}}" placeholder="Your URL"/>
        </div>
    </div>

    <div class="control-group">
        <label class="control-label" for="title">Title</label>
        <div class="controls">
            <input name="title" type="text" value="{{title}}" placeholder="Name your p
ost"/>
        </div>
    </div>

    <div class="control-group">
        <div class="controls">
            <input type="submit" value="Submit" class="btn btn-primary submit"/>
        </div>
    </div>
    <hr/>
    <div class="control-group">
        <div class="controls">
            <a class="btn btn-danger delete" href="#">Delete post</a>
        </div>
    </div>
  </form>
</template>
```

client/views/posts/post_edit.html

А вот и `post_edit.js` менеджер для нашего шаблона:

```
Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    }

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        alert(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },

  'click .delete': function(e) {
    e.preventDefault();

    if (confirm("Delete this post?")) {
      var currentPostId = this._id;
      Posts.remove(currentPostId);
      Router.go('postsList');
    }
  }
});
```

client/views/posts/post_edit.js

К этому моменту большая часть этого кода должна быть вам уже знакома.

У нас есть две функции-обработчика событий - одна для события `submit` на форме, и одна для клика на линке `delete`.

Обработчик события `delete` предельно прост - перехватить событие клика, спросить подтверждение на удаление поста. Если пользователь согласен продолжить, получить ID текущего поста из контекста `this` нашего шаблона, и, наконец, перенаправить пользователя на главную страницу.

Обработчик события редактирования поста слегка длиннее, но не намного сложнее. Мы получаем ID текущего поста из контекста шаблона через `this._id`. Затем мы считываем обновленный пост из полей формы, создаем новый объект `postProperties` и сохраняем обновленный пост в нем.

Затем мы передаем этот объект методу Meteor'a `Collection.update()`. Вторым параметром вызова этого метода идет функция-коллбек, которую метод вызовет под конец обработки данных. В случае ошибки мы покажем диалог с причиной, а в случае успеха пользователь будет перенаправлен на страницу с обновленным постом.

Чтобы пользователи могли легко отредактировать пост, добавим соответствующие линки:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        submitted by {{author}}
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn">Discuss</a>
  </div>
</template>
```

client/views/posts/post_item.html

Имеет смысл показывать линк редактирования поста только в том случае, если этот пост принадлежит вам. Для этого воспользуемся хелпером `ownPost`:

```
Template.postItem.helpers({
  ownPost: function() {
    return this.userId == Meteor.userId();
  },
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  }
});
```

client/views/posts/post_item.js

Форма редактирования поста

Добавлена форма редактирования поста

Открыть на GitHub | Запустить копию

Форма для редактирования поста готова и выглядит отлично. Но вы все еще не сможете отредактировать пост. Что же происходит?

Так как мы недавно удалили пакет `insecure`, все модификации со стороны клиента отвергаются сервером.

Чтобы это починить, создадим правила доступа. Для начала создайте новый `permissions.js` файл в папке `lib`. Это позволит Meteor'у загрузить логику прав доступа с самого начала (и сделать ее доступной на клиенте и сервере):

```
// проверяем что userId на самом деле автор данного документа
ownsDocument = function(userId, doc) {
  return doc && doc.userId === userId;
}
```

lib/permissions.js

В главе **Создание постов** мы избавились от методов `allow()`, так как мы добавляли новые посты через Метод на сервере (который, в свою очередь, игнорирует `allow()`).

Но теперь, когда мы редактируем и удаляем посты на клиенте, давайте вернемся к файлу `posts.js` и добавим блок `allow()`:

```
Posts = new Meteor.Collection('posts');

Posts.allow({
  update: ownsDocument,
  remove: ownsDocument
});

Meteor.methods({
  ...
```

collections/posts.js

Добавлены простые права доступа с проверкой автора поста

Открыть на GitHub | Запустить копию

Только потому, что вам разрешено редактировать собственные посты еще не означает что вам можно редактировать *все* поля. Например, нам бы не хотелось дать пользователям возможность создать пост и затем сменить ID автора на кого-то другого.

Мы воспользуемся функцией `deny()` Meteor'a чтобы ограничить пользователей в редактировании полей:

```
Posts = new Meteor.Collection('posts');

Posts.allow({
  update: ownsDocument,
  remove: ownsDocument
});

Posts.deny({
  update: function(userId, post, fieldNames) {
    // разрешаем редактировать только следующие два поля:
    return (_.without(fieldNames, 'url', 'title').length > 0);
  }
});
```

collections/posts.js

Разрешаем редактировать только определенные поля постов.

Открыть на GitHub | Запустить копию

Мы берем массив `fieldNames` со списком редактируемых полей и используем метод библиотеки **Underscore** `without()` чтобы пропустить через него наш массив и вернуть новый, уже *без* полей `url` или `title`.

Если все в порядке, новый массив будет пуст и его длина должна быть 0. Если же кто-то пытается отредактировать недопустимые поля, длина этого массива будет 1 или больше, и функция вернет `true` (и редактирование поста будет отвергнуто).

Для создания постов мы используем Метод `post`. Для редактирования и удаления постов мы вызываем `update` и `remove` прямо на клиенте, ограничивая доступ через `allow` и `deny`.

Когда же стоит использовать каждую из этих техник?

Когда вещи довольно просты и вы можете адекватно выразить правила редактирования данных через `allow` и `deny`, обычно все проще делать через клиента.

Прямой доступ к базе данных через клиента создает ощущение непосредственности, и может лучше порадовать пользователя отзывчивым интерфейсом, до тех пор пока вы помните правильно реагировать на ошибки сервера (то есть, когда сервер ответит что редактирование базы данных не удалось).

Однако как только вам понадобится совершать действия выходящие за рамки контроля пользователя (например, добавление времени создания поста или его авторства), стоит использовать Метод.

Вызов Метода также лучше подходит в следующих случаях:

- Когда вам нужно узнать или вернуть данные через функцию-коллбек вместо того, чтобы ждать пока сработает реактивность и синхронизация данных.
- Для массивных функций обработки базы данных
- Когда требуется просуммировать или агрегировать данные (например, вызов методов count, average, sum для массивов данных).

Система защиты данных Meteor позволяет контролировать редактирование базы данных без необходимости создавать каждый раз отдельный Метод.

Так как нам нужно было несколько вспомогательных задач для добавления дополнительных полей к посту, а также специальных действий когда URL поста уже был опубликован ранее, использование специального Метода `post` имело смысл.

С другой стороны, нам не нужно было создавать Методы для редактирования и удаления постов. Достаточно было проверить, разрешено ли пользователю совершать эти действия - и это было очень просто сделать с помощью проверок `allow` и `deny`.

С помощью этих коллбеков мы можем явно объявить, какие операции редактирования базы данных разрешены. Дополнительным бонусом является тот факт, что они неплохо интегрируются с системой пользовательских аккаунтов.

Мы можем объявить столько коллбеков `allow`, сколько нам понадобится. Чтобы операция прошла проверку достаточно если *один* из них вернет `true`. Когда `Posts.insert` вызывается браузером (из консоли браузера, или со страницы нашего приложения), сервер в свою очередь будет вызывать все доступные проверки операции `insert` до тех пор, пока одна из них не вернет `true`. Если ни одной из проверок нет, или все они возвращают `false`, сервер вернет ошибочный статус `403`.

Точно так же мы можем создать одну или несколько проверок `deny`. Если хотя бы одна из них вернет `true`, изменения будут отменены и статус `403` будет сгенерирован сервером. Для успешной операции `insert` в базе данных будут вызваны один или несколько `allow` `insert` проверок, а также *все* проверки `deny` `insert`.

Внимание: n/e означает Не Выполнено (Not Executed)

Другими словами, Meteor двигается по листу проверок сверху вниз. Сначала он проходит все проверки `deny`, затем `allow`, и вызывает все функции-коллбеки до тех пор, пока одна из них не вернет `true`.

Приведем наглядный пример. Две проверки `allow()` - первая проверяет соответствие авторства поста текущему пользователю, а вторая выясняет, есть ли у пользователя права администратора. Если текущий пользователь админ, одна из этих проверок гарантировано вернет `true`, и таким образом мы разрешаем ему редактирование всех постов.

Если вы помните, все Методы отвечающие за изменение базы данных (такие как `.update()`) содержат компенсацию задержки передачи данных (равно как и любые другие Методы). Если вы,

скажем, попробуете удалить пост который вам не принадлежит через консоль браузера, вы увидите как пост на мгновение исчезнет, когда локальная коллекция удалит этот пост. Чуть позже он появится назад, когда сервер сообщит что нет, документ не был удален.

Разумеется, подобное поведение пользовательского интерфейса не проблема, когда пользователи играются с консолью - до тех пор пока все изменения ограничиваются *их* браузером. С другой стороны, вам стоит отразить доступные действия в интерфейсе приложения. Например, надо потратить какое-то время на код, чтобы кнопка удаления поста не появлялась с документами, которые пользователю не разрешено удалять.

К счастью, вы можете создать общие правила доступа для клиента и сервера. Например, вы можете написать библиотечную функцию `canDeletePost(user, post)` и сохранить ее в общей папке `/lib`.

Обратите внимание что система ограничения доступа распространяется только на операции с базой данных, пришедшие со стороны клиента. На сервере Meteor предполагает что *все* операции разрешены.

Это означает, что если вы создадите Метод Meteor под названием `deletePost` и сделаете его доступным для вызова на клиенте, любой пользователь сможет удалить любой пост. Скорее всего вы этого не хотите, за исключением если внутри Метода не проверяются права пользователя на удаление постов.

Наконец, еще один способ использовать `deny` - воспользоваться им в качестве коллбека `onX`. Например, вы могли бы добавлять дату и время `lastModified` к постам следующим образом:

```
Posts.deny({
  update: function(userId, doc, fields, modifier) {
    doc.lastModified = +(new Date());
    return false;
  },
  transform: null
});
```

Так как проверки `deny` вызываются для каждой операции `update`, мы знаем что эта операция будет гарантированно вызвана, и соответствующие изменения будут внесены в документ перед его сохранением в базу данных.

Эта техника все же отдает запахом хака, поэтому возможно вы захотите обновлять содержимое документов в соответствующем Методе. Не смотря на это, нам будет полезно знать о таком подходе. Возможно, в будущем появится коллбек в стиле `beforeUpdate` для официального выполнения кода перед тем как документ сохранен.

Использование стандартного диалога `alert()` для уведомления пользователя об ошибках и проблемах вряд ли оставит о нашем приложении хорошее впечатление. Мы можем сделать все гораздо лучше.

Давайте создадим универсальный механизм оповещения об ошибках, который будет уведомлять пользователей не слишком отвлекая их от приложения.

Мы создадим простую систему, которая будет следить за тем, какие ошибки пользователь уже успел просмотреть, а также показывать новые ошибки в специально отведенной области на сайте под названием “flash”.

Наша система будет похожа на сообщения ошибок в Ruby on Rails, но только более утонченную - она будет находиться на клиенте и знать, когда пользователь уже просмотрел сообщение.

Для начала, мы создадим коллекцию для хранения ошибок. Учитывая то, что ошибки относятся только к текущей сессии и их вовсе не нужно сохранять на будущее, мы сделаем что-то новенькое. Мы создадим *локальную коллекцию*. Это означает, что коллекция `Errors` будет существовать только в браузере и не будет даже пытаться синхронизироваться с сервером.

Для этого мы создадим объект ошибки в файле, доступном только на клиенте, с именем коллекции `null`. Мы создадим функцию `throwError` которая будет добавлять новую ошибку в нашу локальную коллекцию:

```
// Локальная коллекция, доступна только на клиенте
Errors = new Meteor.Collection(null);
```

client/helpers/errors.js

Теперь, когда коллекция создана, мы можем добавить функцию `throwError`, которую мы будем вызывать для добавления новых ошибок. Нам не нужно заботиться об обработчиках `allow` или `deny`, так как это локальная коллекция, которая не будет сохранена в базу данных Mongo.

```
throwError = function(message) {
  Errors.insert({message: message})
}
```

client/helpers/errors.js

Преимущество локальной коллекции для сохранения ошибок в том, что как и все коллекции она реактивна. Это позволит нам оперативно отражать ошибки в пользовательском интерфейсе, точно также как и любую другую коллекцию.

Ошибки будут выводиться вверху нашего главного шаблона:

```
<template name="layout">
  <div class="container">
    {{> header}}
    {{> errors}}
    <div id="main" class="row-fluid">
      {{yield}}
    </div>
  </div>
</template>
```

client/views/application/layout.html

Давайте создадим шаблоны `errors` и `error` в файле `errors.html`:

```
<template name="errors">
  <div class="errors row-fluid">
    {{#each errors}}
      {{> error}}
    {{/each}}
  </div>
</template>

<template name="error">
  <div class="alert alert-error">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    {{message}}
  </div>
</template>
```

client/views/includes/errors.html

> Вы наверное обратили внимание, что мы поместили два шаблона в один файл. До последнего момента мы придерживались правила “по одному шаблону в один файл”, но для Meteor все будет работать даже если мы поместим все шаблоны в один файл (хотя при этом наш `main.html` стал бы плохо читаем и огромен в размерах).
>
> В нашем случае оба шаблона довольно кратки. Мы сделаем исключение и поместим оба шаблона в один файл чтобы не засорять наш репозиторий обилием файлов.

Нам осталось только добавить логику поиска ошибок в нашем методе шаблона (helper):

```
Template.errors.helpers({
  errors: function() {
    return Errors.find();
  }
});
```

client/views/includes/errors.js

Простой вывод ошибок.

Открыть на GitHub | Запустить копию

Мы знаем как выводить ошибки, но нам все еще нужно создать некоторое количество ошибок чтобы что-то заработало. Обычно ошибки появляются в процессе ввода пользователями новой информации. Сейчас мы добавим проверку в функции-обработчике события создания нового поста, и выведем сообщение для каждой ошибки.

Вдобавок, если у нас будет ошибка `302` (означающая что пост с данным URL уже существует), мы перенаправим пользователя на страницу с существующим постом. Мы получим `_id` этого поста из `error.details` (если вы помните из главы 7, мы передаем `_id` этого поста как третий аргумент `details` нашего класса `Error`).

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val(),
      message: $(e.target).find('[name=message]').val()
    }

    Meteor.call('post', post, function(error, id) {
      if (error) {
        // показываем ошибку пользователю
        throwError(error.reason);

        if (error.error === 302)
          Router.go('postPage', {_id: error.details})
      } else {
        Router.go('postPage', {_id: id});
      }
    });
  }
});
```

client/views/posts/post_submit.js

На самом деле используем систему вывода ошибок.

Открыть на GitHub | Запустить копию

Попробуйте создать пост и ввести URL `http://meteor.com`. Так как этот URL уже добавлен к посту из тестовых данных, вы скорее всего увидите:

Преднамеренно вызываем ошибку

Если вы уже попробовали клинуть на кнопке закрытия ошибки, вы заметили что ошибка исчезает. Но если вы попробуете открыть другую страницу, ошибка вновь появится. Что же происходит?

Кнопка закрытия ошибки вызывает скрипт Twitter Bootstrap, и не имеет ничего общего с Meteor! Происходит следующее - Bootstrap удаляет `<div>` с сообщением об ошибке из DOM, но не из коллекции Meteor. Само собой, ошибка тут же появится вновь, как только Meteor отрисует страницу заново.

Если мы не хотим, чтобы ошибки восставали из мертвых, напоминая пользователям об их прошлых проступках и сводя их с ума, нам стоит добавить механизм удаления ошибок из коллекции.

Для начала мы изменим функцию `throwError`, добавив в нее свойство `seen`. Это нам пригодится чуть позже для ведения учета, видел ли пользователь уже эту ошибку.

Затем мы создадим простую фукцию `clearErrors` очищающую коллекцию от уже отображенных ошибок:

```
// Local (client-only) collection
Errors = new Meteor.Collection(null);

throwError = function(message) {
  Errors.insert({message: message, seen: false})
}

clearErrors = function() {
  Errors.remove({seen: true});
}
```

client/helpers/errors.js

Дальше, мы добавим очистку от ошибок в роутер, чтобы при открытии новой страницы старые ошибки удалялись автоматически:

```
// ...

Router.before(requireLogin, {only: 'postSubmit'})
Router.before(function() { clearErrors() });
```

lib/router.js

Чтобы наша функция `clearErrors()` заработала, ошибки должны помечаться параметром `seen`.

Для этого стоит учесть один особенный случай. Когда мы выдаем ошибку и перенаправляем пользователя на другую страницу (как в случае с уже существующей ссылкой), перенаправление происходит мгновенно. У пользователя не остается времени увидеть ошибку перед тем, как она будет убрана с экрана.

Вот тут мы и воспользуемся параметром `seen`. Нам нужно установить его значение в `true` если пользователь успел увидеть ошибку.

Для этого мы воспользуемся `Meteor.defer()`. Эта функция сообщает Meteor выполнить коллбек сразу же после того, что происходит сейчас. Вы можете представить себе вызов `defer()` как способ сообщить браузеру подождать 1 миллисекунду, перед тем как двигаться дальше.

Мы просим Meteor установить параметр `seen` в `true` одной миллисекундой позже после того, как шаблон `errors` отрисован. Помните, как мы говорили что перенаправление на другую страницу осуществляется мгновенно? Это означает, перенаправление произойдет до того, как будет вызван обработчик `defer`.

Это то что нам нужно - если обработчик не будет вызван, ошибка не будет отмечена как `seen`, и она не будет удалена из коллекции - соответственно, мы увидим ее на странице, куда был перенаправлен пользователь:

```
Template.errors.helpers({
  errors: function() {
    return Errors.find();
  }
});

Template.error.rendered = function() {
  var error = this.data;
  Meteor.defer(function() {
    Errors.update(error._id, {$set: {seen: true}});
  });
};
```

client/views/includes/errors.js

Следим за тем какие ошибки уже были показаны. Очищаем оши…

Открыть на GitHub | Запустить копию

Функция `rendered` будет вызвана, как только наш шаблон будет отрисован в браузере. Внутри этой функции `this` указывает на текущий объект шаблона, а `this.data` позволит нам обратиться к параметрам объекта, который мы отрисовываем (в данном случае, ошибке).

Фух! Куча работы ради чего-то, что пользователи возможно никогда не увидят!

`rendered`

Функция `rendered` у шаблона вызывается каждый раз, когда шаблон отрисован браузером. Это включает в себя первый раз, когда шаблон появляется на экране. он также будет вызван каждый раз, когда шаблон будет отрисован заново - то есть каждый раз, когда любые из его данных изменятся.

Эти обработчики будут вызываться как минимум дважды - когда приложение загрузится, и когда данные из коллекций будут загружены в шаблон. По этой причине стоит быть осторожным с кодом, который не должен быть вызван дважды (например, диалоги с сообщениями или код отслеживания посещений).

Мы создали неплохой механизм для обработок ошибок. Теперь настало время поделиться им с сообществом Meteor.

Для начала, нам нужно удостовериться, что у нас есть Meteor Developer account. Вы можете зарегистрировать такой на сайте meteor.com. Вам нужно определить имя вашей учетной записи, так как в этой главе оно будет сильно задействовано.

Мы будем использовать имя `tmeasday` для примеров, вы можете заменить его на свое.

Во-первых нам нужно создать структуру пакета где он будет находиться. Для этого мы можем использовать команду `meteor create --package tmeasday:errors`. Заметьте, что Meteor создал папку `packages/tmeasday:errors` с файлами внутри. Мы начнем редактировать файл `package.js`, этот файл уведомляет Meteor о том, как пакет должен быть использован и какие объекты и функции он экспортирует.

```
Package.describe({
  summary: "Механизм отображения ошибок приложения пользователю",
  version: "1.0.0"
});

Package.onUse(function (api, where) {
api.versionsFrom('METEOR@0.9.0');

  api.use(['minimongo', 'mongo-livedata', 'templating'], 'client');

  api.add_files(['errors.js', 'errors_list.html', 'errors_list.js'], 'client');

  if (api.export)
    api.export('Errors');
});
```

packages/tmeasday:errors/package.js

Разрабатывая пакет для реального использования, будет неплохо добавить Git URL репозитория вашего пакета поле `git` в `Package.describe` (если вы публикуете свои пакеты на git сервере). Таким образом пользователи могут изучать исходный код, и если вы используете GitHub, README файл репозитория будет использован в описании на Atmosphere.

Добавим еще три файла в наш пакет (сгенерированный Meteor'ом код можно убрать). Мы можем скопировать их из кода Microscope практически без изменений, за исключением правильного названия переменных и слегка почищенного API:

```
Errors = {
  // Локальная (только для клиента) коллекция
  collection: new Meteor.Collection(null),

  throw: function(message) {
    Errors.collection.insert({message: message, seen: false})
  },
  clearSeen: function() {
    Errors.collection.remove({seen: true});
  }
};
```

packages/tmeasday:errors/errors.js

```
<template name="meteorErrors">
  {{#each errors}}
    {{> meteorError}}
  {{/each}}
</template>

<template name="meteorError">
  <div class="alert alert-error">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    {{message}}
  </div>
</template>
```

packages/tmeasday:errors/errors_list.html

```
Template.meteorErrors.helpers({
  errors: function() {
    return Errors.collection.find();
  }
});

Template.meteorError.rendered = function() {
  var error = this.data;
  Meteor.defer(function() {
    Errors.collection.update(error._id, {$set: {seen: true}});
  });
};
```

packages/tmeasday:errors/errors_list.js

Протестируем изменения локально с помощью Microscope, чтобы убедиться что приложение все еще корректно работает. Для добавления пакета в наш проект мы запустим команду `meteor add tmeasday:errors`. Затем нам нужно удалить существующие файлы, которые больше не нужны:

```
$ rm client/helpers/errors.js
$ rm client/views/includes/errors.html
$ rm client/views/includes/errors.js
```

удаляем старые файлы с помощью консоли bash

Еще нам нужно внести незначительные изменения в код чтобы он использовал правильный API:

```
Router.before(function() { Errors.clearSeen(); });
```

lib/router.js

```
  {{> header}}
  {{> meteorErrors}}
```

client/views/application/layout.html

```
Meteor.call('post', post, function(error, id) {
  if (error) {
    // display the error to the user
    Errors.throw(error.reason);
```

client/views/posts/post_submit.js

```
Posts.update(currentPostId, {$set: postProperties}, function(error) {
  if (error) {
    // display the error to the user
    Errors.throw(error.reason);
```

client/views/posts/post_edit.js

Пакет для обработки ошибок создан и добавлен в приложение.

Открыть на GitHub | Запустить копию

После всех упомянутых изменений наше приложение должно начать работать точно так же, как и до них.

Первым шагом в разработке пакета было его тестирование вместе с приложением. Следующим шагом будет написание тестов, которые как следует протестируют поведение пакета. Meteor упакован встроенным тестером под названием Tinytest, позволяющим легко запускать серии тестов, и дающим спокойствие душе и разуму после того, как вы опубликовали ваш пакет для всех.

Создадим файл с тестами, использующими Tinytest. Они будут тестировать уже существующий код обработки ошибок:

```
Tinytest.add("Errors collection works", function(test) {
  test.equal(Errors.collection.find({}).count(), 0);

  Errors.throw('A new error!');
  test.equal(Errors.collection.find({}).count(), 1);

  Errors.collection.remove({});
});

Tinytest.addAsync("Errors template works", function(test, done) {
  Errors.throw('A new error!');
  test.equal(Errors.collection.find({seen: false}).count(), 1);

  // отрисовываем шаблон
  OnscreenDiv(Spark.render(function() {
    return Template.meteorErrors();
  }));

  // ждем несколько миллисекунд
  Meteor.setTimeout(function() {
    test.equal(Errors.collection.find({seen: false}).count(), 0);
    test.equal(Errors.collection.find({}).count(), 1);
    Errors.clearSeen();

    test.equal(Errors.collection.find({seen: true}).count(), 0);
    done();
  }, 500);
});
```

packages/tmeasday:errors/errors_tests.js

В этих тестах мы проверяем работоспособность функций из `Meteor.Errors`, а также делаем двойную проверку что отрисованный (`rendered`) код в шаблоне все еще работает.

Мы не будем углубляться в детали написания тестов для пакетов Meteor (так как API все еще не заморожен и сильно меняется), но мы надеемся что все довольно просто и понятно.

Чтобы сообщить Meteor, как запускать тесты в `package.js`, используйте следующий код:

```
Package.onTest(function(api) {
  api.use('tmeasday:errors', 'client');
  api.use(['tinytest', 'test-helpers'], 'client');

  api.add_files('errors_tests.js', 'client');
});
```

packages/errors/package.js

Добавлены тесты в пакет.

Открыть на GitHub | Запустить копию

Теперь мы можем запустить серию тестов следующей командой:

```
$ meteor test-packages tmeasday:errors
```

Terminal

Все тесты пройдены успешно

Мы хотим опубликовать наш пакет и сделать его доступным для всех. Для этого отправим пакет на пакетный сервер Meteor'а и таким образом опубликуем его на Atmosphere.

Сделать это очень легко. Нам нужно перейти в папку с пакетом и отправить пакет командой `meteor publish --create`:

```
$ cd packages/tmeasday:errors
$ meteor publish --create
```

Терминал

Теперь, когда пакет опубликован, мы можем смело удалить его из проекта и добавить в приложение:

```
$ rm -r packages/tmeasday:errors
$ meteor add tmeasday:errors
```

Терминал (запускаем из корневой папки нашего приложения)

Пакет удален из приложения.

Открыть на GitHub | Запустить копию

Теперь мы можем увидеть как Meteor впервые загружает наш пакет из репозитория. Отличная работа!

Целью каждого социального сайта является создание сообщества пользователей. Это было бы невозможно, если у людей не было бы способа общаться друг с другом. В этой главе мы добавим в наше приложение возможность оставлять комментарии к постам.

Для начала создадим новую коллекцию для хранения комментариев. Также сразу добавим в нее тестовые комментарии (`fixtures`).

```
Comments = new Meteor.Collection('comments');
```

collections/comments.js

```
// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // create two users
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachaId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sacha = Meteor.users.findOne(sachaId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sacha._id,
    author: sacha.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: now - 7 * 3600 * 1000
  });

  Comments.insert({
    postId: telescopeId,
    userId: tom._id,
    author: tom.profile.name,
    submitted: now - 5 * 3600 * 1000,
    body: 'Interesting project Sacha, can I get involved?'
  });

  Comments.insert({
    postId: telescopeId,
    userId: sacha._id,
    author: sacha.profile.name,
    submitted: now - 3 * 3600 * 1000,
    body: 'You sure can Tom!'
  });

  Posts.insert({
    title: 'Meteor',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://meteor.com',
    submitted: now - 10 * 3600 * 1000
  });

  Posts.insert({
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteorbook.com',
    submitted: now - 12 * 3600 * 1000
  });
}
```

server/fixtures.js

Добавим механизм публикации и подписки на нашу коллекцию:

```
Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function() {
  return Comments.find();
});
```

server/publications.js

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() {
    return [Meteor.subscribe('posts'), Meteor.subscribe('comments')];
  }
});
```

lib/router.js

Добавлена коллекция комментариев, публикация и подписка н…

Открыть на GitHub | Запустить копию

Обратите внимание - чтобы тестовые комментарии оказались в базе данных вам нужно вызвать команду `meteor reset`. Эта команда сбросит все записи в Mongo и перезапишет в нее все тестовые данные (`fixtures`). Учетные записи пользователей также будут стерты, так что придется зарегистрироваться заново.

При создании тестовых записей произошли следующие вещи. Сначала мы создали двух тестовых пользователей - `Tom Coleman` и `Sacha Greif`. Затем мы использовали параметр `id` чтобы найти их в базе данных. От имени каждого пользователя был создан комментарий под первым постом, в комментарии был добавлен `id` поста (объект комментария содержит параметр `postId`) и `id` автора (параметр `userId`). Мы также добавили дату создания комментария, текст самого комментария и *денормализованное* поле `author`.

Также мы подсказали нашему маршрутизатору ожидать загрузки постов и комментариев.

Наши комментарии уже добавляются и читаются из базы данных. Настало время показать их в приложении. К счастью, этот процесс должен быть уже знаком, и вы скорее всего уже представляете следующие шаги:

```
<template name="postPage">
  {{> postItem}}

  <ul class="comments">
    {{#each comments}}
      {{> comment}}
    {{/each}}
  </ul>
</template>
```

client/views/posts/post_page.html

```
Template.postPage.helpers({
  comments: function() {
    return Comments.find({postId: this._id});
  }
});
```

client/views/posts/post_page.js

Мы добавили блок `{{#each comments}}` в шаблон поста, так чтобы `this` указывал на пост в хелпере `comments`. Чтобы найти соответствующие посту комментарии, мы проверяем их по атрибуту текущего поста `postId`.

Учитывая все, что мы узнали о хелперах и шаблонах `handlebars`, отрисовывание комментария происходит привычным способом. Добавим новую папку `comments` внутри папки `views` для всех шаблонов, связанных с комментариями:

```
<template name="comment">
  <li>
    <h4>
      <span class="author">{{author}}</span>
      <span class="date">on {{submittedText}}</span>
    </h4>
    <p>{{body}}</p>
  </li>
</template>
```

client/views/comments/comment.html

Создадим небольшой шаблон-помощник для форматирования даты создания комментария в читаемый вид (если, конечно, вы не из тех редких людей кто чувствует себя комфортно с форматами дат UNIX и шестнадцатиричными кодами цветов).

```
Template.comment.helpers({
  submittedText: function() {
    return new Date(this.submitted).toString();
  }
});
```

client/views/comments/comment.js

Затем мы покажем цифру с количеством комментариев для каждого поста:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        submitted by {{author}},
        <a href="{{pathFor 'postPage'}}">{{commentsCount}} comments</a>
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn">Discuss</a>
  </div>
</template>
```

client/views/posts/post_item.html

И добавим хелпер `commentsCount` для нашего менеджера `postItem`:

```
Template.postItem.helpers({
  ownPost: function() {
    return this.userId == Meteor.userId();
  },
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  },
  commentsCount: function() {
    return Comments.find({postId: this._id}).count();
  }
});
```

client/views/posts/post_item.js

Показываем комментарии на странице с постом

Открыть на GitHub | Запустить копию

Теперь наши тестовые комментарии должны появиться на странице.

Показываем комментарии

Давайте дадим возможность пользователям отправлять новые комментарии. Процесс будет примерно таким же, как и когда мы добавляли возможность создания новых постов.

Для начала мы добавим форму нового комментария под каждым постом:

```
<template name="postPage">
  {{> postItem}}

  <ul class="comments">
    {{#each comments}}
      {{> comment}}
    {{/each}}
  </ul>

  {{#if currentUser}}
    {{> commentSubmit}}
  {{else}}
    <p>Please log in to leave a comment.</p>
  {{/if}}
</template>
```

client/views/posts/post_page.html

Затем мы создадим шаблон с формой нового комментария:

```
<template name="commentSubmit">
  <form name="comment" class="comment-form">
    <div class="control-group">
        <div class="controls">
            <label for="body">Comment on this post</label>
            <textarea name="body"></textarea>
        </div>
    </div>
    <div class="control-group">
        <div class="controls">
            <button type="submit" class="btn">Add Comment</button>
        </div>
    </div>
  </form>
</template>
```

client/views/comments/comment_submit.html

Форма для создания нового комментария

Для отправки комментариев мы вызовем Метод `comment` в менеджере `commentSubmit`, который работает почти также, как и менеджер `postSubmit`:

```
Template.commentSubmit.events({
  'submit form': function(e, template) {
    e.preventDefault();

    var $body = $(e.target).find('[name=body]');
    var comment = {
      body: $body.val(),
      postId: template.data._id
    };

    Meteor.call('comment', comment, function(error, commentId) {
      if (error){
        throwError(error.reason);
      } else {
        $body.val('');
      }
    });
  }
});
```

client/views/comments/comment_submit.js

Точно так же, как ранее мы создали Метод Meteor `post` на сервере, мы создадим еще один Метод под названием `comment` для создания новых комментариев, проверки комментариев на соответствие заданным нами правилам, и наконец сохранения комментариев в коллекцию базы данных.

```
Comments = new Meteor.Collection('comments');

Meteor.methods({
  comment: function(commentAttributes) {
    var user = Meteor.user();
    var post = Posts.findOne(commentAttributes.postId);
    // ensure the user is logged in
    if (!user)
      throw new Meteor.Error(401, "You need to login to make comments");

    if (!commentAttributes.body)
      throw new Meteor.Error(422, 'Please write some content');

    if (!post)
      throw new Meteor.Error(422, 'You must comment on a post');

    comment = _.extend(_.pick(commentAttributes, 'postId', 'body'), {
      userId: user._id,
      author: user.username,
      submitted: new Date().getTime()
    });

    return Comments.insert(comment);
  }
});
```

collections/comments.js

Форма для отправки комментариев готова.

Открыть на GitHub | Запустить копию

Ничего особенного - мы просто проверяем что пользователь залогинен, что у комментария есть текст, и что он относится к конкретному посту.

На настоящий момент мы публикуем все комментарии для всех постов на все подключенные клиенты. Наверное, это чересчур. В конце концов, на клиенте нам требуется только небольшая часть этих данных. Давайте улучшим наш механизм публикаций и подписки - мы будем контролировать, что именно отправляется клиентам.

Если подумать, нам нужна подписка на комментарии только когда пользователь открыл страницу конкретного поста - и даже тогда нам нужны комментарии только для этого поста.

Для начала мы изменим способ подписки на комментарии. До последнего момента мы подписывались на уровне *маршрутизатора*. Это означало что все данные загружались как только маршрутизатор был готов к работе.

Но теперь мы хотели бы, чтобы наша подписка зависела от параметра в адресе страницы - и этот параметр менялся бы в зависимости от страницы. Поэтому мы переместим код подписки с уровня *маршрутизатора* на уровень *маршрута*.

Теперь, вместо загрузки всех данных одновременно с загрузкой приложения мы будем загружать данные когда приложение будет запрашивать определенный адрес. Это негативно отразится на времени загрузки страниц во время работы приложения, но тут мало что можно сделать - если только загрузить все данные на клиент и хранить их там постоянно.

Так будет выглядеть новая функция `waitOn` на уровне маршрута:

```
Router.map(function() {

  //...

  this.route('postPage', {
    path: '/posts/:_id',
    waitOn: function() {
      return Meteor.subscribe('comments', this.params._id);
    },
    data: function() { return Posts.findOne(this.params._id); }
  });

  //...

});
```

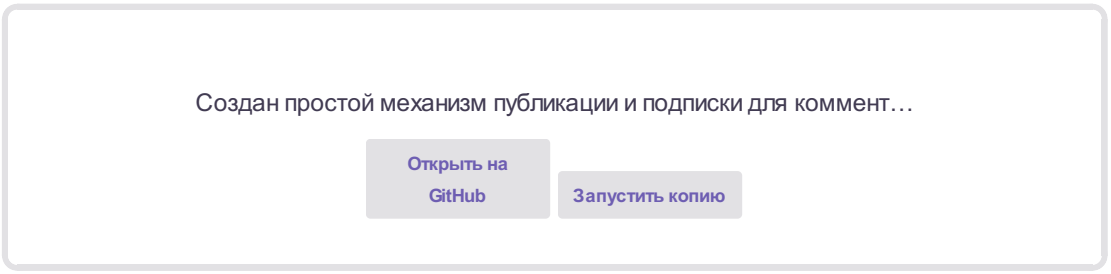
lib/router.js

Вы заметите что мы передаем `this.params._id` как аргумент для подписки. Используем этот параметр, чтобы ограничить публикуемые комментарии только теми, что относятся к данному посту:

```
Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function(postId) {
  return Comments.find({postId: postId});
});
```

server/publications.js



Осталась только одна загвоздка - если мы вернемся на стартовую страницу, у всех постов будет высвечено 0 комментариев.

У вас нет комментариев!

Причина такого поведения проста - мы всегда загружаем комментарии только *одного* поста. Когда же мы вызываем `Comments.find({postId: this._id})` в хелпере `commentsCount` в менеджере `post_item`, Meteor не может найти необходимые данные на клиенте чтобы правильно посчитать количество комментариев.

Лучший решением будет *денормализовать* количество комментариев прямо в пост (если вы не уверены что это означает, не волнуйтесь - следующая вспомогательная глава все прояснит). Несмотря на то, что наш код стал немного сложнее, это отлично компенсируется скоростью работы приложения когда не нужно публиковать *все* комментарии все время.

Добавим свойство `commentsCount` в структуру объекта `post`. Также, обновим структуру тестовых постов (и вызовем `meteor reset` чтобы их перезагрузить. Не забудьте заново зарегистрировать пользователя):

```
var telescopeId = Posts.insert({
  title: 'Introducing Telescope',
  ..
  commentsCount: 2
});

Posts.insert({
  title: 'Meteor',
  ...
  commentsCount: 0
});

Posts.insert({
  title: 'The Meteor Book',
  ...
  commentsCount: 0
});
```

server/fixtures.js

Затем мы удостоверимся, что у всех новых постов 0 комментариев:

```
// выбираем нужные поля
var post = _.extend(_.pick(postAttributes, 'url', 'title', 'message'), {
  userId: user._id,
  author: user.username,
  submitted: new Date().getTime(),
  commentsCount: 0
});

var postId = Posts.insert(post);
```

collections/posts.js

Дальше мы просто обновляем значение свойства `commentsCount` когда создается новый комментарий. У Mongo как раз есть подходящий оператор `$inc` для инкрементного увеличения значения параметра:

```
// update the post with the number of comments
Posts.update(comment.postId, {$inc: {commentsCount: 1}});

return Comments.insert(comment);
```

collections/comments.js

И еще теперь мы можем удалить хелпер `commentsCount` из `client/views/posts/post_item.js`, так как это поле теперь доступно прямо в объекте поста.

Денормализуем количество комментариев в объект поста.

Открыть на GitHub | Запустить копию

Наши пользователи теперь могут общаться друг с другом. Было бы обидно, если бы они пропустили новые комментарии. Следующая глава расскажет о том, как мы сможем уведомить пользователей о новых комментариях.

Денормализация данных означает что данные не сохраняются в “обычном” формате. Другими словами, денормализация означает множество копий одних и тех же данных, но в разных местах.

В предыдущей главе мы денормализовали количество комментариев в объект с постом, чтобы не запрашивать все комментарии с сервера каждый раз. С точки зрения архитектуры данных, мы могли бы этого избежать, ведь в любой момент можно посчитать правильное количество комментариев, запросив их из базы данных. Это было бы медленнее, но точнее.

Денормализование часто означает дополнительную работу для программиста. В нашем случае, каждый раз, когда мы добавляем или удаляем комментарий, нам также надо обновить соответствующие посты, чтобы их поле `commentsCount` отражало верное количество комментариев. По этой самой причине реляционные базы данных вроде MySQL презрительно фыркают в сторону такого подхода.

С другой стороны, у реляционного подхода есть и свои минусы. Без параметра `commentsCount` нам пришлось бы запросить *все* комментарии с сервера каждый раз, когда нам потребовалось бы посчитать их количество - то же самое, что творилось у нас с самого начала. Денормализование данных помогает полностью избежать этого.

> На самом деле *возможно* создать особенную публикацию, которая будет отсылать только значение с количеством комментариев - именно то, в чем мы заинтересованы.
>
> С другой стороны, всегда стоит взвешивать сложность создания подобной публикации против сложностей подхода с денормализацией данных.

Конечно, подобные рассуждения всегда должны опираться на задачи разрабатываемого приложения. Если вы разрабатываете код, в котором точность данных ставится превыше всего, возможно стоит пожертвовать скоростью работы приложения в пользу этой самой точности.

Если вы уже работали с Mongo, вероятно, вы удивились, когда мы создали отдельную коллекцию для комментариев. Вместо этого мы вполне могли бы внедрить лист комментариев прямо в документ с постом.

Оказывается, большинство встроенных инструментов Meteor работают гораздо лучше, если они оперируют на уровне коллекции. Например:

1. Функция-помощник `{{each}}` гораздо эффективнее, когда она работает с курсором (результатом запроса функции `collection.find()`). Та же функция гораздо менее эффективна, когда нужно просматривать массив объектов, внедренных в большой документ.

2. `allow` и `deny` работают на уровне документа. По этой причине фильтрование комментариев легко воплотить, когда они находятся в отдельной коллекции - и гораздо сложнее, если они внедрены в другую коллекцию.
3. DDP работает на уровне атрибутов первого уровня у документов. Это означает, если у `comments` есть свойство `post`, то каждый раз, когда новый комментарий добавлен к посту, сервер будет посылать обновленный лист со всеми комментариями на все подключенные клиенты.
4. Контролировать публикации и подписки гораздо проще на уровне документов. Например, если бы мы хотели разбить длинный список комментариев на несколько страниц, это оказалось бы сложнее, если бы комментарии не были сохранены в своей отдельной коллекции.

Mongo предлагает внедрять документы, чтобы уменьшить количество запросов к базе данных. С другой стороны, если мы вспомним про архитектуру Meteor - большая часть запросов происходит на *клиенте*, где запросы к MiniMongo происходят мгновенно.

> Есть хороший аргумент, согласно которому вам *не стоит* денормализовать данные. Мы рекомендуем прочитать следующий блогпост **Почему вам никогда не стоит использовать MongoDB** от Sarah Mei.

Теперь, когда пользователи могут оставлять комментарии к постам друг друга, было бы здорово сообщать им о начале диалога.

Для этого мы будем уведомлять автора поста о том, что был добавлен новый комментарий, и предоставлять ссылку, по которой комментарий можно посмотреть.

Это как раз тот тип функциональности, при котором Meteor демонстрирует свои сильные стороны: благодаря real-time природе Meteor такие уведомления будут отображаться *мгновенно*. Нам не нужно ждать, пока пользователь обновит страницу, или что-либо проверять - мы можем просто добавлять новые уведомления без необходимости писать для этого какой-нибудь специальный код.

Мы будем создавать уведомление каждый раз, когда кто-то оставляет комментарий к вашему посту. В будущем уведомления могут быть также использованы и при многих других сценариях, но сейчас этого будет достаточно, чтобы держать пользователя в курсе происходящего.

Создадим коллекцию `Notifications`, а также функцию `createCommentNotification`, которая будет добавлять соответствующее уведомление для каждого нового комментария к вашим постам.

Так как мы будем обновлять уведомления с клиентской стороны, нужно удостовериться в надежности вызова ‘allow’. Мы проверим это следующим образом:

- Пользователь, вызывающий `update`, является владельцем обновляемого уведомления.
- Пользователь пытается обновить только одно поле.
- Этим единственным полем является свойство `read`.

```
Notifications = new Mongo.Collection('notifications');

Notifications.allow({
  update: function(userId, doc, fieldNames) {
    return ownsDocument(userId, doc) &&
      fieldNames.length === 1 && fieldNames[0] === 'read';
  }
});

createCommentNotification = function(comment) {
  var post = Posts.findOne(comment.postId);
  if (comment.userId !== post.userId) {
    Notifications.insert({
      userId: post.userId,
      postId: post._id,
      commentId: comment._id,
      commenterName: comment.author,
      read: false
    });
  }
};
```

lib/collections/notifications.js

Подобно постам и комментариям, коллекция `Notifications` будет доступна и на клиенте, и на сервере. Так как нам нужна будет возможность обновлять уведомления после того, как пользователь их увидит, мы разрешаем выполнение операции `update`, убедившись, как обычно, что пользователь может редактировать только свои собственные данные.

Мы также написали несложную функцию, которая во время добавления комментария проверяет пост, определяет, кому о нем нужно сообщить, и создает новые уведомления.

У нас уже есть серверный метод для создания комментариев, поэтому остается всего лишь добавить в него вызов нашей функции. Мы заменяем `return Comments.insert(comment);` на `comment._id = Comments.insert(comment)`, чтобы сохранить `_id` нового комментария в переменной, а затем вызываем функцию `createCommentNotification`:

```
Comments = new Mongo.Collection('comments');

Meteor.methods({
  comment: function(commentAttributes) {

    //...

    comment = _.extend(commentAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    // обновляем количество комментариев для поста
    Posts.update(comment.postId, {$inc: {commentsCount: 1}});

    // создаем комментарий и сохраняем id
    comment._id = Comments.insert(comment);

    // создаем уведомление, информируя пользователя о новом комментарии
    createCommentNotification(comment);

    return comment._id;
  }
});
```

lib/collections/comments.js

Затем публикуем уведомления:

```
Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function(postId) {
  check(postId, String);
  return Comments.find({postId: postId});
});

Meteor.publish('notifications', function() {
  return Notifications.find();
});
```

server/publications.js

И подписываемся на клиенте:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('posts'), Meteor.subscribe('notifications')]
  }
});
```

lib/router.js

Added basic notifications collection.

Открыть на GitHub | Запустить копию

Теперь мы можем добавить список уведомлений к заголовку.

```
<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" da
ta-target="#navigation">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="{{pathFor 'postsList'}}">Microscope</a>
      </div>
      <div class="collapse navbar-collapse" id="navigation">
        <ul class="nav navbar-nav">
          {{#if currentUser}}
            <li>
              <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
            </li>
            <li class="dropdown">
              {{> notifications}}
            </li>
          {{/if}}
        </ul>
        <ul class="nav navbar-nav navbar-right">
          {{> loginButtons}}
        </ul>
      </div>
    </div>
  </nav>
</template>
```

client/templates/includes/header.html

Создадим шаблоны `notifications` и `notificationItem` (они будут делить между собой файл `notifications.html`):

```
<template name="notifications">
  <a href="#" class="dropdown-toggle" data-toggle="dropdown">
    Notifications
    {{#if notificationCount}}
      <span class="badge badge-inverse">{{notificationCount}}</span>
    {{/if}}
    <b class="caret"></b>
  </a>
  <ul class="notification dropdown-menu">
    {{#if notificationCount}}
      {{#each notifications}}
        {{> notificationItem}}
      {{/each}}
    {{else}}
      <li><span>No Notifications</span></li>
    {{/if}}
  </ul>
</template>

<template name="notificationItem">
  <li>
    <a href="{{notificationPostPath}}">
      <strong>{{commenterName}}</strong> commented on your post
    </a>
  </li>
</template>
```

client/templates/notifications/notifications.html

Как мы видим, план состоит в том, чтобы каждое уведомление содержало ссылку на пост с соответствующим комментарием и имя пользователя, который оставил этот комментарий.

Далее нам нужно убедиться, что мы выбираем правильный список уведомлений в хелпере, а также помечаем уведомления как “read” (“прочитанные”), когда пользователь проходит по указанной в них ссылке.

```
Template.notifications.helpers({
  notifications: function() {
    return Notifications.find({userId: Meteor.userId(), read: false});
  },
  notificationCount: function(){
    return Notifications.find({userId: Meteor.userId(), read: false}).count();
  }
});

Template.notification.helpers({
  notificationPostPath: function() {
    return Router.routes.postPage.path({_id: this.postId});
  }
});

Template.notification.events({
  'click a': function() {
    Notifications.update(this._id, {$set: {read: true}});
  }
});
```

client/templates/notifications/notifications.js

Display notifications in the header.

Открыть на GitHub | Запустить копию

Вы могли бы подумать, что уведомления не так уж сильно отличаются от ошибок: они действительно очень схожи по структуре. Тем не менее у них есть одно ключевое различие: в случае с уведомлениями мы создали настоящую синхронизованную между клиентом и сервером коллекцию. Это значит, что наши уведомления *постоянны* и при использовании одной и той же учетной записи будут оставаться неизменными при обновлении браузера и для разных устройств.

Попробуйте открыть второй браузер (к примеру, Firefox), создать новую учетную запись и добавить комментарий к посту, который вы создали, используя свою основную учетную запись (ту, что осталась открытой в Chrome). Вы должны видеть что-то вроде этого:

Displaying notifications.

Итак, с функционированием наших уведомлений все в порядке. Но есть одна небольшая проблема: они находятся в публичном доступе.

Если ваш второй браузер все еще открыт, попробуйте выполнить в его консоли следующий код:

```
❯ Notifications.find().count();
1
```

Browser console

Новый пользователь (тот, что *добавил комментарий*) не должен иметь никаких уведомлений. То уведомление, которое они оба могут видеть в коллекции `Notifications`, на самом деле принадлежит нашему *первоначальному* пользователю.

Даже если не учитывать потенциальные проблемы с конфиденциальностью, мы просто не можем себе позволить загружать полную коллекцию уведомлений в браузер каждого пользователя. Для относительно крупного сайта это может привести к перегрузке памяти, доступной браузеру, и серьезным проблемам с производительностью.

Эта проблема может быть решена при помощи **публикаций**. Мы можем использовать их, чтобы уточнить, какую часть нашей коллекции мы хотим отправить каждому браузеру.

С этой целью мы должны возвращать из публикации курсор, отличный от `Notifications.find()`. А конкретнее, нам нужен курсор, соответствующий уведомлениям только для текущего пользователя.

Сделать это достаточно просто, учитывая, что функция `publish` имеет доступ к `_id` текущего пользователя через `this.userId`:

```
Meteor.publish('notifications', function() {
  return Notifications.find({userId: this.userId, read: false});
});
```

server/publications.js

Only sync notifications that are relevant to the user.

Открыть на GitHub | Запустить копию

Если мы снова проверим два наших браузера, мы должны увидеть две разных коллекции с уведомлениями:

```
❯ Notifications.find().count();
1
```

Browser console (user 1)

```
❯ Notifications.find().count();
0
```

Browser console (user 2)

На самом деле, список уведомлений должен изменяться даже тогда, когда вы входите в систему или выходите из нее. Это происходит из-за того, что публикации автоматически обновляются при смене учетной записи.

Наше приложение становится все более функциональным, и в силу того, что много новых пользователей регистрируются и начинают постить ссылки, возникает риск бесконечной домашней страницы. Мы решим эту проблему в следующей главе, добавив нумерацию страниц.

Ситуации, в которых необходимо самостоятельно писать код для отслеживания зависимостей, встречаются редко. Однако такой код, без сомнения, полезно понимать, чтобы следить за процессом разрешения зависимостей.

Представьте, что мы хотим выяснить, сколько друзей в Facebook текущего пользователя лайкнули каждый пост на Microscope. Допустим, что мы уже проработали все детали аутентификации пользователя через Facebook, добавили необходимые вызовы API и получили интересующие нас данные. Теперь у нас на клиенте есть асинхронная функция, которая возвращает количество лайков, - `getFacebookLikeCount(user, url, callback)`.

Важно помнить, что данная функция является *не реактивной* и не выполняется в реальном времени. Она будет посылать HTTP запрос к Facebook, получать данные и делать их доступными приложению через асинхронный коллбек. Однако функция не будет самостоятельно перезапускаться, когда изменится количество лайков на Facebook, а интерфейс не станет реагировать на изменения данных, лежащих в его основе.

Чтобы это исправить, мы можем начать с использования `setInterval` и вызывать нашу функцию каждые несколько секунд:

```
currentLikeCount = 0;
Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId).url,
      function(err, count) {
        if (!err)
          currentLikeCount = count;
      });
  }
}, 5 * 1000);
```

Каждый раз, когда мы проверяем переменную `currentLikeCount`, мы ожидаем получить правильное число для конкретного момента с ошибкой в пределах пяти секунд. Теперь мы можем использовать данную переменную в хелпере следующим образом:

```
Template.postItem.likeCount = function() {
  return currentLikeCount;
}
```

Тем не менее, ничто пока не вызывает повторную отрисовку шаблона в случае изменения `currentLikeCount`. Хотя переменная теперь имитирует режим реального времени в том смысле, что она изменяется сама по себе, она не является *реактивной* и по-прежнему не может правильно взаимодействовать с остальной экосистемой Meteor.

Реактивность Meteor опосредована *зависимостями (dependencies)* - структурами данных,

которые отслеживают набор вычислений.

Как мы уже видели в главе про реактивность, вычисление - это часть кода, которая использует реактивные данные. В нашем случае существует вычисление, которое было создано исключительно для шаблона `postItem`, и у каждого хелпера в менеджере этого шаблона тоже есть свое вычисление.

Вы можете воспринимать вычисление как участок кода, который “заботится” о реактивных данных. В случае изменений в данных именно вычисление будет об этом проинформировано (при помощи `invalidate()`), и оно же будет решать, нужно ли предпринимать какие-либо действия.

Чтобы превратить нашу переменную `currentLikeCount` в реактивный источник данных, нам нужно отслеживать все вычисления, которые ее используют в рамках зависимости. Это предполагает ее превращение из переменной в функцию, возвращающую значение:

```
var _currentLikeCount = 0;
var _currentLikeCountListeners = new Tracker.Dependency();

currentLikeCount = function() {
  _currentLikeCountListeners.depend();
  return _currentLikeCount;
}

Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId),
      function(err, count) {
        if (!err && count !== _currentLikeCount) {
          _currentLikeCount = count;
          _currentLikeCountListeners.changed();
        }
      });
  }
}, 5 * 1000);
```

Мы только что установили зависимость `_currentLikeCountListeners`, которая следит за всеми вычислениями, использующими `currentLikeCount()`. Когда значение `currentLikeCount()` изменяется, мы вызываем функцию `changed()` для этой зависимости, что инвалидирует все наблюдаемые вычисления.

Затем вычисления могут работать с изменением, рассматривая различные случаи по отдельности.

Если вам кажется, что это большое количество шаблонного кода для простого реактивного источника данных, то вы правы, и в Meteor есть встроенные инструменты, позволяющие упростить процесс (точно так же как вместо того, чтобы использовать вычисления напрямую, вы обычно просто применяете autorun). Существует базовый пакет `reactive-var`, который делает абсолютно то же самое, что и функция `currentLikeCount()`. Если мы добавим его:

```
meteor add reactive-var
```

Мы можем использовать его, чтобы немного упростить код:

```
var currentLikeCount = new ReactiveVar();

Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId),
      function(err, count) {
        if (!err) {
          currentLikeCount.set(count);
        }
      });
  }
}, 5 * 1000);
```

Теперь мы будем вызывать `currentLikeCount.get()` из хелпера, и все будет работать как прежде. Кроме этого существует базовый пакет `reactive-dict`, предоставляющий реактивное хранилище для пар “ключ-значение” (почти как `Session`), который тоже может быть полезным.

Angular - это работающая только на клиентской стороне библиотека для реактивной отрисовки, разработанная ребятами из Google. Очень пояснительным является сравнение подходов Meteor и Angular к отслеживанию зависимостей, так как их различия существенны.

Мы знаем, что модель Meteor использует блоки кода, называемые вычислениями. Эти вычисления отслеживаются специальными “реактивными” источниками данных (функциями), которые заботятся об их инвалидации в случае необходимости. Таким образом, источник данных *явно* информирует все свои зависимости, когда им нужно вызвать `invalidate()`. Имейте в виду, что, хотя это обычно происходит в случае изменения данных, потенциально инвалидация также может быть запущена источником данных по иным причинам.

Кроме того, хотя вычисления обычно перезапускаются только в результате инвалидации, вы можете настроить их по своему желанию. Все это дает нам высокий уровень контроля над реактивностью.

В Angular реактивность опосредована объектом `scope (область видимости)`. Область видимости может рассматриваться как простой объект JavaScript с парой специальных методов.

Когда вы хотите установить реактивную зависимость от значения в области видимости, вы вызываете `scope.$watch`, передавая выражение, которое вас интересует (к примеру, какие части области видимости нужно отслеживать), а также функцию-слушатель (англ. listener function), которая будет выполняться каждый раз, когда выражение будет изменяться. Таким образом, мы явно устанавливаем, что конкретно мы хотим делать в случае изменения значения выражения.

Возвращаясь назад к нашему примеру с Facebook, мы бы написали:

```
$rootScope.$watch('currentLikeCount', function(likeCount) {
  console.log('Current like count is ' + likeCount);
});
```

Конечно, так же как в Meteor вы редко определяете вычисления, в Angular вы не так уж часто явно вызываете `$watch`, потому что `{{expressions}}` и директивы `ng-model` автоматически устанавливают наблюдатели (англ. watchers), которые потом заботятся о перерисовке в случае изменений.

Когда подобное реактивное значение изменяется, должна быть вызвана функция `scope.$apply()`. Она заново оценивает каждый наблюдатель области видимости, но вызывает функции-слушатели только для тех, значения вычислений которых *изменились*.

Таким образом, `scope.$apply()` похожа на `dependency.changed()` за исключением того, что она действует на уровне области видимости, а не предоставляет вам право указать, какие точно функции-слушатели должны быть вычислены заново. Этот незначительный дефицит контроля позволяет Angular самому очень разумно и эффективно определять такие функции.

С Angular наша функция `getFacebookLikeCount()` выглядела бы примерно так:

```
Meteor.setInterval(function() {
  getFacebookLikeCount(Meteor.user(), Posts.find(postId),
    function(err, count) {
      if (!err) {
        $rootScope.currentLikeCount = count;
        $rootScope.$apply();
      }
    });
}, 5 * 1000);
```

Стоит признать, что Meteor берет на себя большую часть тяжелой работы и позволяет использовать реактивность без особых усилий с нашей стороны. Однако мы надеемся, что изучение этих моделей будет полезным, если когда-нибудь вам придется работать над чем-то более сложным.

Наше приложение Microscope продвигается ударными темпами, и оно определенно станет хитом когда мы его запустим.

По этой причине стоит задуматься о быстродействии приложения, и о том как поток запросов повлияет на производительность, когда сотни и тысячи пользователей ринутся создавать новые посты.

Ранее мы говорили о том, как коллекция на клиенте должна хранить только часть данных, доступных на сервере. Мы даже создали такие коллекции для уведомлений и комментариев.

Не смотря на это, мы все еще публикуем все наши посты за раз, для всех подключенных пользователей. Когда количество постов станет измеряться тысячами, это станет большой проблемой. Чтобы избежать ее, нам надо начать разбивать посты на отдельные страницы.

Для начала давайте создадим больше тестовых постов, чтобы было что разбивать на страницы.

```
// Fixture data
if (Posts.find().count() === 0) {

  //...

  Posts.insert({
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteorbook.com',
    submitted: now - 12 * 3600 * 1000,
    commentsCount: 0
  });

  for (var i = 0; i < 10; i++) {
    Posts.insert({
      title: 'Test post #' + i,
      author: sacha.profile.name,
      userId: sacha._id,
      url: 'http://google.com/?q=test-' + i,
      submitted: now - i * 3600 * 1000,
      commentsCount: 0
    });
  }
}
```

server/fixtures.js

После запуска команды `meteor reset` вы должны получить примерно такую картину:

Displaying dummy data.

Добавили достаточно постов чтобы их можно было разбивать …

Открыть на GitHub | Запустить копию

В лучших традициях современных веб-приложений мы создадим механизм, который будет подгружать новые посты по мере прокрутки страницы вниз. Для начала мы загрузим, скажем, 10 постов, а внизу высветим ссылку “Загрузить еще”. По щелчку на этой ссылке мы подгрузим еще 10 постов, и так *до бесконечности*. Таким образом мы сможем контролировать всю нашу систему разбиения данных на страницы с помощью одного единственного параметра, означающего количество постов, единовременно выводимых на экран.

Нам надо придумать способ сообщить об этом параметре серверу, чтобы тот знал, сколько постов посылать клиенту. У нас уже есть подписка на публикацию `posts` на маршрутизаторе. Мы воспользуемся ей, чтобы дать маршрутизатору возможность управлять нашими страницами.

Самый простой способ передать параметр на сервер будет через URL. Например, в таком формате - `http://localhost:3000/25` - здесь мы передаем значение `25`, про которое сервер догадается, что оно означает количество постов. Дополнительной фишкой будет то, что если пользователь случайно (или намеренно) перезагрузит страницу в браузере, он снова получит то же самое количество постов, что и ранее.

Для этого нам понадобится изменить способ подписки на посты. Точно так же, как и в главе *Комментарии*, мы переместим код подписки с уровня *маршрутизатора* на уровень *маршрута*.

Если вы уже запутались - не пугайтесь. Сейчас все станет яснее, когда мы начнем писать код.

Сначала мы уберем подписку на публикацию `posts` в блоке `Router.configure()`. Удалите `Meteor.subscribe('posts')` и оставьте только подписку на уведомления - `notifications`:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() {
    return [Meteor.subscribe('notifications')]
  }
});
```

lib/router.js

Затем мы добавим параметр `postsLimit` в адрес маршрута. Символ `?` означает, что параметр необязательный. Таким образом наш маршрут будет совпадать не только с `http://localhost:3000/50`, но и с обычным `http://localhost:3000`.

```
Router.map(function() {
  //...

  this.route('postsList', {
    path: '/:postsLimit?'
  });
});
```

lib/router.js

Стоит особенно отметить что маршрут в виде `/:parameter?` будет совпадать со всеми возможными маршрутами. Так как каждый маршрут будет последовательно проверен на совпадение с текущим адресом, стоит уделить особенное внимание объявлению маршрутов в порядке уменьшения конкретности.

Другими словами, более точные маршруты вроде `/posts/:id` должны быть объявлены в начале, а наш маршрут `postsList` стоит переместить ближе к концу файла, так как он будет совпадать практически с любым адресом.

Настало время бросить вызов серьезной проблеме подписки и нахождения верных данных. Определим значение по-умолчанию для случая когда параметр `postsLimit` отсутствует. Пусть это будет “5” - такое значение позволит нам сгенерировать множество страниц для списка постов.

```
Router.map(function() {
  //..

  this.route('postsList', {
    path: '/:postsLimit?',
    waitOn: function() {
      var postsLimit = parseInt(this.params.postsLimit) || 5;
      return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: postsLimit});
    }
  });
});
```

lib/router.js

Обратите внимание на то, как мы передаем JavaScript объект `{limit: postsLimit}` вместе с именем нашей публикации `posts`. Этот объект послужит параметром `options`, когда сервер вызовет `Posts.find()` чтобы получить порцию постов. Давайте переключимся на код сервера и воплотим это:

```
Meteor.publish('posts', function(options) {
  return Posts.find({}, options);
});

Meteor.publish('comments', function(postId) {
  return Comments.find({postId: postId});
});

Meteor.publish('notifications', function() {
  return Notifications.find({userId: this.userId});
});
```

server/publications.js

Наш код публикаций в свою очередь сообщает серверу, что он может доверять всем объектам JavaScript, которые посылает клиент (в нашем случае, `{limit: postsLimit}`). Доверие сервера настолько велико, что он может использовать этот объект в качестве параметра вызова `find()`. Это позволяет пользователям посылать любые опции запроса через консоль браузера.

В нашем случае это вполне безобидно, так как все что пользователь может сделать это поменять посты местами, или изменить значение параметра `limit` (чего мы и добиваемся).

Подобного подхода стоит избегать, когда у объектов есть секретные неопубликованные поля с данными частного характера. Пользователь запросто сможет получить данные из этих полей, слегка подправив содержимое объекта `fields`. По той же причине объект запроса не стоит использовать напрямую как параметр вызова `find()`.

Безопаснее будет передавать отдельные параметры вместо целого объекта - чтобы избежать передачи ненужных полей:

```
Meteor.publish('posts', function(sort, limit) {
  return Posts.find({}, {sort: sort, limit: limit});
});
```

Теперь, когда мы подписываемся на данные на уровне маршрутизатора, стоит установить контекст данных. Мы слегка изменим наш традиционный подход, заставив функцию `data` вернуть объект JavaScript вместо курсора на данные в Mongo. Это позволит создать именной контекст данных, который мы назовем `posts`.

Традиционно контекст данных был доступен как `this` внутри шаблона, но теперь он будет доступен через `posts`. В остальном, следующий код должен быть уже знаком:

```
Router.map(function() {
  this.route('postsList', {
    path: '/:postsLimit?',
    waitOn: function() {
      var limit = parseInt(this.params.postsLimit) || 5;
      return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
    },
    data: function() {
      var limit = parseInt(this.params.postsLimit) || 5;
      return {
        posts: Posts.find({}, {sort: {submitted: -1}, limit: limit})
      };
    }
  });

  //..
});
```

lib/router.js

Теперь когда мы задаем контекст данных на уровне маршрутизатора, можно окончательно избавиться от метода шаблона `posts` в файле `posts_list.js`. И так как мы назвали наш контекст `posts` (точно так же, как и метод), нам даже не нужно трогать шаблон `postsList`.

Файл маршрутизатора `router.js` теперь должен выглядеть так:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() {
    return [Meteor.subscribe('notifications')]
  }
});

Router.map(function() {
  //...

  this.route('postsList', {
    path: '/:postsLimit?',
    waitOn: function() {
      var limit = parseInt(this.params.postsLimit) || 5;
      return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
    },
    data: function() {
      var limit = parseInt(this.params.postsLimit) || 5;
      return {
        posts: Posts.find({}, {sort: {submitted: -1}, limit: limit})
      };
    }
  });
});
```

lib/router.js

Поправили маршрут postsList чтобы он принимал параметр li…

Открыть на GitHub | Запустить копию

Давайте опробуем нашу новенькую систему разбития результатов на страницы в действии. Изменяя параметр в URL мы можем задавать количество постов, выводимых на главную страницу. Попробуйте открыть `http://localhost:3000/3`. Вы должны увидеть что-то вроде такого:

Контролируем количество постов на главной странице.

Почему мы решили подгружать новые посты по мере прокрутки, а не отдельные страницы по 10 постов на каждой? Ведь так, например, делает Google. Все дело в природе реального времени, на которой построен Meteor.

Давайте представим что мы разбиваем коллекцию `Posts` на страницы, как это делает Google с результатами поиска. Мы перешли на вторую страницу, которая высвечивает посты с 10 по 20. Что произойдет, если другой пользователь удалит любой из предыдущих 10 постов?

Так как наше приложение работает в реальном времени, наши данные тут же изменятся. Пост номер 10 превратится в 9 и пропадет со страницы, в то время как пост номер 11 займет его место. В конце-концов результатом окажется то, что на глазах у пользователя посты поменяют места без видимой на то причины.

Даже если бы наш UX дизайн перетерпел бы подобное поведение интерфейса, традиционное разбиение на страницы вовсе нетривиально в техническом воплощении.

Вернемся к предыдущему примеру. Мы опубликовали посты от 10 до 20 из коллекции `Posts`, но как же вы найдете эти посты на клиенте? Вы не можете выбрать посты от 10 до 20, так как их всего 10 в коллекции на клиенте.

Одним из решений было бы опубликовать эти 10 постов на сервере, и затем вызвать `Posts.find()` на клиенте чтобы выбрать для отображения *все* опубликованные посты.

Это сработает только если у вас одна единственная подписка. Но что если у вас появится больше одной подписки на посты, как у нас вскоре и произойдет?

Представим что одна подписка запрашивает посты от 10 до 20, а вторая от 30 до 40. Теперь у вас загружено 20 постов на клиенте, и ни малейшего представления какой из постов принадлежит которой подписке.

Из-за всех этих причин традиционный способ разбиения коллекций на страницы плохо работает вместе с Meteor.

Вы могли заметить что мы дважды повторили линию кода `var limit = parseInt(this.params.postsLimit) || 5;`. Вдобавок, использование предопределенных величин в коде, вроде этого числа “5”, является плохой практикой. Мир от этого не рухнет, но код стоит немного реорганизовать согласно принципам DRY - Don’t Repeat Yourself - “Не повторяйтесь”.

Открываем новую сторону маршрутизатора Iron Router - *Контроллер Маршрутизатора - Route Controller*. Это удобный способ сгруппировать несколько фишек маршрутизатора в один пакет, который легко используется другими маршрутами. В этот раз мы используем его для одного единственного маршрута, но уже в следующей главе вы увидите, насколько он облегчит нам жизнь.

```
PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  limit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.limit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  data: function() {
    return {posts: Posts.find({}, this.findOptions())};
  }
});

Router.map(function() {
  //...

  this.route('postsList', {
    path: '/:postsLimit?',
    controller: PostsListController
  });
});
```

lib/router.js

Рассмотрим код. Сначала мы создали контроллер наследуя его от `RouteController`. Затем был инициализирован параметр `template`, а также новый параметр - `increment`.

Дальше мы задали функцию `limit` которая вернет значение текущего ограничения на количество постов на странице. Функция `findOptions` возвращает объект с параметрами поиска `options`. Возможно, сейчас она вам покажется лишней, но уже скоро она нам понадобится.

Затем мы определили функции `waitOn` и `data` - теперь они используют нашу функцию `findOptions`.

Напоследок через параметр `controller` мы сообщили маршруту `postsList` использовать наш новый контроллер.

Изменили маршрут postsLists чтобы он перенаправлял на кон…

Открыть на GitHub | Запустить копию

Разбиение на страницы работает, и наш код выглядит просто отлично. Осталась одна проблема - переход по страницам пока что работает только, если вы вручную будете менять параметр количества постов в адресной строке. Давайте сделаем все чуть проще и приятнее в использовании.

Нам понадобится кнопка в конце списка с постами - “Загрузить еще постов”. Каждый раз когда пользователь ее нажмет, количество постов на странице увеличится на 5. Если наш текущий URL `http://localhost:3000/5`, нажатие на кнопке должно изменить его на `http://localhost:3000/10`.

Как и ранее, мы добавим логику разбиения на страницы в маршрут. Помните как мы передали контекст данных именной переменной, вместо того чтобы использовать анонимный курсор? Точно так же не существует правила, по которому функция `data` может передавать одни курсоры. Мы воспользуемся той же техникой чтобы сгенерировать URL для кнопки “Загрузить еще постов”.

```
PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  limit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.limit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().fetch().length === this.limit();
    var nextPath = this.route.path({postsLimit: this.limit() + this.increment});
    return {
      posts: this.posts(),
      nextPath: hasMore ? nextPath : null
    };
  }
});
```

lib/router.js

Давайте внимательнее взглянем на это волшебство в маршрутизаторе. Как вы помните, маршрут `postsList` (который наследуется от контроллера `PostsListController`, над которым мы как раз работаем) принимает параметр `postsLimit`.

Когда мы передаем объект `{postsLimit: this.limit() + this.increment}` вызову функции `this.route.path()`, мы говорим маршруту `postsList` создать новый путь, используя этот объект JavaScript как контекст данных.

Другими словами, это то же самое что и использование метода Handlebars `{{pathFor 'postsList'}}`, за исключением того, что мы заменяем непосредственное `this` на наш собственный контекст данных.

Мы берем этот новый путь и добавляем его в контекст данных шаблона, но *только* если еще остались посты. Как это работает на практике?

Вызов `this.limit()` возвращает количество постов, отображаемых на странице. Это будет либо значение в текущем URL, либо значение по-умолчанию (5) - если в URL нет этого параметра.

С другой стороны `this.posts` ссылается на текущий курсор базы данных, и `this.posts.count()` сосчитает количество постов в этом курсоре.

Теперь, если мы запросим `n` количество постов и получим ровно столько постов, можно оставить кнопку “Загрузить еще постов” на странице. Но если мы запросим `n` постов, а получим *меньше* чем `n` в ответ, значит посты закончились и кнопку “Загрузить еще” можно спрятать.

Однако, есть еще один момент. Что если количество постов в базе данных *равно* `n`? В этом случае клиент запросит `n` постов, получит в ответ ровно `n`, и кнопка “Загрузить еще” останется на виду.

К сожалению здесь нет простого решения, и на данный момент мы оставим все как есть.

Все что осталось это добавить сама ссылка “Загрузить еще” после списка постов на странице, и добавить немного логики чтобы эта ссылка отображалась только, если еще остались посты:

```
<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}">Load more</a>
    {{/if}}
  </div>
</template>
```

client/views/posts/posts_list.html

Теперь список постов должен выглядеть так:

Кнопка “Загрузить еще” на странице

Добавили nextPath() в контроллер и используем его для раз…

Открыть на GitHub

Запустить копию

Разбиение на страницы работает, но одна проблема сильно портит впечатление. Каждый раз когда кто-то нажимает ссылку “Загрузить еще” и маршрутизатор запрашивает больше постов, приложение переходит на шаблон `loading` пока данные запрашиваются. В результате страница перепрыгивает на самый верх, и приходится прокручивать вниз чтобы продолжить чтение постов.

Было бы гораздо лучше, если бы мы оставались на той же странице пока грузятся данные, одновременно показывая какой-либо индикатор что данные действительно грузятся. Именно для

этих целей и существует пакет `iron-router-progress`.

Этот пакет позволит нам добавить индикатор загрузки наверху экрана, в стиле браузера Safari на iOS или сайтов вроде Medium и YouTube. Все что нужно сделать это установить сам пакет:

```
meteor add mrt:iron-router-progress
```

Командная консоль bash

Благодаря волшебству умных пакетов `smart packages`, индикатор прогресса тут же заработает в нашем приложении. Он будет активирован для каждого маршрута, и автоматически спрятан как только все данные для маршрута будут загружены.

Из-за того что мы хотим показать список постов даже если мы переходим между страницами, мы не хотим, чтобы триггер загрузка запускался для наших страниц `PostListController` – мы можем это добиться если не будем ждать подписок:

```
PostsListController = RouteController.extend({
  // ...

  onBeforeAction: function() {
    this.postsSub = Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.limit();
    return {
      posts: this.posts(),
      ready: this.postsSub.ready,
      nextPath: hasMore ? this.nextPath() : null
    };
  }
});
```

lib/router.js

Вместо подписки в `waitOn` блоке, мы подписываемся в `onBeforeAction`, держа ссылку на подписку (subscription handle) в `this.postsSub`, таким образом мы сообщаем о “готовности” шаблону.

Затем, в шаблоне, мы можем показать бегунок загрузки в конце списка постов, когда мы подгружаем новый набор постов. Для этого мы проверяем состояние “готовности”, которые мы передали ранее:

```
<template name="postsList">
  <div class="posts">
    {{#each postsWithRank}}
      {{> postItem}}
    {{/each}}

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}">Load more</a>
    {{else}}
      {{#unless ready}}
        {{> spinner}}
      {{/unless}}
    {{/if}}
  </div>
</template>
```

client/views/posts/posts_list.html

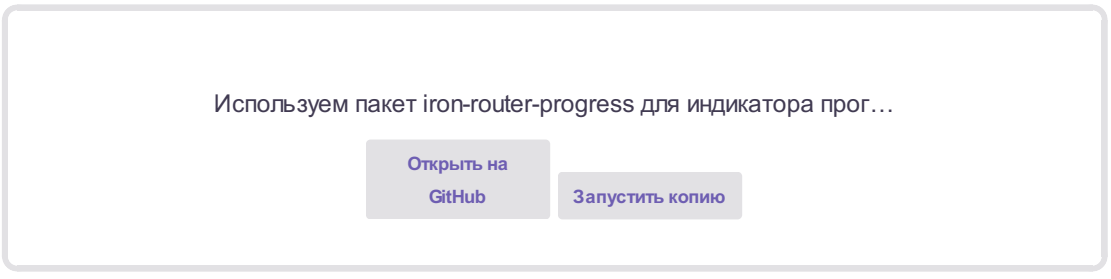
Сделаем еще одну небольшую микро-поправку. Отключим `iron-router-progress` для маршрута `postSubmit`, так как ему не нужно ждать никаких данных (в конце-концов это просто пустая форма на станице):

```
Router.map(function() {

  //...

  this.route('postSubmit', {
    path: '/submit',
    disableProgress: true
  });
});
```

lib/router.js



На данный момент по-умолчанию мы загружаем пять самых новых постов. Но что произойдет если кто-то откроет страницу одного из постов?

Пустой шаблон.

Если вы попробуете открыть один из постов, приложение нарисует шаблон с пустым постом. Здесь есть определенный смысл: мы сообщили маршрутизатору подписаться на публикации `posts` когда загружается маршрут `postsList`. Но мы не сообщили ему что делать с маршрутом `postPage`.

Пока что мы умеем подписываться только на лист из `n` последних постов. Как запросить у сервера один конкретный пост? Оказывается, здесь есть один секрет: у коллекции может быть одновременно несколько публикаций!

Чтобы вернуть наши потерявшиеся посты, мы добавим новую публикацию `singlePost`, которая будет публиковать один пост согласно запрошенному параметру `_id`.

```
Meteor.publish('posts', function(options) {
  return Posts.find({}, options);
});

Meteor.publish('singlePost', function(id) {
  return id && Posts.find(id);
});
```

server/publications.js

Теперь мы можем подписаться на посты на клиенте. У нас уже есть подписка на публикацию `comments` в функции `waitOn` маршрута `postPage`. Мы просто добавим здесь еще одну подписку на `singlePost`. Не забудьте добавить подписку в маршрут `postEdit`, ведь ему потребуются те же самые данные:

```
Router.map(function() {

  //...

  this.route('postPage', {
    path: '/posts/:_id',
    waitOn: function() {
      return [
        Meteor.subscribe('singlePost', this.params._id),
        Meteor.subscribe('comments', this.params._id)
      ];
    },
    data: function() { return Posts.findOne(this.params._id); }
  });

  this.route('postEdit', {
    path: '/posts/:_id/edit',
    waitOn: function() {
      return Meteor.subscribe('singlePost', this.params._id);
    },
    data: function() { return Posts.findOne(this.params._id); }
  });

  /...

});
```

lib/router.js

Подписываемся на посты по-отдельности, чтобы можно было з…

Открыть на GitHub

Запустить копию

Теперь, когда у нас работает разбиение гигантского количества постов на страницы, у приложения не будет проблем с обилием данных, и пользователи смогут запостить еще больше ссылок. Было бы здорово как-то сортировать все эти ссылки и дать пользователям возможность голосовать за лучшие из них. Об этом и будет следующая глава.

Теперь, когда наш сайт становится все популярнее, быстро найти нужную ссылку становится той еще задачей. Чтобы решить эту проблему, хорошо бы нам сделать систему оценок для упорядочивания постов.

Мы можем сделать сложную систему оценивания с кармой, уменьшающимися со временем очками, или еще каким-нибудь из многих способов (большинство которых уже сделаны в **Telescope**, старшем брате Microscope). Но для нашего приложения мы не будем городить что-то сложное, а просто оценим посты по количеству полученных за пост голосов.

Начнем с того, что дадим возможность пользователям голосовать за посты.

Мы будем хранить список голосов пользователей для каждого поста, чтобы помнить о том, надо ли нам показывать кнопку для голосования или нет, чтобы исключить возможность проголосовать дважды.

> Мы будем публиковать эти списки голосов всем пользователям, что автоматически сделает эти данные публично доступными через консоль браузера.
>
> Это тот проблемный случай приватности данных, который может возникнуть вследствие того, как устроена работа с коллекциями. К примеру, хотим ли мы, чтобы пользователи видели кто голосовал за пост? В нашем случае, публичный доступ к этой информации не будет иметь последствий. Но важно как минимум отметить, что такая проблема есть.
>
> Так же следует учесть, что если бы мы *все же хотели* ограничить часть из этой информации, нам следовало бы удостовериться в том, чтобы пользователь не смог бы исказить значения параметров нашей публикации посредством удаления этой опции на стороне сервера, либо не передавая объект целиком от клиента на сервер.

Мы так же денормализуем общее количество голосовавших за пост для того, чтобы было легче получать эти данные. Для этого мы добавим два дополнительных атрибута к нашим ‘постам’ - ‘проголосовавшие’ (`upvoters`) и ‘голоса’ (`votes`). Давайте начнем с того, что добавим их в наш fixture-файл:

```
// Fixture данные
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // создадим двух пользователей
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachaId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sacha = Meteor.users.findOne(sachaId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sacha._id,
    author: sacha.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: now - 7 * 3600 * 1000,
    commentsCount: 2,
    upvoters: [], votes: 0
  });

  Comments.insert({
    postId: telescopeId,
    userId: tom._id,
    author: tom.profile.name,
    submitted: now - 5 * 3600 * 1000,
    body: 'Интересный проектик, Sacha, можно я тоже приму участие?'
  });

  Comments.insert({
    postId: telescopeId,
    userId: sacha._id,
    author: sacha.profile.name,
    submitted: now - 3 * 3600 * 1000,
    body: 'Разумеется можешь, Tom!'
  });

  Posts.insert({
    title: 'Meteor',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://meteor.com',
    submitted: now - 10 * 3600 * 1000,
    commentsCount: 0,
    upvoters: [], votes: 0
  });

  Posts.insert({
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteorbook.com',
    submitted: now - 12 * 3600 * 1000,
    commentsCount: 0,
    upvoters: [], votes: 0
  });

  for (var i = 0; i < 10; i++) {
    Posts.insert({
      title: 'Test post #' + i,
      author: sacha.profile.name,
      userId: sacha._id,
      url: 'http://google.com/?q=test-' + i,
      submitted: now - i * 3600 * 1000,
      commentsCount: 0,
      upvoters: [], votes: 0
    });
```

```
  }
}
```

server/fixtures.js

Как всегда, запустив `meteor reset` перезапустим приложение, и создадим нового пользователя. Так же давайте убедимся в том, что два новых атрибута инициализировались при создании нового поста:

```
//...

// проверим не было ли уже постов с такой же ссылкой
if (postAttributes.url && postWithSameLink) {
  throw new Meteor.Error(302,
    'Эта ссылка уже была опубликована',
    postWithSameLink._id);
}

// возьмем только нужные (безопасные) поля
var post = _.extend(_.pick(postAttributes, 'url', 'title', 'message'), {
  userId: user._id,
  author: user.username,
  submitted: new Date().getTime(),
  commentsCount: 0,
  upvoters: [],
  votes: 0
});

var postId = Posts.insert(post);

return postId;

//...
```

collections/posts.js

Сначала добавим кнопку для голосования в часть, отвечающую за посты:

```
<template name="postItem">
  <div class="post">
    <a href="#" class="upvote btn">⬆</a>
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        {{votes}} Votes,
        submitted by {{author}},
        <a href="{{pathFor 'postPage'}}">{{commentsCount}} comments</a>
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn">Discuss</a>
  </div>
</template>
```

client/views/posts/post_item.html

Кнопка голосования

Далее вызовем Метод, отвечающий за голосование, когда пользователь кликнет по кнопке:

```
//...

Template.postItem.events({
  'click .upvote': function(e) {
    e.preventDefault();
    Meteor.call('upvote', this._id);
  }
});
```

client/views/posts/post_item.js

И в конце, изменим наш `collections/posts.js` файл и добавим Метод на серверной стороне, который будет увеличивать голоса для поста:

```
Meteor.methods({
  post: function(postAttributes) {
    //...
  },

  upvote: function(postId) {
    var user = Meteor.user();
    // удостоверимся, что пользователь залогинен
    if (!user)
      throw new Meteor.Error(401, "Надо залогиниться чтобы голосовать");

    var post = Posts.findOne(postId);
    if (!post)
      throw new Meteor.Error(422, 'Пост не найден');

    if (_.include(post.upvoters, user._id))
      throw new Meteor.Error(422, 'Вы уже голосовали за этот пост');

    Posts.update(post._id, {
      $addToSet: {upvoters: user._id},
      $inc: {votes: 1}
    });
  }
});
```

collections/posts.js

Добавили простой алгоритм голосования.

Открыть на GitHub | Запустить копию

Этот Метод достаточно прямолинеен. Сначала мы делаем парочку проверок, чтобы удостовериться что пользователь залогинен, и что пост действительно существует. После этого проверяем, голосовал ли уже пользователь за этот пост. Если не голосовал, мы увеличиваем общее количество голосов за пост на один, и добавляем пользователя к полю “upvoters”.

Заключительный шаг интересен тем, что мы использовали пару новых Mongo операторов. Существует множество других операторов, которые следует знать - но этот чудовищно полезен: `$addToSet` добавляет элемент к массиву только в том случае, если такового еще в нем нет, а оператор `$inc` попросту увеличивает на единицу цифровое поле.

Если пользователь не залогинен, то он не сможет проголосовать. Чтобы отразить это в нашем дизайне, добавим helper, который по условию добавит `disabled` CSS класс к кнопке “Проголосовать”.

```
<template name="postItem">
  <div class="post">
    <a href="#" class="upvote btn {{upvotedClass}}">⬆</a>
    <div class="post-content">
      //...
  </div>
</template>
```

client/views/posts/post_item.html

```
Template.postItem.helpers({
  ownPost: function() {
    //...
  },
  domain: function() {
    //...
  },
  upvotedClass: function() {
    var userId = Meteor.userId();
    if (userId && !_.include(this.upvoters, userId)) {
      return 'btn-primary upvotable';
    } else {
      return 'disabled';
    }
  }
});

Template.postItem.events({
  'click .upvotable': function(e) {
    e.preventDefault();
    Meteor.call('upvote', this._id);
  }
});
```

client/views/posts/post_item.js

Мы меняем наш класс с `.upvote` на `.upvotable`, поэтому надо не забыть также изменить обработчик на клик.

Обесцвечиваем кнопку голосования.

Обесцветили кнопку голосования если пользователь незалоги…

Открыть на GitHub | Запустить копию

Вы наверное заметили, что посты, у которых есть всего один голос, отмечены как “1 vote**s**”. Пора нам сделать так, чтобы окончания ставились правильно. Делать множественную форму числа - это довольно сложный процесс, но мы пока сделаем все достаточно просто (*примечание*: в отличии от английского языка, в русском нам понадобилось бы немного доработать оригинальный helper, потому что нам понадобится еще одно условие когда количество голосов равно двум, а кроме того окончания еще и будут зависеть от рода, но это все мы опустим, ибо наш сайт работает на английском). Мы создадим такой Handlebars helper, который мы сможем использовать повсюду:

```
UI.registerHelper('pluralize', function(n, thing) {
  // простой как барабан pluralizer
  if (n === 1) {
    return '1 ' + thing;
  } else {
    return n + ' ' + thing + 's';
  }
});
```

client/helpers/handlebars.js

Все helper'ы, которые мы создали до этого были привязаны к менеджеру и шаблону, к которым они применялись. Но если использовать `UI.registerHelper`, мы создадим *глобальный* helper, который мы сможем использовать с любым шаблоном:

```
<template name="postItem">
//...
<p>
  {{pluralize votes "Vote"}},
  submitted by {{author}},
  <a href="{{pathFor 'postPage'}}">{{pluralize commentsCount "comment"}}</a>
  {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
</p>
//...
</template>
```

client/views/posts/post_item.html

Совершенствуем омножествитель.

Добавили helper для выставления правильного окончания.

Открыть на GitHub | Запустить копию

Теперь мы должны увидеть “1 vote”.

Наш код для голосования выглядит вполне себе, но можно сделать его еще лучше. В нашем Методе, отвечающим за голосование, мы делаем два запроса в Mongo: один чтобы взять пост, а второй для того чтобы его обновить.

С этим есть пару нюансов. Во-первых, не очень то эффективно два раза обращаться к базе данных. А во-вторых, что более важно, у нас появляется так называемое состояние гонки (англ. race condition). В нашем случае мы как бы следуем по такому алгоритму:

1. Взять пост из базы данных.
2. Проверить голосовал ли пользователь за этот пост.
3. Если не голосовал, то проголосовать.

Однако что если тот же пользователь будет голосовать за один и тот же пост в промежутке между первым и третьим шагами? Наш текущий код позволит проголосовать за один и тот же пост дважды. Хорошо что на этот счет Mongo позволяет нам быть умнее, объединив первый и третий шаги в один вызов:

```
Meteor.methods({
  post: function(postAttributes) {
    //...
  },

  upvote: function(postId) {
    var user = Meteor.user();
    // удостоверимся, что пользователь залогинен
    if (!user)
      throw new Meteor.Error(401, "Надо залогиниться чтобы голосовать");

    Posts.update({
      _id: postId,
      upvoters: {$ne: user._id}
    }, {
      $addToSet: {upvoters: user._id},
      $inc: {votes: 1}
    });
  }
});
```

collections/posts.js

Улучшенный алгоритм голосования.

Открыть на GitHub | Запустить копию

То, что мы говорим этим, так это “найти все посты вот с этим `id`, где пользователь еще не голосовал, и обновить их таким вот образом”. Если пользователь *еще не* проголосовал, то, конечно же, найдется пост с таким `id`. А если пользователь *уже* проголосовал, то результатом запроса будет пустое множество, и далее ничего не произойдет.

Обратной стороной данного подхода является то, что теперь мы не сможем сказать голосовал ли уже пользователь за этот пост (поскольку мы избавились от запроса в базу данных на соответствующую проверку). Но пользователь и так поймет, голосовал ли он, если кнопка “Проголосовать” будет неактивной.

Давайте предположим, что вы решили схитрить и переместить ваш пост на первое место в рейтинге, немного подправив количество голосов за него:

```
> Posts.update(postId, {$set: {votes: 10000}});
```

Консоль браузера.

(Где `postId` это id вашего поста)

Такого рода наглая попытка поиграть с системой будет поймана нашим `deny()` callback-ом (тот что `collections/posts.js`, помните?) и будет немедленно сведена на нет.

Однако если присмотреться повнимательнее, мы сможем увидеть компенсацию задержки в действии. Это произойдет достаточно быстро, однако можно будет заметить как на короткое время пост переместится на первое место, а потом грохнется обратно там где он был.

Так что же произошло? В нашей локальной коллекции `Posts` обновление произошло без происшествий. Это случилось молниеносно, и наш пост переместился на первое место. В это время на сервере в таком такое обновлении было отказано. Поэтому некоторое время спустя (речь идет о нескольких миллисекундах, если вы пользуетесь Meteor'ом на своей локальной машине), сервер вернул error, тем самым говоря локальной коллекции откатиться обратно.

В конце концов получается, что до тех пор пока сервер не подтвердит запрос, пользовательскому интерфейсу ничего не остается как доверять локальной коллекции. А как только будет получен ответ с серверной части, пользовательский интерфейс прореагирует соответствующе, чтобы отразить результат ответа сервера.

Теперь когда у нас есть количество голосов для каждого поста, давайте покажем список из лучших постов. Для этого мы научимся управлять двумя отдельными подписками коллекции постов, а так же сделаем наш `postsList` шаблон немного более универсальным.

Начнем с того, что нам нужно *две* подписки, одна на каждый вид выборки, или, можно сказать, тип сортировки. Магия тут в том, что обе подписки будут подписаны на одну и ту же коллекцию, но только с разными аргументами!

Мы так же создадим два маршрута, `newPosts` и `bestPosts`, доступных по адресам `/new` и `/best`, соответственно (с учетом использования, например, `/new/5` и `/best/5` для нужд перелистывания на соответствующую страницу).

Для этого мы *расширим* наш `PostsListController` контроллер через два различных контроллера `NewPostsListController` и `BestPostsListController`. Это позволит нам использовать те же самые параметры маршрутизации для маршрутов `home` и `newPosts`, дав `NewPostsListController` контроллеру наследование. К тому же это классная демонстрация того, насколько гибким может быть Iron Router.

```
PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  limit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: this.sort, limit: this.limit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.limit();
    return {
      posts: this.posts(),
      nextPath: hasMore ? this.nextPath() : null
    };
  }
});

NewPostsListController = PostsListController.extend({
  sort: {submitted: -1, _id: -1},
  nextPath: function() {
    return Router.routes.newPosts.path({postsLimit: this.limit() + this.increment})
  }
});

BestPostsListController = PostsListController.extend({
  sort: {votes: -1, submitted: -1, _id: -1},
  nextPath: function() {
    return Router.routes.bestPosts.path({postsLimit: this.limit() + this.increment})
  }
});

Router.map(function() {
  this.route('home', {
    path: '/',
    controller: NewPostsListController
  });

  this.route('newPosts', {
    path: '/new/:postsLimit?',
    controller: NewPostsListController
  });

  this.route('bestPosts', {
    path: '/best/:postsLimit?',
    controller: BestPostsListController
  });
  // ..
});
```

lib/router.js

Заметьте, что теперь когда у нас более чем один маршрут, мы выносим `nextPath` логику из `PostsListController` в `BestPostsListController`, поскольку путь будет различным в обоих случаях.

Так же, сортируя по `votes`, мы используем еще сортировку по времени, чтобы порядок получился правильным.

Теперь, когда наш новый контроллер сделан, мы со спокойной душой можем удалить предыдущий `postsList` маршрут. Просто удалим следующий код:

```
this.route('postsList', {
 path: '/:postsLimit?',
 controller: PostsListController
})
```

lib/router.js

Добавим так же ссылки в заголовок:

```
<template name="header">
  <header class="navbar">
    <div class="navbar-inner">
      <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </a>
      <a class="brand" href="{{pathFor 'home'}}">Microscope</a>
      <div class="nav-collapse collapse">
        <ul class="nav">
          <li>
            <a href="{{pathFor 'newPosts'}}">New</a>
          </li>
          <li>
            <a href="{{pathFor 'bestPosts'}}">Best</a>
          </li>
          {{#if currentUser}}
            <li>
              <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
            </li>
            <li class="dropdown">
              {{> notifications}}
            </li>
          {{/if}}
        </ul>
        <ul class="nav pull-right">
          <li>{{> loginButtons}}</li>
        </ul>
      </div>
    </div>
  </header>
</template>
```

client/views/include/header.html

Нам так же надо обновить наш обработчик удаления поста:

```
  'click .delete': function(e) {
    e.preventDefault();

    if (confirm("Delete this post?")) {
      var currentPostId = this._id;
      Posts.remove(currentPostId);
      Router.go('home');
    }
  }
```

client/views/posts_edit.js

Когда все это сделано, у нас теперь есть список постов ранжированных по количеству собранных голосов:

Ранжируем по голосам.

Добавили пути для списков постов и страницы для их отобра…

Открыть на GitHub | Запустить копию

Теперь когда у нас есть две страницы для отображения списка постов, трудно понять какой именно список мы сейчас смотрим. Поэтом давайте вернемся к заголовку для того, чтобы сделать его более очевидным. Мы создадим новый менеджер `header.js` и создадим helper, который будет использовать текущий путь и один или более именованных маршрутов для того, чтобы установить активный класс для нашей навигации:

Причина зачем нам необходимо поддерживать множество имен маршрутов в том, что оба маршрута `home` и `newPosts` (которые соответствуют путям `/` и `/new` соответственно) используют один и тот же шаблон. Это означает, что наш `activeRouteClass` должен быть достаточно умным, чтобы делать тэг `<li>` активным в обоих случаях.

```
<template name="header">
  <header class="navbar">
    <div class="navbar-inner">
      <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </a>
      <a class="brand" href="{{pathFor 'home'}}">Microscope</a>
      <div class="nav-collapse collapse">
        <ul class="nav">
          <li class="{{activeRouteClass 'home' 'newPosts'}}">
            <a href="{{pathFor 'newPosts'}}">New</a>
          </li>
          <li class="{{activeRouteClass 'bestPosts'}}">
            <a href="{{pathFor 'bestPosts'}}">Best</a>
          </li>
          {{#if currentUser}}
            <li class="{{activeRouteClass 'postSubmit'}}">
              <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
            </li>
            <li class="dropdown">
              {{> notifications}}
            </li>
          {{/if}}
        </ul>
        <ul class="nav pull-right">
          <li>{{> loginButtons}}</li>
        </ul>
      </div>
    </div>
  </header>
</template>
```

client/views/includes/header.html

```
Template.header.helpers({
  activeRouteClass: function(/* route names */) {
    var args = Array.prototype.slice.call(arguments, 0);
    args.pop();

    var active = _.any(args, function(name) {
      return Router.current() && Router.current().route.name === name
    });

    return active && 'active';
  }
});
```

client/views/includes/header.js

Показываем активную страницу.

До этого момента мы не использовали какие-либо специальные конструкции. Так же, как и любые другие Handlebars тэги, тэги шаблонного helper'а тоже могут принимать аргументы.

И, разумеется, хоть вы и можете передавать именные аргументы в вашу функцию, вы также можете передать любое количество анонимных параметров, и доставать их через `arguments` объект внутри функции.

В последнем случае, вы вероятно захотите сконвертировать `arguments` объект в обычный JavaScript массив, и затем вызвать `pop()` на него чтобы избавится от '/', добавленного в конец Handlebars'ом.

Для каждого пути `activeRouteClass` helper возьмет список маршрутных имен, а затем использует с ними Underscore'овский метод `any()` чтобы проверить, проходит ли любой из путей тест (проще говоря, совпадает ли наш URL с каким-либо путем из списка маршрутных имен).

Если какой-либо из путей совпадет с текущим путем, то `any()` вернет `true`. А еще мы воспользуемся особенностью pattern'a `boolean && string` в JavaScript, когда `false && myString` возвратит `false`, а `true && myString` возвратит `myString`.

Добавили активные классы к заголовку.

Открыть на GitHub | Запустить копию

Теперь пользователи могут в реальном времени голосовать за посты, а мы увидем как они прыгают туда-сюда в виду изменения голосов за них. Но разве не было бы круто, если бы все это не сопровождалось парочкой уместных плавных анимаций?

К этому моменту у вас должно быть хорошее представление о том, как между собой работают публикации и подписки. Давайте отбросим в сторону все ограничения и рассмотрим несколько продвинутых примеров.

В **нашей первой главе про публикации** мы познакомились с наиболее распространенными способами работы с публикациями и подписками. Также мы узнали, как функция `_publishCursor` позволяет легко создавать их для наших сайтов.

Давайте вспомним, как именно работает `_publishCursor`. Эта функция находит все документы совпадающие с текущим курсором, и отправляет их в коллекцию на клиенте с *тем же именем*. Обратите внимание, что имя *публикации* здесь не играет никакой роли.

Это означает что мы можем иметь *больше одной публикации*, соединяющей коллекции на клиенте и сервере.

Мы уже видели этот подход в действии в **главе разбиения постов на страницы**, когда публиковалось подмножество постов одновременно с отдельно выбранным постом.

Другим похожим сценарием является публикация *краткого содержания* большого списка документов, одновременно с подробными деталями отдельно выбранного документа.

Двойная публикация коллекции

```
Meteor.publish('allPosts', function() {
  return Posts.find({}, {fields: {title: true, author: true}});
});

Meteor.publish('postDetail', function(postId) {
  return Posts.find(postId);
});
```

Теперь, когда клиент подписан на обе публикации (мы используем `autorun` чтобы быть уверенными, что подписка `postDetail` получает верное значение параметра `postId`), его коллекция `'posts'` пополняется одновременно из двух источников - список заголовков и имена авторов из первой подписки, а также подробные детали поста из второй.

Вы наверное уже догадались, что один и тот же пост публикуется через обе публикации. Первая - `allPosts` - опубликует только часть полей поста, в то время как вторая - `postDetail` - опубликует его целиком. К счастью, Meteor умеет совмещать результаты таких публикаций в один и тот же документ на клиенте, просто объединяя поля и не создавая копий.

Отлично, теперь при отрисовке списка с кратким содержанием постов у нас будет ровно столько данных, сколько требуется. В то же время, когда мы будем загружать страницу с одним

конкретным постом, у нас снова будет все, что нужно для ее отображения. Конечно же, нам нужно позаботиться о том, чтобы клиент не ожидал мгновенной доступности всех полей у всех постов - это довольно распространенная загвоздка.

Стоит упомянуть, что вы не ограничены в вариациях параметров у публикуемых документов. Вы можете опубликовать одни и те же поля в обоих публикациях, но упорядочить сам список публикуемых документов по-разному.

```
Meteor.publish('newPosts', function(limit) {
  return Posts.find({}, {sort: {submitted: -1}, limit: limit});
});

Meteor.publish('bestPosts', function(limit) {
  return Posts.find({}, {sort: {votes: -1, submitted: -1}, limit: limit});
});
```

server/publications.js

Мы только что наблюдали, как можно опубликовать коллекцию более одного раза. Оказывается, похожего результата можно добиться и другим способом: создать одну публикацию, но подписаться на нее *несколько* раз.

В Microscope мы подписываемся на публикацию `posts` несколько раз, но Iron Router заботится об этих подписках за нас. Не смотря на это, нам ничто не мешает подписаться несколько раз *одновременно*.

Скажем, нам нужно загрузить одновременно самые последние и самые лучшие посты:

Подписываемся дважды на одну публикацию

Мы создаем одну публикацию:

```
Meteor.publish('posts', function(options) {
  return Posts.find({}, options);
});
```

Затем мы подписываемся на эту публикацию несколько раз. Это примерно то же самое, что мы делаем в Microscope:

```
Meteor.subscribe('posts', {submitted: -1, limit: 10});
Meteor.subscribe('posts', {baseScore: -1, submitted: -1, limit: 10});
```

Что же тут происходит? Каждый браузер открывает *две* разных подписки, каждая из которых присоединяется к *одной и той же* публикации на сервере.

Каждая подписка задает разные аргументы, и для каждой подписки свой список документов собирается на сервере и отправляется в коллекцию клиента.

В отличие от традиционных реляционных баз данных как MySQL, которые используют *объединения данных* (*joins*), базы данных типа NoSQL вроде Mongo используют *денормализацию* и *внедрение*. Давайте рассмотрим, как это работает в случае с Meteor.

Взглянем на конкретный пример. Мы добавили комментарии к постам, и пока что были вполне довольны тем, как комментарии публикуются только к одному конкретному посту, открытому в данный момент пользователем.

Предположим, мы хотели бы отобразить *все* комментарии ко *всем* постам на главной странице (эти посты будут меняться, когда мы будем пролистывать страницы списка). Этот случай дает хороший повод внедрить комментарии в посты. По этой же причине мы когда-то денормализовали значение количества постов в документ с постом.

Конечно, мы могли бы просто внедрить комментарии в посты, и избавиться от коллекции `Comments`. Однако, как мы уже помним из главы *Денормализация*, это лишило бы нас многих приятных фишек работы с отдельной коллекцией.

Оказывается, есть один трюк с подпиской, который позволяет внедрить комментарии в посты, одновременно сохраняя свою коллекцию с комментариями.

Предположим что одновременно со списком постов на главной странице мы хотели бы подписаться на список с 2 последними комментариями к каждому посту.

Если бы комментарии были отдельной публикацией, добиться подобной функциональности было бы сложно. Особенно, если список постов был бы как-то ограничен (например, до 10 последних постов). Нам пришлось бы написать публикацию вроде такой:

Две коллекции в одной подписке

```
Meteor.publish('topComments', function(topPostIds) {
  return Comments.find({postId: topPostIds});
});
```

Это было бы проблемой с точки зрения быстродействия, ведь публикацию пришлось бы разрывать и создавать заново каждый раз, когда меняется список `topPostIds`.

Эту проблему можно обойти, если принять тот факт, что можно не только иметь несколько *публикаций* для одной *коллекции*, но и несколько *коллекций* на одну *публикацию*:

```
Meteor.publish('topPosts', function(limit) {
  var sub = this, commentHandles = [], postHandle = null;

  // посылаем два последних комментария одного поста
  function publishPostComments(postId) {
    var commentsCursor = Comments.find({postId: postId}, {limit: 2});
    commentHandles[post._id] =
      Meteor.Collection._publishCursor(commentsCursor, sub, 'comments');
  }

  postHandle = Posts.find({}, {limit: limit}).observeChanges({
    added: function(id, post) {
      publishPostComments(post._id);
      sub.added('posts', id, post);
    },
    changed: function(id, fields) {
      sub.changed('posts', id, fields);
    },
    removed: function(id) {
      // прекращаем следить за изменениями к коментариям поста
      commentHandles[id] && commentHandles[id].stop();
      // удаляем пост
      sub.removed('posts', id);
    }
  });

  sub.ready();

  // проверяем что мы все почистили (внимание: `_publishCursor`
  // делает это за нас с помощью наблюдателей за комментариями)
  sub.onStop(function() { postsHandle.stop(); });
});
```

Обратите внимание, мы ничего не возвращаем в этой публикации. Вместо этого посылается сообщение в `sub` с помощью функции `.added()` и ее друзей. Так что нам не нужно просить `_publishCursor` сделать это за нас с помощью возвращаемого курсора.

Теперь, каждый раз когда будет публиковаться пост, вместе с ним будут публиковаться два последних комментария к этому посту. И все это с помощью одного единственного вызова подписки.

Не смотря на то, что Meteor мало рекламирует подобный подход, вы можете также ознакомиться с пакетом `publish-with-relations` на Atmosphere, который позволяет легко использовать подобный сценарий.

Что еще интересного мы можем сделать с новообретенным знанием о гибкости подписок? Например, если не используется `_publishCursor`, на нас больше не распространяется ограничение, что коллекция на сервере должна иметь то же самое имя и на клиенте.

Одна коллекция для двух подписок

Одна из причин по которой стоит воспользоваться этой фишкой это паттерн *Single Table Inheritance - Наследование в одной таблице*.

Предположим, мы хотели бы ссылаться на различные типы объектов из наших постов. У каждого из типов были бы общие поля, в добавок к своим уникальным полям. Например, мы могли бы создать движок для блога в стиле Tumblr, где у каждого поста может быть ID, время создания, заголовок, но также может быть картинка, видео, линк или просто текст.

Не смотря на то, что у нас одна единственная коллекция `Resources` на сервере, мы можем превратить ее в несколько коллекций на клиенте типа `Видео`, `Картинки` и так далее. В этом нам помогут несколько волшебных строк кода:

```
  Meteor.publish('videos', function() {
    var sub = this;

    var videosCursor = Resources.find({type: 'video'});
    Meteor.Collection._publishCursor(videosCursor, sub, 'videos');

    // _publishCursor не вызывает следующую функцию за нас в случае если мы вызываем дан
ный код несколько раз
    sub.ready();
  });
```

Мы сообщаем `_publishCursor` публиковать наши видео, точно так же, как это делал бы курсор - но вместо публикации в коллекцию `resources` на клиенте, мы публикуем их в коллекцию `'videos'`.

Хорошая ли это идея? Не будем высказывать здесь мнений. В любом случае неплохо знать то, на что способен Meteor.

В нашем приложении уже есть голосование в реальном времени, голосование за посты и рейтинги лучших постов. Из-за такого обилия функционала наши посты прыгают по странице и это не производит хорошего впечатления на пользователей. В этой главе мы научимся добавлять анимацию, чтобы сгладить все перемещения элементов.

Прежде, чем приступить к самой веселой части (анимировании элементов приложения), нам необходимо понять, как Meteor взаимодействует с DOM (или Document Object Model - набор элементов HTML, представляющие содержимое страниц).

Важный момент, который следует усвоить, заключается в том, что элементы DOM на самом деле не могут быть перемещены; однако их можно удалять и создавать (запомните, что это ограничение DOM, а не Meteor). И чтобы создать иллюзию перемены элементов A и B местами, Meteor будет удалять элемент B и создавать его новую копию перед элементом A.

Из-за этого процесс анимации становится непростым. Мы не можем просто переместить элемент B в его новую позицию, так как B будет удален сразу же, как только Meteor обновит страницу (которая, благодаря реактивности, обновится мгновенно). Но не волнуйтесь, мы найдем решение.

Был 1980 год, самый разгар Холодной Войны. Олимпийские игры проходили в Москве, и Советский Союз был полон решимости победить любой ценой в забеге на 100 метров. Для этого группа лучших советских ученых экипировала одного из своих бегунов телепортером, чтобы мгновенно переместить его к финишной черте сразу после выстрела.

К счастью, судьи сразу заметили нарушение, и атлету пришлось телепортироваться обратно на старт, чтобы быть допущенным к участию в гонке, как все остальные.

Мои исторические источники ненадежны, поэтому вам не стоит принимать эту историю за чистую монету. Но попробуйте держать в уме аналогию про “советского бегуна с телепортером”, пока читаете эту главу.

Когда Meteor получит обновление и реактивно изменит DOM, наш пост, словно советский бегун, мгновенно перенесется в его финальное положение. Но ни на Олимпийских играх, ни в нашем приложении мы не можем просто телепортировать все подряд. Поэтому, мы телепортируем элемент назад к стартовой позиции и будем двигать его (другими словами, анимировать) к финишной черте.

Итак, чтобы поменять посты A и B (расположенные в позициях p1 и p2 соответственно), мы совершим следующие действия:

1. Удалим B
2. Создадим B' перед A, в DOM
3. Телепортируем B' в p2
4. Телепортируем A в p1
5. Анимируем A в p2
6. Анимируем B' в p1

Следующая диаграмма объясняет эти шаги в деталях:

Повторюсь, в шагах 3 и 4 мы не *анимируем* A и B' в их позиции, а мгновенно их “тепепортируем”. Так как это происходит мгновенно, пользователю может показаться что они никогда не были удалены, и корректные положения обоих элементов анимируются вперед к их новым значениям.

К счастью, Meteor берет на себя заботу о двух первых шагах, так что нам нужно подумать только о шагах с 3 по 6.

Более того, в шагах 5 и 6 все что нам нужно сделать это передвинуть элементы в их должное положение. Таким образом, в действительности, нам нужно подумать только о пунктах 3 и 4, т.е., отправить элементы в их начальную позицию анимации.

До этого момента мы говорили о том, *как* анимировать наши посты, но не *когда* анимировать их.

Для шагов 3 и 4 ответ будет следующим - всякий раз, когда изменяется свойство поста ‘_rank’ (от которого и зависит позиция поста в списке).

Шаги 5 и 6 будут хитрее. Представьте следующую вещь: если вы сообщите абсолютно логичному андроиду бежать на север 5 минут, а затем бежать 5 минут на на юг, он скорее всего сообразит, что если он должен закончить пробежку на стартовом месте, будет проще сохранить энергию и не бежать вовсе.

И если вы хотите убедится, что ваш андроид бежал 10 минут, вам придется *подождать*, пока он не пробежит первые 5 минут, и *затем* приказать ему бежать обратно.

Браузер работает похожим образом: если мы просто дадим обе инструкции одновременно, то старые координаты будут переписаны новыми и ничего не случится. Другими словами, браузер должен регистрировать изменение позиций как отдельные точки во времени, иначе он не сможет их анимировать.

Meteor не предоставляет встроенного коллбека на этот случай, но мы можем имитировать его, используя `Meteor.setTimeout()`, который просто берет функцию и откладывает ее выполнение на несколько миллисекунд.

Чтобы анимировать реорганизацию постов в списке, нам следует вторгнуться на территорию CSS. Давайте быстро повторим основные правила, как элементы позиционируются на странице с помощью CSS.

Элементы страницы по умолчанию используют **статичное - static** позиционирование. Статически размещенный элемент просто располагается в потоке на странице, и его экранные координаты не могут быть изменены или анимированны.

Relative - относительное позиционирование в свою очередь подразумевает, что элемент позиционируется традиционным способом на странице, но может быть также сдвинут *относительно своего изначального положения*.

Absolute - абсолютное позиционирование делает еще один шаг вперед и позволяет вам задать конкретные координаты x/y относительно **корневого документа** или **первого абсолютно или относительно позиционированного элемента-родителя**.

Мы будем использовать относительное позиционирование для анимации наших постов. Мы уже позаботились о CSS для вас, но если вам хочется добавить стили самостоятельно, просто добавьте этот код в ваш CSS файл:

```
.post{
  position:relative;
  transition:all 300ms 0ms ease-in;
}
```

client/stylesheets/style.css

Шаги 5-6 будут совсем простыми: вам нужно только сбросить значение `top` на `0px` (это его значение по-умолчанию), и наши посты плавно сдвинутся обратно к их “нормальной” позиции.

Другими словами, наша задача - понять, *откуда* анимировать посты (шаги 3 и 4). То есть, на сколько пикселей нужно сдвинуть посты. Но это совсем несложно: правильный сдвиг можно вычислить, отняв координаты новой позиции поста от предыдущей.

Мы могли бы использовать `position:absolute` вместе с относительным родительским элементом для позиционирования наших постов. Но у абсолютного позиционирования есть один большой минус - они совершенно выпадают из структуры страницы, заставляя родительский элемент сжаться, словно внутри ничего нет.

Чтобы компенсировать это, нам придется воспользоваться арсеналом JavaScript, искуственно выставив размеры родительского контейнера - вместо того, чтобы дать браузеру возможность позиционировать элементы естественным образом. Так что лучше использовать относительное позиционирование.

Есть еще одна загвоздка. В то время как элемент A остается в DOM и, таким образом, “помнит” свою предыдущую позицию, элемент B переживает реинкарнацию и возвращается к жизни как новый элемент B', с абсолютно стертой памятью.

Чтобы решить эту задачу, мы воспользуемся **локальной коллекцией**. В нее мы сохраним текущую позицию поста на странице. Локальная коллекция работает точно так же, как и обычная коллекция Meteor, за исключением того, что она остается *только* в памяти браузера (не отправляет данные на сервер). Таким образом, даже если пост удален и воссоздан заново, мы все еще сможем понять, с какого места на странице его надо анимировать.

Мы много говорили про рейтинг постов, но этот “рейтинг” не существует как отдельное свойство поста, а является всего лишь следствием того, в каком порядке посты сохранены в нашей коллекции. Если мы хотим анимировать посты согласно их рейтингу, нам надо наколдовать это свойство из воздуха.

Обратите внимание: мы не можем просто добавить свойство `rank` в базу данных, так как рейтинг это относительное свойство, которое просто зависит от того, как мы сортируем посты (то есть, отдельно взятый пост может иметь первое место при сортировке по дате, но третье место при сортировке по количеству голосов).

В идеале, мы хотели бы добавить это свойство в коллекции `newPosts` и `topPosts`, но Meteor на данный момент не предлагает подходящего механизма для этого.

Вместо этого мы добавим свойство `rank` на самом последнем шаге, в менеджере шаблона `postList`:

```
Template.postsList.helpers({
  postsWithRank: function() {
    this.posts.rewind();
    return this.posts.map(function(post, index, cursor) {
      post._rank = index;
      return post;
    });
  }
});
```

/client/views/posts/posts_list.js

Вместо того чтобы просто вернуть курсор `Posts.find({}, {sort: {submitted: -1}, limit: postsHandle.limit()})` как наш предыдущий хелпер `posts`, `postsWithRank` принимает курсор и добавляет свойство `_rank` к каждому из его документов.

Не забудьте обновить шаблон `postsList`:

```
<template name="postsList">
  <div class="posts">
    {{#each postsWithRank}}
      {{> postItem}}
    {{/each}}

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}">Load more</a>
    {{/if}}
  </div>
</template>
```

/client/views/posts/posts_list.html

> Meteor является одной из самых передовых сред разработки веб-приложений. Но одна из его особенностей как будто пришла со времен видеомагнитофонов и записи на видеокассеты. Мы говорим о функции `rewind()`.
>
> Каждый раз, когда вы вызываете функции `forEach()`, `map()`, или `fetch()` для курсора, вам нужно перемотать курсор на место перед тем, как им можно воспользоваться снова.
>
> Поэтому, на всякий случай, курсор стоит перематывать каждый раз и не рисковать возможностью ошибиться.

Так как наша анимация будет влиять на CSS атрибуты и классы нашего DOM элемента, мы добавим динамический хелпер `{{attributes}}` нашему шаблону `postItem`:

```
<template name="postItem">
  <div class="post" {{attributes}}>

  //..

</template>
```

/client/views/posts/post_item.html

Используя хелпер `{{attributes}}`, мы также открываем скрытое свойство Spacebars - любое свойство возвращаемого объекта `attributes` будет автоматически соотнесено с HTML атрибутами DOM элемента (такими как `class`, `style`, и так далее).

Давайте соберем все вместе, создав хелпер `attributes`:

```
var POST_HEIGHT = 80;
var Positions = new Meteor.Collection(null);

Template.postItem.helpers({

  //..

  attributes: function() {
    var post = _.extend({}, Positions.findOne({postId: this._id}), this);
    var newPosition = post._rank * POST_HEIGHT;
    var attributes = {};

    if (! _.isUndefined(post.position)) {
      var offset = post.position - newPosition;
      attributes.style = "top: " + offset + "px";
      if (offset === 0)
        attributes.class = "post animate"
    }

    Meteor.setTimeout(function() {
      Positions.upsert({postId: post._id}, {$set: {position: newPosition}})
    });

    return attributes;
  }
});

//..
```

/client/views/posts/post_item.js

В начале нашего документа мы устанавливаем значение `height` - высоту для каждого DOM элемента - то есть, наших `div` элементов с постами. Если что-то повлияет на данное значение `height`, например, длинный текст заголовка поста займет больше одной строки, наша анимация сломается. Но на данный момент, чтобы слегка упростить вещи, мы предположим, что каждый пост будет ровно 80 пикселей в высоту.

Далее мы объявляем локальную коллекцию под названием `Positions`. Обратите внимание как мы передаем `null` в качестве аргумента - это дает Meteor знать что мы создаем именно локальную коллекцию (только для клиента).

Все готово, чтобы создать наш хелпер `attributes`.

Иногда бывает непросто понять, когда кусочек реактивного кода будет запущен. Давайте подробнее взглянем на хелпер `attributes`.

Как любой хелпер, он будет запущен, когда шаблон будет отрисован. Из-за его зависимости к атрибуту `_rank` он также будет перезапущен каждый раз, когда рейтинг поста изменится. И, наконец, его зависимость от коллекции `Positions` означает, что он будет перезапущен, когда данный объект будет отредактирован.

Следовательно, хелпер может быть запущен два или три раза подряд. Такое положение вещей может прозвучать расточительным, но это именно то, как реактивность работает. Как только вы привыкнете к ней, это станет неотъемлемой частью разработки приложения, подходом к написанию кода.

Для начала мы найдем позицию поста в коллекции `Positions` и расширим `this` (который, в данном случае, соотносится с текущим постом) результатом нашего запроса. Мы воспользуемся атрибутом `_rank` для расчета новых координат DOM элемента относительно начала страницы.

Затем мы должны позаботиться о двух сценариях - либо хелпер запущен потому, что шаблон отрисовывается (A), или он запущен реактивно, потому что изменено значение атрибута (B).

Нам нужно анимировать элемент только в случае B, поэтому для начала мы убедимся, что свойство `post.position` существует и имеет значение (*как именно* оно было создано, мы скоро узнаем).

Вдобавок ко всему, сценарий B имеет два возможных варианта: B1 и B2; либо мы *телепортируем* наш элемент DOM назад на стартовую позицию (его предыдущую позицию), либо мы *анимируем* его с предыдущей позиции на новую.

Тут-то и стоит воспользоваться переменной `offset`. Так как мы используем *относительное* - *relative* позиционирование, нужно рассчитать новые координаты *относительно* текущей позиции. Этого можно достичь вычитанием новой позиции из предыдущей.

Чтобы отличить случай B1 от B2, мы просто проверим на значение свойство `offset`: если `offset` отличается от 0, это означает что мы *отодвигаем* элемент от его первоначального положения, и мы можем добавить класс `animate` к этому элементу, чтобы его перемещение было анимировано с помощью волшебства CSS transition.

Эти три ситуации (A, B1 и B2) запускаются реактивным способом, когда значение определенных атрибутов меняется. В этом случае, функция `setTimeout` запускает переопределение реактивного контекста, изменяя коллекцию `Positions`.

Поэтому, когда пользователь впервые загружает страницу, весь реактивный процесс происходит

следующим образом:

- Хелпер `attributes` запускается в первый раз.
- Значение `post.position` не определено **(A)**.
- `setTimeout` запускается и определяет значение `post.position`.
- Хелпер `attributes` реактивно перезапускается.
- Перемещения поста не произошло, поэтому значение параметра `offset` меняется с 0 на 0 (анимация не происходит) **(B2)**.

А вот что происходит, когда пользователь голосует за пост:

- Значение `_rank` меняется, что запускает переопределение хелпера `attributes`.
- Значение `post.position` определено **(B)**.
- Значение `offset` не равно 0, значит анимации нет **(B1)**.
- Запускается `setTimeout`, который переопределяет значение `post.position`.
- Хелпер `attributes` реактивно перезапускается.
- Значение `offset` меняется назад на 0 (с анимацией) **(B2)**.

Откройте сайт и попробуйте проголосовать за несколько постов. Вы должны увидеть, как посты с плавной грацией перемещаются вверх и вниз.

Добавили анимацию к реорганизации списка постов.

Открыть на GitHub | Запустить копию

Наши посты правильно реорганизуются, но у нас все еще отсутствует анимация для новых постов. Вместо того, чтобы скучно добавлять посты поверх списка, давайте добавим эффект плавного появления.

```
//..

attributes: function() {
  var post = _.extend({}, Positions.findOne({postId: this._id}), this);
  var newPosition = post._rank * POST_HEIGHT;
  var attributes = {};

  if (_.isUndefined(post.position)) {
    attributes.class = 'post invisible';
  } else {
    var delta = post.position - newPosition;
    attributes.style = "top: " + delta + "px";
    if (delta === 0)
      attributes.class = "post animate"
  }

  Meteor.setTimeout(function() {
    Positions.upsert({postId: post._id}, {$set: {position: newPosition}})
  });

  return attributes;
}

//..
```

/client/views/posts/post_item.js

Мы изолируем сценарий **(A)** и добавляем нашему элементу CSS класс `invisible`. Когда хелпер реактивно перезапускается, и элемент получает класс `animate`, разница в значениях прозрачности (opacity) будет анимирована, и элемент плавно проявится на странице.

Плавное проявление постов.

Открыть на GitHub | Запустить копию

Вы наверное обратили внимание, что мы используем CSS класс `.invisible` для запуска анимации вместо того, чтобы анимировать CSS параметр `opacity` напрямую, как мы это делали с параметром `top`. Это все потому, что нам нужно было анимировать `top` до определенного значения, которое зависело от данных конкретного поста.

С другой стороны, в данном случае мы только хотим показать или спрятать элемент вне зависимости от его содержимого. Хорошей практикой обычно является держать стили CSS и логику JavaScript отдельно друг от друга, поэтому здесь мы будем только добавлять и удалять класс элемента, а саму анимацию определим в стилях CSS.

Наша анимация должна работать так, как и задумывалась с самого начала. Загрузите приложение и попробуйте! Также вы можете поиграть с классами `.post.animated` и попробовать разные эффекты перехода - transitions. Подсказка: **CSS функции с разнообразными кривыми анимации** отличное место для старта!

Мы надеемся, что предыдущие главы хорошо показали вам, что кроется за постройкой Meteor приложения. Каков будет ваш дальнейший путь?

Прежде всего, вы можете купить **Полное (Full)** или **Премиум (Premium)** издание книги, чтобы получить доступ к дополнительным главам. Эти главы проведут вас через жизненные сценарии, такие как наладка API для вашего приложения, интеграция с посторонними сервисами и миграция данных.

В дополнение к официальной **документации**, **Справочник Meteor (Meteor Manual)** копает глубже в конкретные темы как Blaze и Deps.

Если вы хотите глубже окунуться в хитросплетения Meteor, тогда мы очень рекомендуем взглянуть на **Evented Mind** от Криса Мэсэра (Chris Mather), платформу с обучающими видео, где вы можете найти более 50 видео только о Meteor (новые видео выходят каждую неделю).

Один из лучших способов держать руку на пульсе Meteor - подписаться на еженедельную подборку **MeteorHacks** от Аруноды Сусирипала (Arunoda Susiripala). Блог MeteorHacks является отличным ресурсом для продвинутых инструкций по Meteor.

Atmosphere, индекс пакетов Meteor. Отличное место, чтобы узнать что-то новое: вы можете обнаружить новые пакеты и взглянуть на их исходный код, чтобы понять общие практики в сообществе.

Внимание: Atmosphere частично поддерживается Томом Колманом (Tom Coleman), одним из авторов этой книги.

Meteorpedia - это энциклопедия про всякое о Meteor. И конечно же, разработана на Meteor!

Джош и Рай (Josh & Ry) из **Differential**(Meteor-ориентированного агенства) еженедельно записывают **Meteor Подкаст (The Meteor Podcast)**. Это еще один отличный путь держать ухо

востро о событиях в сообществе Meteor.

Стэфан Хокхаус (Stephan Hochhaus) собрал довольно обширный список ресурсов о Meteor: **Meteor resources**.

Мануэль Шоебель (Manuel Schoebel) ведет блог с хорошими статьми о Meteor. От него не отстает и **Gentlenode blog**.

Если вы где-то застряли, то лучше всего спросить на **Stack Overflow**. Не забудьте пометить вопрос ярлыком ‘meteor’.

И наконец, лучший способ не отставать от жизни Meteor - это быть активным участником сообщества. Мы рекомендуем подписаться на рассылочный лист **Meteor mailing list** и следить за **Meteor Core** и **Meteor Talk** форумами на Google Groups, а также зарегистрируйтесь на форуме **Crater.io**.

Каждый месяц во всем мире проходит огромное количество сходок о Meteor. Некоторые записи выступлений и докладов можно найти на **Youtube канале Meteor**.

Когда мы говорим о Клиенте (Client), обычно имеется в виду код программы, который запускается в веб-браузере пользователей. Это может быть как и традиционный браузер (например, Firefox, Google Chrome или Safari), так и что-нибудь сложное (например, UIWebView в родном приложении для iPhone).

Meteor Collection, Коллекция - это хранилище данных, которое автоматически синхронизируется между клиентом и сервером. У Коллекций есть имя (например, `posts`) и обычно они доступны и на клиенте, и на сервере.

Хоть и работают они по разному, у Коллекций на клиенте и на сервере есть общий интерфейс (свойства и методы), который базируется на интерфейсе MongoDB.

Вычисление, Computation - это блок кода, который запускается каждый раз, когда хоть один из реактивных источников информации, которые вычисление использует, меняется.

Если у вас есть реактивный источник данных (например, переменная Session) и вы хотите реагировать на его изменения, вам нужно будет для этого запустить вычисление.

Курсор, Cursor - это результат запроса Коллекции MongoDB. На клиентской стороне, курсор - это не просто массив с результатами запроса, а *реактивный* источник данных, который можно обозревать: получать уведомления, когда уместные документы добавляются, удаляются или обновляются (added, removed, changed).

Distributed Data Protocol (DDP)(протокол распределенных данных) - протокол связи Meteor, который используется для синхронизации Коллекций и вызова Методов. DDP был предназначен как протокол общего назначения, который занимает нишь HTTP для приложений реального времени, которые требуют много операций с информацией.

Deps (от англ. “Dependencies” - зависимости) - это реактивная система Meteor. Deps используется за кулисами на клиентской части для поддержания автоматической синхронизации HTML с моделью данных (data model) лежащей в основе.

MongoDB является документо-ориентированным хранилищем данных, поэтому объекты

хранящиеся в Коллекциях называют “документами”. Они являются простыми JavaScript объектами (за исключением того, что они не могут содержать функций) с единственным специальным полем `_id`, которое Meteor использует для слежения за объектами переданных через DDP.

Когда шаблону (Template) нужно отрисовать (render) на странице что-то более сложное, чем свойство документа, на помощь приходит вспомогательный метод, который будет вызван шаблоном.

Это механизм, который позволяет симулировать вызов удаленного Метода (Method) на стороне клиента, чтобы избежать запаздывание во время ожидания ответа от сервера.

Meteor Method - это удаленный вызов процедуры (RPC) с клиентской стороны к серверу, которые умеют следить за обновлениями коллекций и выполняют Компенсацию Задержки Передачи Данных.

Коллекция на стороне клиента является структурой данных, которая хранится в оперативной памяти и предоставляет интерфейс схожий с MongoDB. Библиотека, которая поддерживает это поведение называется “MiniMongo”. Название указывает на то, что это малая версия Mongo, которая выполняется полностью в памяти.

Пакет Meteor может содержать код работающий на стороне сервер, код работающий на стороне клиенте, инструкции, как обрабатывать ресурсы (например, из SCSS получить CSS) и ресурсы для обработки.
Пакет можно представить как очень продвинутую библиотеку. Meteor поставляется с уже обширным набором пакетов. Также стоит обратить внимание на **Atmosphere** - коллекцию пакетов разработанных сообществом и третьими лицами.

Публикацией является именованный набор данных, который может быть определен индивидуально для каждого пользователя, который на нее подпишется. Вы можете определить публикации на сервере.

Meteor сервер - это HTTP и DDP сервер, который исполняется на Node.js. Он состоит из всех библиотек Meteor вместе с вашим JavaScript кодом для стороны сервера. Когда вы запускаете Meteor сервер, он подсоединяется к базе данных Mongo (которая запускается сама в режиме разработки).

Session в Meteor называется реактивный источник данных, который используется вашим приложением на стороне клиента, чтобы задать и отследить состояние в котором находится пользователь.

Подпиской является соединение к публикации для определенного клиента. Подписка - это код, который исполняется в браузере и переговаривает с публикацией на сервере, а также хранит данные в синхронизированном состоянии.

Шаблон - это механизм генерации HTML кода с помощью JavaScript. По-умолчанию, Meteor поддерживает Handlebars, систему шаблонов без собственной логики, однако в планах поддержка большего числа систем шаблонов.

Когда шаблон отрисовывает содержимое, он обращается к JavaScript объекту, который предоставляет данные для этой конкретной отрисовки. Обычно эти объекты являются простыми структурами JavaScript (plain-old-JavaScript-objects (POJOs)), зачастую - документы из коллекции. Однако они могут быть более сложными и иметь собственные методы.