РАЗРАБОТКА ВЕБ-САЙТОВ*, REACTJS*, NODE.JS*, JAVASCRIPT*, БЛОГ КОМПАНИИ MAIL.RU GROUP

Руководство по работе с Redux

ПЕРЕВОД

Infected 16 июня 2016 в 18:20 204k

Оригинал: Tero Parviainen

Сегодня Redux — это одно из наиболее интересных явлений мира JavaScript. Он выделяется из сотни библиотек и фреймворков тем, что грамотно решает множество разных вопросов путем введения простой и предсказуемой модели состояний, уклоне на функциональное программирование и неизменяемые данные, предоставления компактного API. Что ещё нужно для счастья? Redux — библиотека очень маленькая, и выучить её API не сложно. Но у многих людей происходит своеобразный разрыв шаблона — небольшое количество компонентов и добровольные ограничения чистых функций и неизменяемых данных могут показаться неоправданным принуждением. Каким именно образом работать в таких условиях?

В этом руководстве мы рассмотрим создание с нуля full-stack

приложения с использованием Redux и Immutable-js. Применив подход TDD, пройдём все этапы конструирования Node+Redux бэкенда и React+Redux фронтенда приложения. Помимо этого мы будем использовать такие инструменты, как ES6, Babel, Socket.io, Webpack и Mocha. Набор весьма любопытный, и вы мигом его освоите!

Содержание статьи

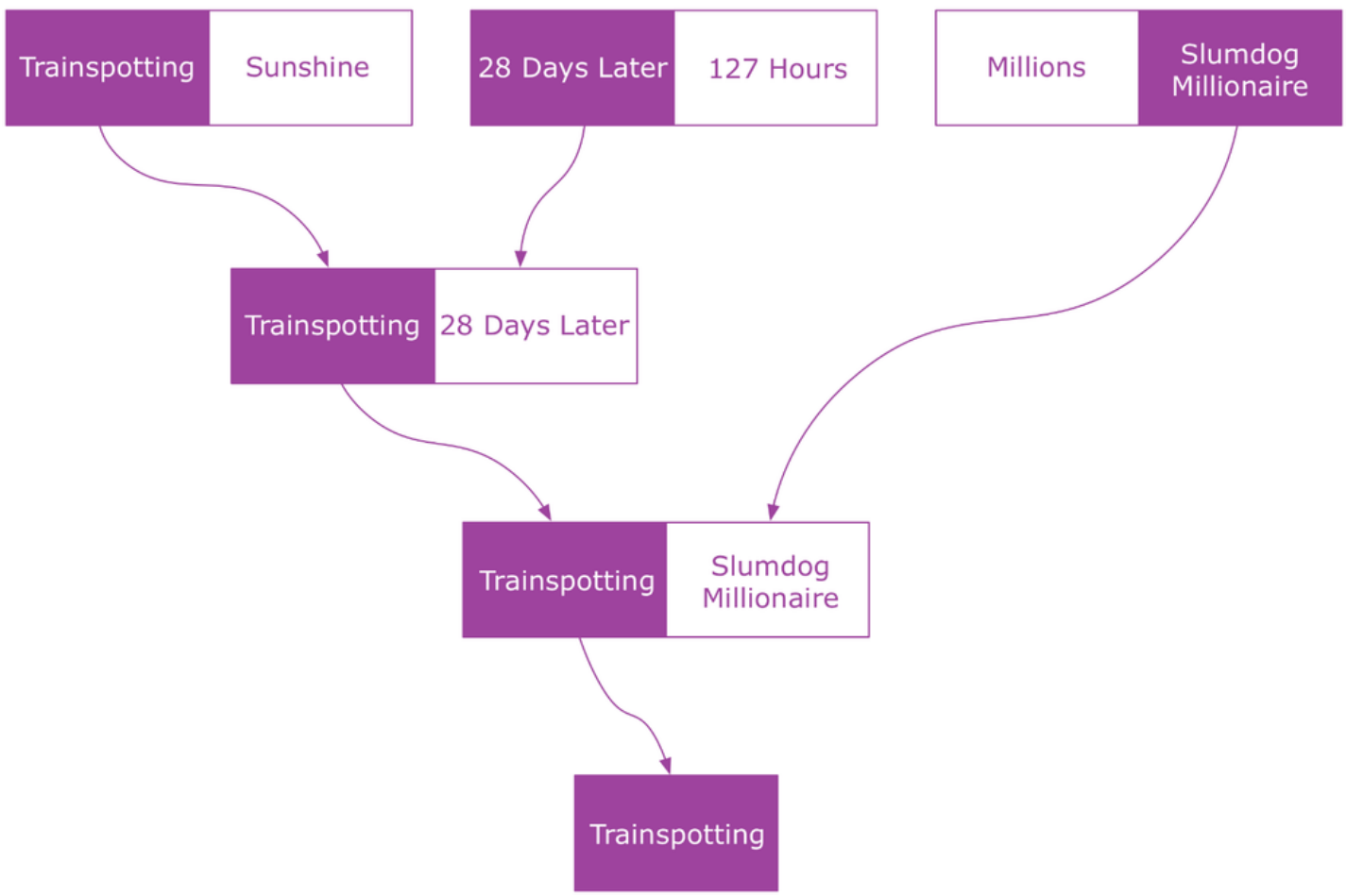
1. Что вам понадобится

Данное руководство будет наиболее полезным для разработчиков, которые уже умеют писать JavaScript-приложения. Как уже упоминалось, мы будем использовать Node, ES6, React, Webpack и Babel, и если вы хотя бы немного знакомы с этими инструментами, никаких проблем с продвижением не будет. Даже если не знакомы, вы сможете понять основы по пути.

В качестве хорошего пособия по разработке веб-приложений с помощью React, Webpack и ES6, можно посоветовать SurviveJS. Что касается инструментов, то вам понадобится Node с NPM и ваш любимый текстовый редактор.

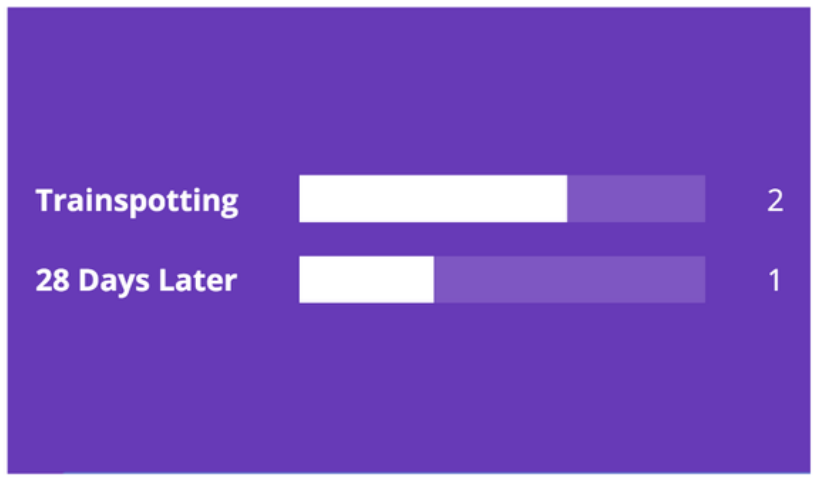
2. Приложение

Мы будем делать приложение для «живых» голосований на вечеринках, конференциях, встречах и прочих собраниях. Идея заключается в том, что пользователю будет предлагаться коллекция позиций для голосования: фильмы, песни, языки программирования, цитаты с Horse JS, и так далее. Приложение будет располагать элементы парами, чтобы каждый мог проголосовать за своего фаворита. В результате серии голосований останется один элемент — победитель. Пример голосования за лучший фильм Дэнни Бойла:



Приложение будет иметь два разных пользовательских интерфейса:

- Интерфейс для голосования можно будет использовать на любом устройстве, где запускается веб-браузер.
- Интерфейс результатов голосования может быть выведен на проектор или какой-то большой экран. Результаты голосования будут обновляться в реальном времени.



3. Архитектура

Структурно система будет состоять из двух приложений:

- Браузерное приложение на React, предоставляющее оба пользовательских интерфейса.

- Серверное приложение на Node, содержащее логику голосования.

Взаимодействие между приложениями будет осуществляться с помощью WebSockets. Redux поможет нам организовать код клиентской и серверной частей. А для хранения состояний будем применять структуры Immutable.

Несмотря на большое сходство клиента и сервера — к примеру, оба будут использовать Redux, — это не универсальное/изоморфное приложение, и приложения не будут совместно использовать какой-либо код. Скорее это можно охарактеризовать как распределённую систему из двух приложений, взаимодействующих друг с другом с помощью передачи сообщений.

4. Серверное приложение

Сначала напишем Node-приложение, а затем — React. Это позволит нам не отвлекаться от реализации базовой логики приложения, прежде чем мы перейдём к интерфейсу. Поскольку мы создаём серверное приложение, будем знакомиться с Redux и Immutable и узнаем, как будет устроено построенное на них приложение. Обычно Redux ассоциируется с React-проектами, но его применение вовсе ими не ограничивается. В частности, мы узнаем, насколько Redux может быть полезен и в других контекстах!

По ходу чтения этого руководства я рекомендую вам писать

приложение с нуля, но можете скачать исходники с GitHub.

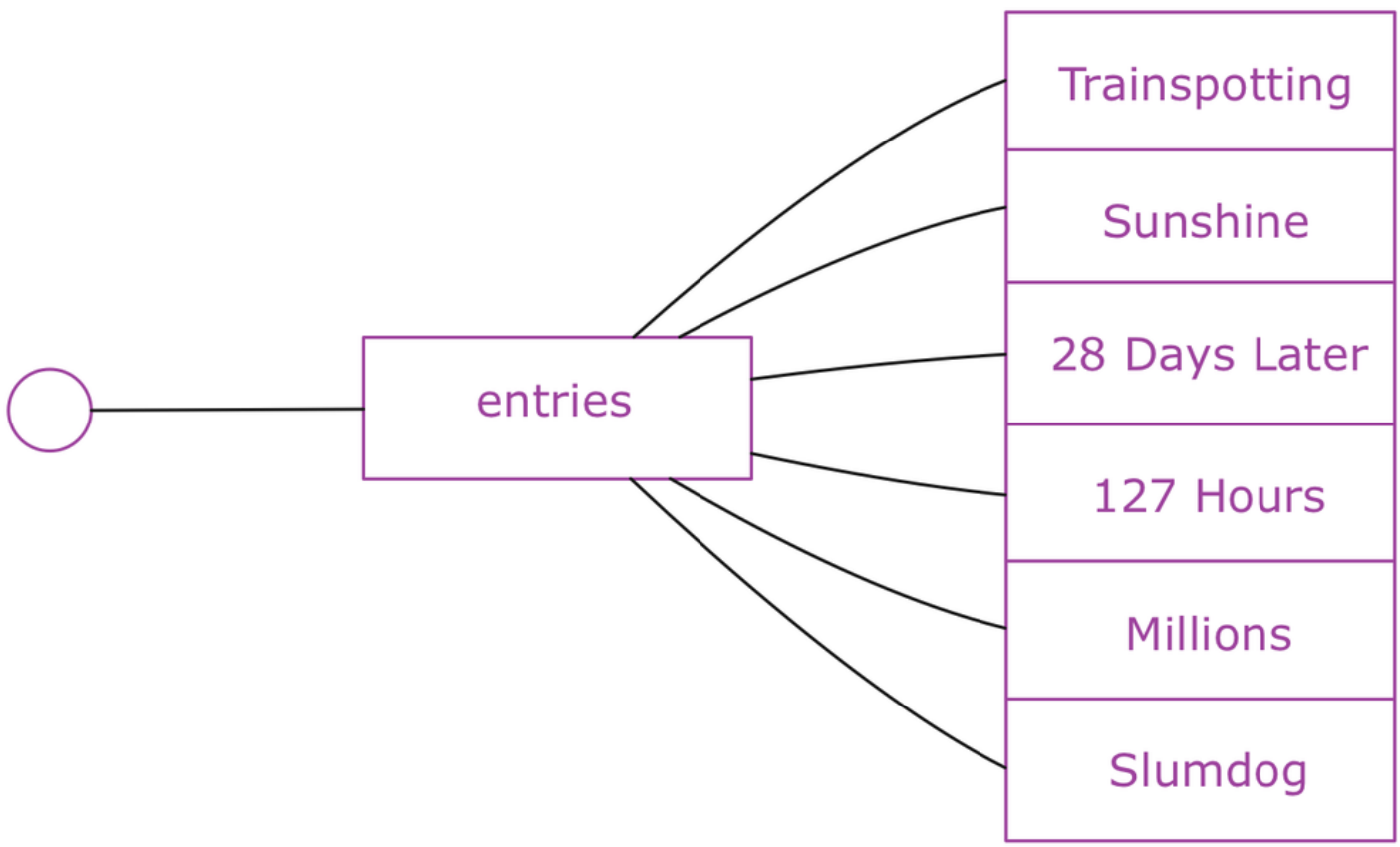
4.1. Разработка дерева состояний приложения

Создание приложения с помощью Redux зачастую начинается с продумывания структуры данных *состояния приложения (application state)*. С её помощью описывается, что происходит в приложении в каждый момент времени. Состояние (state) есть у любого фреймворка и архитектуры. В приложениях на базе Ember и Backbone состояние хранится в моделях (Models). В приложениях на базе Angular состояние чаще всего хранится в фабриках (Factories) и сервисах (Services). В большинстве Flux-приложений состояние является хранилищем (Stores). А как это сделано в Redux?

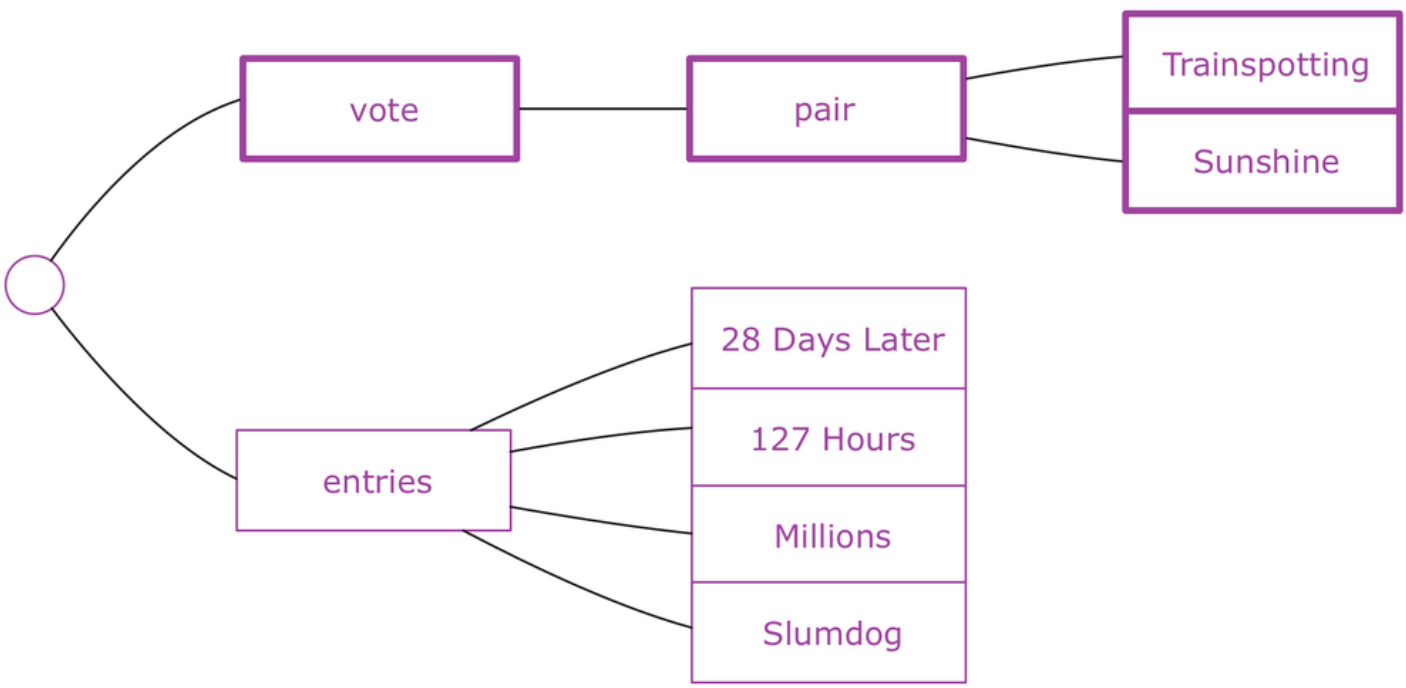
Главное его отличие в том, что все состояния приложения хранятся в единственной древовидной структуре. Таким образом все, что необходимо знать о состоянии приложения, содержится в одной структуре данных из ассоциативных (map) и обычных массивов. Как вы вскоре увидите, у этого решения есть немало последствий. Одним из важнейших является то, что вы можете отделить состояние приложения от его *поведения*. Состояние — это чистые данные. Оно не содержит никаких методов или функций, и оно не упрятано внутрь других объектов. *Всё находится в одном месте.* Это может показаться ограничением, особенно если у вас есть опыт объектно-ориентированного программирования. Но на самом деле это проявление большей свободы, поскольку вы можете сконцентрироваться на одних лишь данных. Очень многое логически вытечет из проектирования

состояний приложения если вы уделите этому достаточно времени.

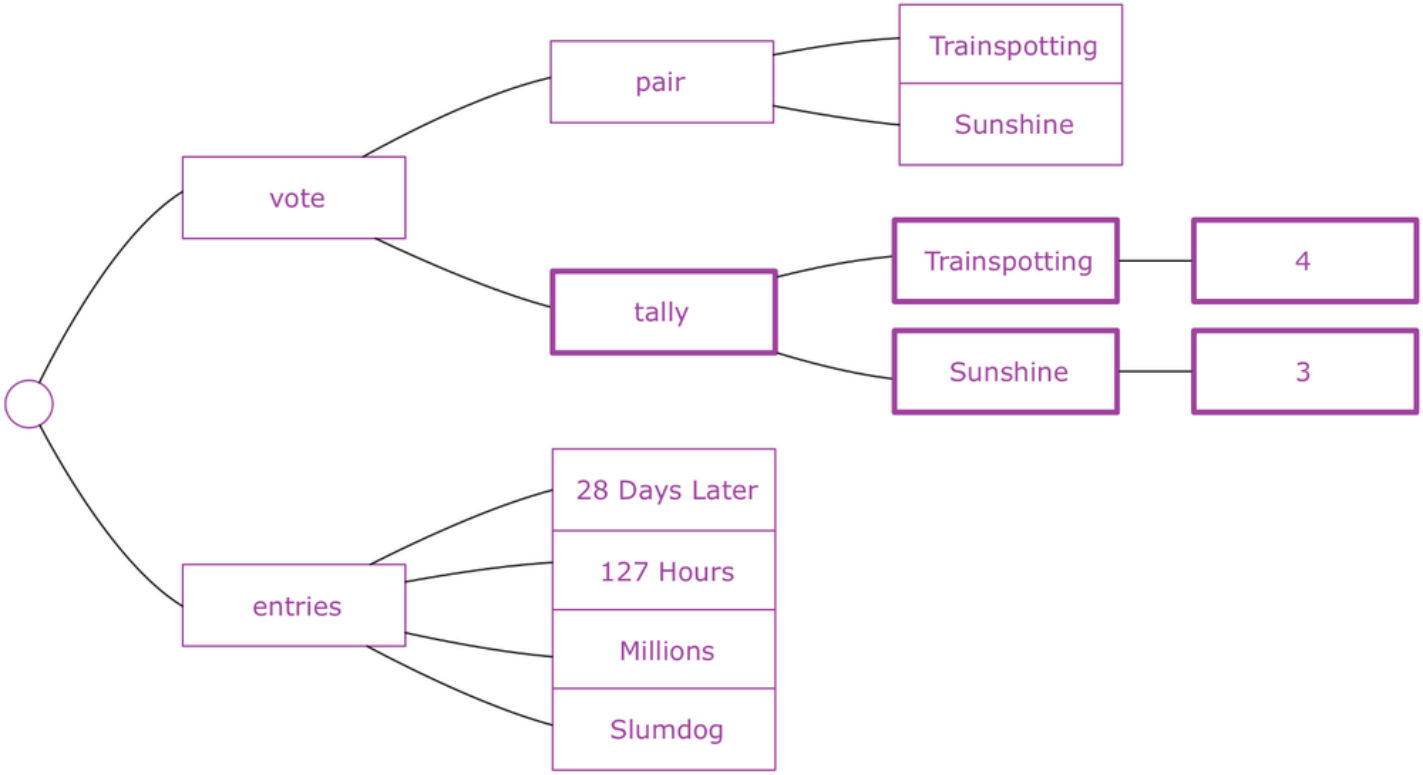
Я не хочу сказать, что вам всегда нужно сначала полностью разрабатывать дерево состояний, а затем создавать остальные компоненты приложения. Обычно это делают параллельно. Но мне кажется, что полезнее сначала в общих чертах представить себе, как должно выглядеть дерево в разных ситуациях, прежде чем приступать к написанию кода. Давайте представим, каким может быть дерево состояний для нашего приложения голосований. Цель приложения — иметь возможность голосовать внутри пар объектов (фильмы, музыкальные группы). В качестве начального состояния приложения целесообразно сделать просто коллекцию из позиций, которые будут участвовать в голосовании. Назовём эту коллекцию *entries (записи)*:



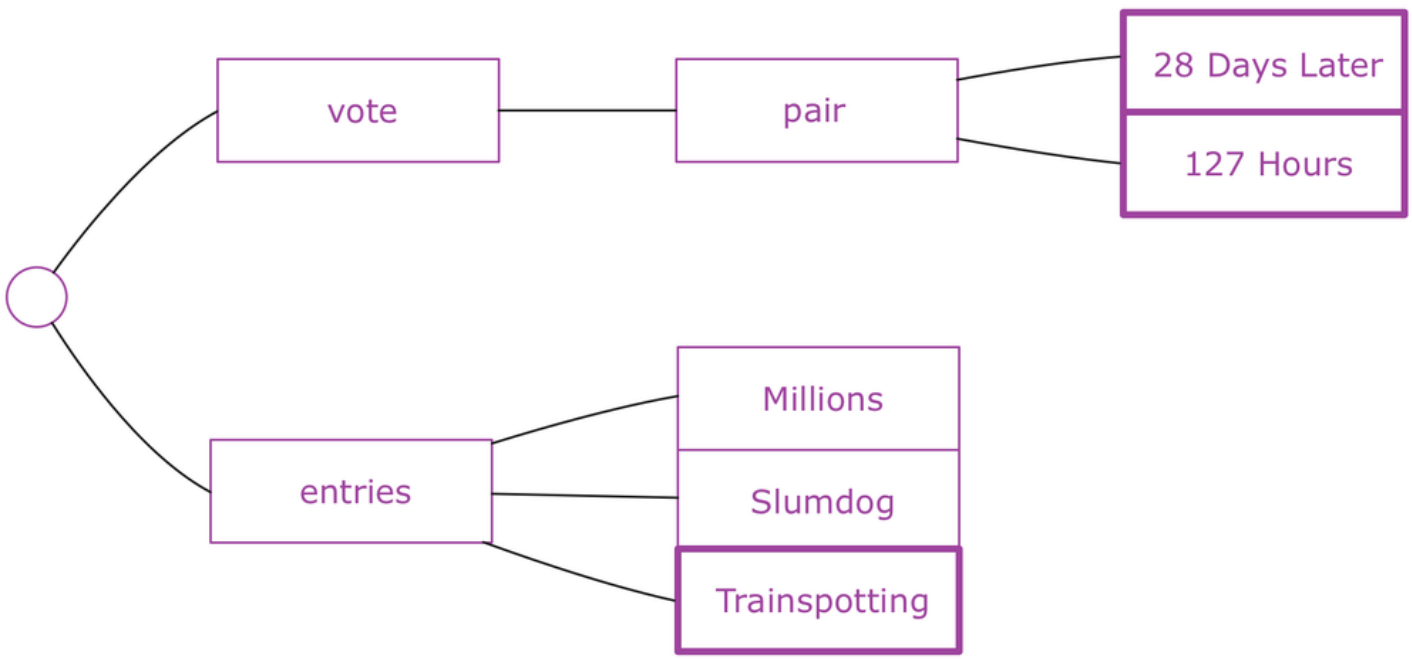
После начала голосования нужно как-то отделить позиции, которые участвуют в голосовании в данный момент. В состоянии может быть сущность *vote*, содержащая пару позиций, из которых пользователь должен выбрать одну. Естественно, эта пара должна быть извлечена из коллекции *entries*:



Также нам нужно вести учёт результатов голосования. Это можно делать с помощью другой структуры внутри *vote*:



По завершении текущего голосования проигравшая запись выкидывается, а победившая возвращается обратно в *entries* и помещается в конец списка. Позднее она снова будет участвовать в голосовании. Затем из списка берётся следующая пара:



Эти состояния циклически сменяют друг друга до тех пор, пока в

коллекции есть записи. В конце останется только одна запись, которая объявляется победителем, а голосование завершается:

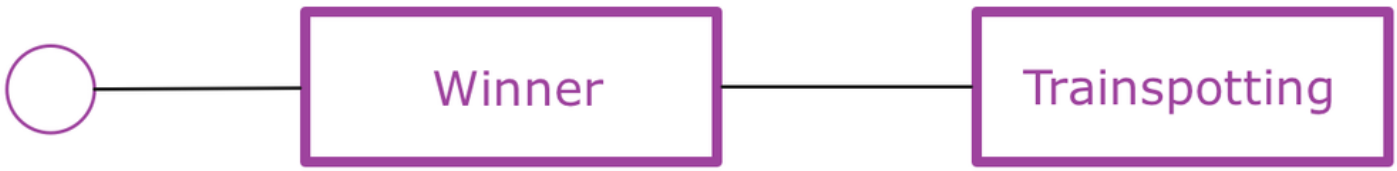


Схема кажется вполне разумной, начнём её реализовывать. Есть много разных способов разработки состояний под эти требования, возможно, этот вариант и не оптимальный. Но это не особенно важно. Начальная схема должна быть просто хорошей для старта. Главное, что у нас есть понимание того, как должно работать наше приложение. И это ещё до того, как мы перешли к написанию кода!

4.2. Настройка проекта

Пришло время засучить рукава. Для начала нужно создать папку проекта, а затем инициализировать его в качестве NPM-проекта:

```
mkdir voting-server
cd voting-server
npm init -y
```

В созданной папке пока что лежит одинокий файл `package.json`. Писать код мы будем в спецификации ES6. Хотя Node начиная с версии 4.0.0 поддерживает много возможностей ES6, необходимые нам модули все же остались за бортом. Поэтому нам нужно добавить в наш проект Babel, чтобы мы могли воспользоваться всей мощью ES6 и транспилировать код в ES5:

```
npm install --save-dev babel-core babel-cli babel-preset-es2015
```

Также нам понадобятся библиотеки для написания unit тестов:

```
npm install --save-dev mocha chai
```

В качестве фреймворка для тестирования будем использовать Mocha. Внутри тестов будем использовать Chai в роли библиотеки для проверки ожидаемого поведения и состояний. Запускать тесты мы будем с помощью команды `mocha`:

```
./node_modules/mocha/bin/mocha --compilers js:babel-core/register --
recursive
```

После этого Mocha будет рекурсивно искать все тесты проекта и запускать их. Для транспилинга ES6-кода перед его запуском будет использоваться Babel. Для удобства можно хранить эту команду в `package.json`:

```
package.json
"scripts": {
  "test": "mocha --compilers js:babel-core/register --recursive"
},
```

Теперь нам нужно включить в Babel поддержку ES6/ES2015. Для этого активируем уже установленный нами пакет `babel-preset-es2015`. Далее просто добавим в `package.json` секцию `"babel"`:

```
package.json
"babel": {
  "presets": ["es2015"]
}
```

Теперь с помощью команды `npm` мы можем запускать наши тесты:

```
npm run test
```

Команда `test:watch` может использоваться для запуска процесса, отслеживающего изменения в нашем коде и запускающего тесты после каждого изменения:

```
package.json
"scripts": {
  "test": "mocha --compilers js:babel-core/register --recursive",
  "test:watch": "npm run test -- --watch"
},
```

Разработанная в Facebook библиотека Immutable предоставляет нам ряд полезных структур данных. Мы обсудим её в следующей главе, а пока просто добавим в проект наряду с библиотекой chai-immutable, которая добавляет в Chai поддержку сравнения Immutable-структур:

```
npm install --save immutable
npm install --save-dev chai-immutable
```

Подключать chai-immutable нужно до запуска каких-либо тестов.

Сделать это можно с помощью файла `test_helper`:

```
test/test_helper.js
import chai from 'chai';
import chaiImmutable from 'chai-immutable';
chai.use(chaiImmutable);
```

Теперь сделаем так, чтобы Mocha подгрузил этот файл до запуска тестов:

```
package.json
"scripts": {
  "test": "mocha --compilers js:babel-core/register --require
./test/test_helper.js  --recursive",
  "test:watch": "npm run test -- --watch"
},
```

Теперь у нас есть все, чтобы начать.

4.3. Знакомство с неизменяемыми данными

Второй важный момент, связанный с архитектурой Redux: состояние — это не просто дерево, а *неизменяемое дерево (immutable tree)*. Структура деревьев из предыдущей главы может навести на мысль, что код должен менять состояние приложения просто обновляя деревья: заменяя элементы в ассоциативных массивах, удаляя их из массивов и т.д. Но в Redux всё делается по-другому. Дерево состояний в Redux-приложении представляет собой *неизменяемую структуру данных (immutable data structure)*. Это значит, что пока дерево существует, оно не меняется. Оно всегда сохраняет одно и то же состояние. И переход к другому

состоянию осуществляется с помощью создания *другого* дерева, в которое внесены необходимые изменения. То есть два следующих друг за другом состояния приложения хранятся в двух отдельных и независимых деревьях. А переключение между деревьями осуществляется с помощью вызова *функции*, принимающей текущее состояние и *возвращающей* следующее.

Хорошая ли это идея? Обычно сразу указывают на то, что если все состояния хранятся в одном дереве и вы вносите все эти безопасные обновления, то можно без особых усилий сохранять историю состояний приложения. Это позволяет реализовать undo/redo “бесплатно” — можно просто задать предыдущее или следующее состояние (дерево) из истории. Также можно сериализовать историю и сохранить её на будущее, или поместить ее в хранилище для последующего проигрывания, что может оказать неоценимую помощью в отладке.

Но мне кажется, что, помимо всех этих дополнительных возможностей, главное достоинство использование неизменяемых данных заключается в упрощении кода. Вам приходится программировать чистые функции: они только принимают и возвращают данные, и больше ничего. Эти функции ведут себя предсказуемо. Вы можете вызывать их сколько угодно раз, и они всегда будут вести себя одинаково. Давайте им одни и те же

аргументы, и будете получать одни и те же результаты. Тестирование становится тривиальным, ведь вам не нужно настраивать заглушки или иные фальшивки, чтобы «подготовить вселенную» к вызову функции. Есть просто входные и выходные данные.

Поскольку мы будем описывать состояние нашего приложения неизменяемыми структурами, давайте потратим немного времени на знакомство с ними, написав несколько unit-тестов, иллюстрирующих работу.

Если же вы уверенно работаете с неизменяемыми данными и библиотекой Immutable, то можете приступить к следующему разделу.

Для ознакомления с идеей неизменяемости можно для начала поговорить о простейшей структуре данных. Допустим, у вас есть приложение-счётчик, состояние которого представляет собой число. Скажем, оно меняется от 0 до 1, потом до 2, потом до 3 и т.д. В принципе, мы уже думаем о числах как о неизменяемых данных. Когда счётчик увеличивается, то число не *изменяется*. Да это и невозможно, ведь у чисел нет «сеттеров». Вы не можете сказать `42.setValue(43)`.

Так что мы просто получаем *другое* число, прибавляя к предыдущему единицу. Это можно сделать с помощью чистой функции. Её аргументом будет текущее состояние, а возвращаемое значение будет использоваться в качестве следующего состояния. Вызываемая функция не меняет текущее

состояние. Вот её пример, а также unit тест к ней:

```
test/immutable_spec.js
import {expect} from 'chai';

describe('immutability', () => {

  describe('a number', () => {

    function increment(currentState) {
      return currentState + 1;
    }

    it('is immutable', () => {
      let state = 42;
      let nextState = increment(state);

      expect(nextState).to.equal(43);
      expect(state).to.equal(42);
    });

  });

});
```

Очевидно, что `state` не меняется при вызове `increment`, ведь числа неизменяемы!

Как вы могли заметить, этот тест ничего не делает с нашим приложением, мы его пока и не писали вовсе.

Тесты могут быть просто инструментом обучения для нас. Я часто нахожу полезным изучать новые API или методики с помощью написания модульных тестов, прогоняющих какие-то идеи. В книге Test-Driven Development подобные тесты получили название «обучающих тестов».

Теперь распространим идею неизменяемости на все виды структур данных, а не только на числа.

С помощью Immutable списков мы можем, к примеру, сделать приложение, чьим состоянием будет список фильмов. Операция добавления нового фильма создаст *новый список, который представляет собой комбинацию старого списка и добавляемой позиции*. Важно отметить, что после этой операции старое состояние *остаётся неизменённым*:

```
test/immutable_spec.js
import {expect} from 'chai';
import {List} from 'immutable';

describe('immutability', () => {

  // ...

  describe('A List', () => {

    function addMovie(currentState, movie) {
      return currentState.push(movie);
    }

    it('is immutable', () => {
      let state = List.of('Trainspotting', '28 Days Later');
      let nextState = addMovie(state, 'Sunshine');

      expect(nextState).to.equal(List.of(
        'Trainspotting',
        '28 Days Later',
        'Sunshine'
      ));
      expect(state).to.equal(List.of(
        'Trainspotting',
        '28 Days Later'
      ));
    });

  });
```

```
});
```

А если бы мы вставили фильм в обычный массив, то старое состояние изменилось бы. Но вместо этого мы используем списки из Immutable, поэтому применяем ту же семантику, что и в предыдущем примере с числами.

При вставке в обычный массив старое состояние изменилось бы. Но поскольку мы используем Immutable списки, то имеем ту же семантику, что и в примере с числами.

Эта идея также хорошо применима и к полноценным деревьям состояний. Дерево является вложенной структурой списков (lists), ассоциативных массивов (maps) и других типов коллекций. Применяемая к нему операция создаёт *новое дерево состояния*, оставляя предыдущее в неприкосновенности. Если дерево представляет собой ассоциативный массив с ключом `movies`, содержащим список фильмов, то добавление новой позиции подразумевает необходимость создания нового массива, в котором ключ `movies` указывает на новый список:

```
test/immutable_spec.js
import {expect} from 'chai';
import {List, Map} from 'immutable';

describe('immutability', () => {

  // ...

  describe('a tree', () => {

    function addMovie(currentState, movie) {
      return currentState.set(
```

```
        'movies',
        currentState.get('movies').push(movie)
      );
    }

    it('is immutable', () => {
      let state = Map({
        movies: List.of('Trainspotting', '28 Days Later')
      });
      let nextState = addMovie(state, 'Sunshine');

      expect(nextState).to.equal(Map({
        movies: List.of(
          'Trainspotting',
          '28 Days Later',
          'Sunshine'
        )
      }));
      expect(state).to.equal(Map({
        movies: List.of(
          'Trainspotting',
          '28 Days Later'
        )
      }));
    });

  });

});
```

Здесь мы видим точно такое же поведение, как и прежде, расширенное для демонстрации работы с вложенными структурами. Идея неизменяемости применима к данным всех форм и размеров.

Для операций над подобными вложенными структурами в Immutable есть несколько вспомогательных функций, облегчающих «залезание» во вложенные данные ради получения обновлённого значения. Для краткости кода можем использовать функцию

update:

```
test/immutable_spec.js
function addMovie(currentState, movie) {
  return currentState.update('movies', movies =>
movies.push(movie));
}
```

Похожую функцию мы будем использовать в нашем приложении для обновления состояния приложения. В API Immutable скрывается немало других возможностей, и мы лишь рассмотрели верхушку айсберга.

Неизменяемые данные являются ключевым аспектом архитектуры Redux, но не существует жесткого требования использовать именно библиотеку Immutable. В официальной документации Redux по больше части упоминаются простые объекты и массивы JavaScript, и от их изменения воздерживаются *по соглашению*.

Существует ряд причин, по которым в нашем же руководстве будет использована библиотека Immutable:

- Структуры данных в Immutable разработаны с нуля, чтобы быть неизменяемыми и предоставляют API, который позволяет удобно выполнять операции над ними.
- Я разделяю точки зрения Рича Хайки, согласно которой не существует такой вещи, как неизменяемость по соглашению. Если вы используете структуры данных, которые могут быть изменены, то рано или поздно кто-нибудь ошибётся и сделает

это. Особенно если вы новичок. Вещи вроде Object.freeze() помогут вам не ошибиться.
- Неизменяемые структуры данных являются персистентными, то есть их внутренняя структура такова, что создание новой версии является эффективной операцией с точки зрения времени и потребления памяти, особенно в случае больших деревьев состояний. Использование обычных объектов и массивов может привести к избыточному копированию, снижающему производительность.

4.4. Реализация логики приложения с помощью чистых функций

Познакомившись с идеей неизменяемых деревьев состояний и функциями, оперирующими этими деревьями, можно перейти к созданию логики нашего приложения. В её основу лягут рассмотренные выше компоненты: древовидная структура и набор функций, создающих новые версии этого дерева.

4.4.1. Загрузка записей

В первую очередь, приложение должно «загружать» коллекцию записей для голосования. Можно сделать функцию `setEntries`, берущую предыдущее состояние и коллекцию, и создающую новое состояние, включив туда записи. Вот тест для этой функции:

```
test/core_spec.js
import {List, Map} from 'immutable';
import {expect} from 'chai';

import {setEntries} from '../src/core';
```

```
describe('application logic', () => {

  describe('setEntries', () => {

    it('добавляет записи к состоянию', () => {
      const state = Map();
      const entries = List.of('Trainspotting', '28 Days Later');
      const nextState = setEntries(state, entries);
      expect(nextState).to.equal(Map({
        entries: List.of('Trainspotting', '28 Days Later')
      }));
    });

  });

});
```

Первоначальная реализация `setEntries` делает только самое простое: ключу `entries` в ассоциативном массиве состояния присваивает в качестве значения указанный список записей. Получаем первое из спроектированных нами ранее деревьев.

```
src/core.js
export function setEntries(state, entries) {
  return state.set('entries', entries);
}
```

Для удобства разрешим входным записям представлять собой обычный JavaScript-массив (или что-нибудь итерируемое). В дереве состояния же должен присутствовать Immutable список (`List`):

```
test/core_spec.js
it('преобразует в immutable', () => {
  const state = Map();
  const entries = ['Trainspotting', '28 Days Later'];
```

```
  const nextState = setEntries(state, entries);
  expect(nextState).to.equal(Map({
    entries: List.of('Trainspotting', '28 Days Later')
  }));
});
```

Для удовлетворения этому требованию будем передавать записи в конструктор списка:

```
src/core.js
import {List} from 'immutable';

export function setEntries(state, entries) {
  return state.set('entries', List(entries));
}
```

4.4.2. Запуск голосования

Голосование можно запустить вызовом функции `next` при состоянии, уже имеющем набор записей. Таким образом будет осуществлён переход от первого ко второму из спроектированных деревьев.

Этой функции не нужны дополнительные аргументы. Она должна создавать ассоциативный массив `vote`, в котором по ключу `pair` лежат две первые записи. При этом записи, которые в данный момент участвуют в голосовании, больше не должны находиться в списке `entries`:

```
test/core_spec.js
import {List, Map} from 'immutable';
import {expect} from 'chai';
import {setEntries, next} from '../src/core';
```

```
describe('логика приложения', () => {

  // ..

  describe('далее', () => {

    it('берёт для голосования следующие две записи', () => {
      const state = Map({
        entries: List.of('Trainspotting', '28 Days Later',
'Sunshine')
      });
      const nextState = next(state);
      expect(nextState).to.equal(Map({
        vote: Map({
          pair: List.of('Trainspotting', '28 Days Later')
        }),
        entries: List.of('Sunshine')
      }));
    });

  });

});
```

Реализация функции будет объединять (merge) обновление со старым состоянием, обособляя первые записи лежат в отдельный список, а остальные — в новую версию списка `entries`:

```
src/core.js
import {List, Map} from 'immutable';

// ...

export function next(state) {
  const entries = state.get('entries');
  return state.merge({
    vote: Map({pair: entries.take(2)}),
    entries: entries.skip(2)
  });
}
```

4.4.3. Голосование

По мере продолжения голосования, пользователь должен иметь возможность отдавать голос за разные записи. И при каждом новом голосовании на экране должен отображаться текущий результат. Если за конкретную запись уже голосовали, то её счётчик должен увеличиться.

```
test/core_spec.js
import {List, Map} from 'immutable';
import {expect} from 'chai';
import {setEntries, next, vote} from '../src/core';

describe('логика приложения', () => {

  // ...

  describe('vote', () => {

    it('создаёт результат голосования для выбранной записи', () => {
      const state = Map({
        vote: Map({
          pair: List.of('Trainspotting', '28 Days Later')
        }),
        entries: List()
      });
      const nextState = vote(state, 'Trainspotting');
      expect(nextState).to.equal(Map({
        vote: Map({
          pair: List.of('Trainspotting', '28 Days Later'),
          tally: Map({
            'Trainspotting': 1
          })
        }),
        entries: List()
      }));
    });

    it('добавляет в уже имеющийся результат для выбранной записи',
() => {
      const state = Map({
        vote: Map({
```

```
          pair: List.of('Trainspotting', '28 Days Later'),
          tally: Map({
            'Trainspotting': 3,
            '28 Days Later': 2
          })
        }),
        entries: List()
      });
      const nextState = vote(state, 'Trainspotting');
      expect(nextState).to.equal(Map({
        vote: Map({
          pair: List.of('Trainspotting', '28 Days Later'),
          tally: Map({
            'Trainspotting': 4,
            '28 Days Later': 2
          })
        }),
        entries: List()
      }));
    });

  });

});
```

С помощью функции fromJS из Immutable можно более лаконично создать все эти вложенные схемы и списки.

Прогоним тесты:

```
src/core.js
export function vote(state, entry) {
  return state.updateIn(
    ['vote', 'tally', entry],
    0,
    tally => tally + 1
  );
}
```

Использование updateIn позволяет не растекаться мыслью по

древу. В этом коде говорится: «возьми путь вложенной структуры данных ['vote', 'tally', 'Trainspotting'] и примени эту функцию. Если какие-то ключи отсутствуют, то создай вместо них новые массивы (Map). Если в конце отсутствует значение, то инициализируй нулем». Именно такого рода код позволяет получать удовольствие от работы с неизменяемыми структурами данных, так что стоит уделить этому время и попрактиковаться.

4.4.4. Переход к следующей паре

По окончании голосования по текущей паре, переходим к следующей. Нужно сохранить победителя и добавить в конец списка записей, чтобы позднее он снова принял участие в голосовании. Проигравшая запись просто выкидывается. В случае ничьей сохраняются обе записи.

Добавим эту логику к имеющейся реализации next:

```
test/core_spec.js
describe('next', () => {

  // ...

  it('помещает победителя текущего голосования в конец списка
записей', () => {
    const state = Map({
      vote: Map({
        pair: List.of('Trainspotting', '28 Days Later'),
        tally: Map({
          'Trainspotting': 4,
          '28 Days Later': 2
        })
      }),
      entries: List.of('Sunshine', 'Millions', '127 Hours')
    });
    const nextState = next(state);
```

```
    expect(nextState).to.equal(Map({
      vote: Map({
        pair: List.of('Sunshine', 'Millions')
      }),
      entries: List.of('127 Hours', 'Trainspotting')
    }));
  });

  it('в случае ничьей помещает обе записи в конец списка', () => {
    const state = Map({
      vote: Map({
        pair: List.of('Trainspotting', '28 Days Later'),
        tally: Map({
          'Trainspotting': 3,
          '28 Days Later': 3
        })
      }),
      entries: List.of('Sunshine', 'Millions', '127 Hours')
    });
    const nextState = next(state);
    expect(nextState).to.equal(Map({
      vote: Map({
        pair: List.of('Sunshine', 'Millions')
      }),
      entries: List.of('127 Hours', 'Trainspotting', '28 Days
Later')
    }));
  });

});
```

В нашей реализации мы просто соединяем победителей текущего голосования с записями. А находить этих победителей можно с помощью новой функции `getWinners`:

```
src/core.js
function getWinners(vote) {
  if (!vote) return [];
  const [a, b] = vote.get('pair');
  const aVotes = vote.getIn(['tally', a], 0);
  const bVotes = vote.getIn(['tally', b], 0);
  if      (aVotes > bVotes)  return [a];
  else if (aVotes < bVotes)  return [b];
```

```
  else                        return [a, b];
}

export function next(state) {
  const entries = state.get('entries')
                       .concat(getWinners(state.get('vote')));
  return state.merge({
    vote: Map({pair: entries.take(2)}),
    entries: entries.skip(2)
  });
}
```

4.4.5. Завершение голосования

В какой-то момент у нас остаётся лишь одна запись — победитель, и тогда голосование завершается. И вместо формирования нового голосования, мы явным образом назначаем эту запись победителем в текущем состоянии. Конец голосования.

```
test/core_spec.js
describe('next', () => {

  // ...

  it('когда остаётся лишь одна запись, помечает её как победителя',
() => {
    const state = Map({
      vote: Map({
        pair: List.of('Trainspotting', '28 Days Later'),
        tally: Map({
          'Trainspotting': 4,
          '28 Days Later': 2
        })
      }),
      entries: List()
    });
    const nextState = next(state);
    expect(nextState).to.equal(Map({
      winner: 'Trainspotting'
    }));
  });
```

```
});
```

В реализации `next` нужно предусмотреть обработку ситуации, когда после завершения очередного голосования в списке записей остаётся лишь одна позиция:

```
src/core.js
export function next(state) {
  const entries = state.get('entries')
                       .concat(getWinners(state.get('vote')));
  if (entries.size === 1) {
    return state.remove('vote')
                .remove('entries')
                .set('winner', entries.first());
  } else {
    return state.merge({
      vote: Map({pair: entries.take(2)}),
      entries: entries.skip(2)
    });
  }
}
```

Здесь можно было бы просто вернуть `Map({winner: entries.first()})`. Но вместо этого мы снова берём старое состояние и явным образом убираем из него ключи `vote` и `entries`. Это делается с прицелом на будущее: может случиться так, что в нашем состоянии появятся какие-то сторонние данные, которые нужно будет в неизменном виде передать с помощью этой функции. В целом, в основе функций трансформирования состояний лежит хорошая идея — всегда преобразовывать старое состояние в новое, вместо создания нового состояния с нуля.

Теперь у нас есть вполне приемлемая версия основной логики

нашего приложения, выраженная в виде нескольких функций. Также мы написали для них unit тесты, которые дались нам довольно легко: никаких преднастроек и заглушек. В этом и проявляется красота чистых функций. Можно просто вызвать их и проверить возвращаемые значения.

Обратите внимание, что мы пока ещё даже не установили Redux. При этом спокойно занимались разработкой логики приложения, не привлекая «фреймворк» к этой задаче. Есть в этом что-то чертовски приятное.

4.5. Использование Actions и Reducers

Итак, у нас есть основные функции, но мы не будем вызывать их в Redux напрямую. Между функциями и внешним миром расположен слой косвенной адресации: действия (`Actions`).

Это простые структуры данных, описывающие изменения, которые должны произойти с состоянием вашего приложения. По сути это описание вызова функции, упакованное в маленький объект. По соглашению, каждое действие имеет атрибут `type`, описывающий, для какой операции это действие предназначено. Также могут использоваться и дополнительные атрибуты. Вот несколько примеров действий, подходящих для наших основных функций:

```
{type: 'SET_ENTRIES', entries: ['Trainspotting', '28 Days Later']}

{type: 'NEXT'}

{type: 'VOTE', entry: 'Trainspotting'}
```

При таком способе выражения нам ещё понадобится превратить их в нормальные вызовы основных функций. В случае с `VOTE` должен выполняться следующий вызов:

```
// Этот action
let voteAction = {type: 'VOTE', entry: 'Trainspotting'}
// должен сделать это:
return vote(state, voteAction.entry);
```

Теперь нужно написать шаблонную функцию (generic function), принимающую любое действие — в рамках текущего состояния — и вызывающую соответствующую функцию ядра. Такая функция называется *преобразователем* (`reducer`):

```
src/reducer.js
export default function reducer(state, action) {
  // Определяет, какую функцию нужно вызвать, и делает это
}
```

Теперь нужно убедиться, что наш reducer способен обрабатывать каждое из трёх действий:

```
test/reducer_spec.js
import {Map, fromJS} from 'immutable';
import {expect} from 'chai';

import reducer from '../src/reducer';

describe('reducer', () => {

  it('handles SET_ENTRIES', () => {
    const initialState = Map();
    const action = {type: 'SET_ENTRIES', entries:
['Trainspotting']};
```

```
    const nextState = reducer(initialState, action);

    expect(nextState).to.equal(fromJS({
      entries: ['Trainspotting']
    }));
  });

  it('handles NEXT', () => {
    const initialState = fromJS({
      entries: ['Trainspotting', '28 Days Later']
    });
    const action = {type: 'NEXT'};
    const nextState = reducer(initialState, action);

    expect(nextState).to.equal(fromJS({
      vote: {
        pair: ['Trainspotting', '28 Days Later']
      },
      entries: []
    }));
  });

  it('handles VOTE', () => {
    const initialState = fromJS({
      vote: {
        pair: ['Trainspotting', '28 Days Later']
      },
      entries: []
    });
    const action = {type: 'VOTE', entry: 'Trainspotting'};
    const nextState = reducer(initialState, action);

    expect(nextState).to.equal(fromJS({
      vote: {
        pair: ['Trainspotting', '28 Days Later'],
        tally: {Trainspotting: 1}
      },
      entries: []
    }));
  });

});
```

В зависимости от типа действия reducer должен обращаться к одной из функций ядра. Он также должен знать, как извлечь из

действия дополнительные аргументы для каждой из функций:

```
src/reducer.js
import {setEntries, next, vote} from './core';

export default function reducer(state, action) {
  switch (action.type) {
  case 'SET_ENTRIES':
    return setEntries(state, action.entries);
  case 'NEXT':
    return next(state);
  case 'VOTE':
    return vote(state, action.entry)
  }
  return state;
}
```

Обратите внимание, что если reducer не распознает действие, то просто вернёт текущее состояние.

К reducer-ам предъявляется важное дополнительное требование: если они вызываются с незаданным состоянием, то должны знать, как проинициализировать его правильным значением. В нашем случае исходным значением является ассоциативный массив. Таким образом, состояние `undefined` должно обрабатываться, как если бы мы передали пустой массив:

```
test/reducer_spec.js
describe('reducer', () => {

  // ...

  it('has an initial state', () => {
    const action = {type: 'SET_ENTRIES', entries:
['Trainspotting']};
    const nextState = reducer(undefined, action);
    expect(nextState).to.equal(fromJS({
      entries: ['Trainspotting']
```

```
    }));
  });

});
```

Поскольку логика нашего приложения расположена в `core.js`, то здесь же можно объявить начальное состояние:

```
src/core.js
export const INITIAL_STATE = Map();
```

Затем мы импортируем его в reducer-е и используем в качестве значения по умолчанию для аргумента состояния:

```
src/reducer.js
import {setEntries, next, vote, INITIAL_STATE} from './core';

export default function reducer(state = INITIAL_STATE, action) {
  switch (action.type) {
  case 'SET_ENTRIES':
    return setEntries(state, action.entries);
  case 'NEXT':
    return next(state);
  case 'VOTE':
    return vote(state, action.entry)
  }
  return state;
}
```

Любопытно то, как абстрактно reducer можно использовать для перевода приложения из одного состояния в другое при помощи действия любого типа. В принципе, взяв коллекцию прошлых действий, вы действительно можете просто преобразовать её в

текущее состояние. Именно поэтому функция называется *преобразователь*: она заменяет собой вызов callback-а.

```
test/reducer_spec.js
it('может использоваться с reduce', () => {
  const actions = [
    {type: 'SET_ENTRIES', entries: ['Trainspotting', '28 Days
Later']},
    {type: 'NEXT'},
    {type: 'VOTE', entry: 'Trainspotting'},
    {type: 'VOTE', entry: '28 Days Later'},
    {type: 'VOTE', entry: 'Trainspotting'},
    {type: 'NEXT'}
  ];
  const finalState = actions.reduce(reducer, Map());

  expect(finalState).to.equal(fromJS({
    winner: 'Trainspotting'
  }));
});
```

Способность *создавать и/или проигрывать* коллекции действий является главным преимуществом модели переходов состояний с помощью action/reducer, по сравнению с прямым вызовом функций ядра. Поскольку actions — это объекты, которые можно сериализовать в JSON, то вы, к примеру, можете легко отправлять их в Web Worker, и там уже выполнять логику reducer-а. Или даже можете отправлять их по сети, как мы это сделаем ниже.

Обратите внимание, что в качестве actions мы используем простые объекты, а не структуры данных из Immutable. Этого требует от нас Redux.

4.6. Привкус Reducer-композиции

Согласно логике нашего ядра, каждая функция принимает и возвращает полное состояние приложения.

Но можно легко заметить, что в больших приложениях этот подход может оказаться не лучшим решением. Если каждая операция в приложении должна знать о структуре всего состояния, то ситуация быстро может стать нестабильной. Ведь для изменения состояния потребуется внести кучу других изменений.

Лучше всего в любых возможных случаях выполнять операции в рамках как можно меньшей части состояния (или в *поддереве*). Речь идёт о модульности: функциональность работает только с какой-то одной частью данных, словно остальное и не существует.

Но в нашем случае приложение такое маленькое, что у нас не возникнет вышеописанных проблем. Хотя кое-что улучшить мы всё же можем: функции `vote` можно не передавать всё состояние приложения, ведь она работает только с одноимённым сегментом `vote`. И только о нём ей достаточно знать. Для отображения этой идеи мы можем модифицировать наши unit тесты для `vote`:

```
test/core_spec.js
describe('vote', () => {

  it('создаёт результат голосования для выбранной записи', () => {
    const state = Map({
      pair: List.of('Trainspotting', '28 Days Later')
    });
    const nextState = vote(state, 'Trainspotting')
    expect(nextState).to.equal(Map({
      pair: List.of('Trainspotting', '28 Days Later'),
      tally: Map({
```

```
        'Trainspotting': 1
      })
    }));
  });

  it('добавляет в уже имеющийся результат для выбранной записи', ()
=> {
    const state = Map({
      pair: List.of('Trainspotting', '28 Days Later'),
      tally: Map({
        'Trainspotting': 3,
        '28 Days Later': 2
      })
    });
    const nextState = vote(state, 'Trainspotting');
    expect(nextState).to.equal(Map({
      pair: List.of('Trainspotting', '28 Days Later'),
      tally: Map({
        'Trainspotting': 4,
        '28 Days Later': 2
      })
    }));
  });

});
```

Как видите, код теста упростился, а это обычно хороший знак!

Теперь реализация `vote` должна просто брать соответствующий сегмент состояния и обновлять счетчик голосования:

```
src/core.js
export function vote(voteState, entry) {
  return voteState.updateIn(
    ['tally', entry],
    0,
    tally => tally + 1
  );
}
```

Далее reducer должен взять состояние и передать функции `vote`

только необходимую часть.

```
src/reducer.js
export default function reducer(state = INITIAL_STATE, action) {
  switch (action.type) {
  case 'SET_ENTRIES':
    return setEntries(state, action.entries);
  case 'NEXT':
    return next(state);
  case 'VOTE':
    return state.update('vote',
                        voteState => vote(voteState, action.entry));
  }
  return state;
}
```

Это лишь небольшой пример подхода, важность которого сильно возрастает с увеличением размера приложения: главная функция-reducer просто передаёт отдельные сегменты состояния reducer-ам уровнем ниже. Мы отделяем задачу поиска нужного сегмента дерева состояний от применения обновления к этому сегменту.

Гораздо подробнее шаблоны *reducer-композиции* рассмотрены в соответствующей секции документации Redux. Также там объясняются некоторые вспомогательные функции, во многих случаях облегчающие использование reducer-композиции.

4.7. Использование Redux Store

Теперь, когда у нас есть reducer, можно начать думать, как всё это подключить к Redux.

Как мы только что видели, если у вас есть коллекция всех

действий, которые когда либо будут иметь место в вашем приложении, что вы можете просто вызвать `reduce` и получить на выходе финальное состояние приложения. Конечно, обычно у вас нет такой коллекции. Действия осуществляются постепенно, по мере возникновения разных событий: когда пользователь взаимодействует с приложением, когда данные приходят из сети, по триггеру таймаута.

Приспособиться к ситуации помогает хранилище — *Redux Store*. Как подсказывает логика, это объект, в котором хранится состояние нашего приложения.

Хранилище инициализируется reducer-функцией, наподобие уже реализованной нами:

```
import {createStore} from 'redux';

const store = createStore(reducer);
```

Далее можно *передать (dispatch)* действия в store, который затем воспользуется reducer-ом для применения этих действий к текущему состоянию. В качестве результата этой процедуры мы получим *следующее* состояние, которое будет находиться в Redux-Store.

```
store.dispatch({type: 'NEXT'});
```

Вы можете получить из хранилища текущее состояние в любой

момент времени:

```
store.getState();
```

Давайте настроим и экспортируем Redux Store в файл `store.js`. Но сначала протестируем: нам нужно создать хранилище, считать его начальное состояние, передать action и наблюдать изменённое состояние:

```
test/store_spec.js
import {Map, fromJS} from 'immutable';
import {expect} from 'chai';

import makeStore from '../src/store';

describe('store', () => {

  it('хранилище сконфигурировано с помощью правильного
преобразователя', () => {
    const store = makeStore();
    expect(store.getState()).to.equal(Map());

    store.dispatch({
      type: 'SET_ENTRIES',
      entries: ['Trainspotting', '28 Days Later']
    });
    expect(store.getState()).to.equal(fromJS({
      entries: ['Trainspotting', '28 Days Later']
    }));
  });

});
```

Перед созданием Store нам нужно добавить Redux в проект:

```
npm install --save redux
```

Теперь можно создавать `store.js`, в котором вызовем `createStore` с нашим reducer-ом:

```
src/store.js
import {createStore} from 'redux';
import reducer from './reducer';

export default function makeStore() {
  return createStore(reducer);
}
```

Итак, Redux Store соединяет части нашего приложения в целое, которое можно использовать как центральную точку — здесь находится текущее состояние, сюда приходят actions, которые переводят приложение из одного состояния в другое с помощью логики ядра, транслируемой через reducer.

Вопрос: Сколько переменных в Redux-приложении вам нужно?
Ответ: Одна. Внутри хранилища.

На первый взгляд это звучит странно. По крайне мере, если у вас не так много опыта в функциональном программировании. Как можно сделать хоть что-то полезное всего лишь с одной переменной?

Но больше нам и *не нужно*. Текущее дерево состояний — единственная вещь, которая изменяется со временем в нашем базовом приложении. Всё остальное — это константы и неизменяемые значения.

Примечательно, насколько мала площадь соприкосновения между кодом нашего приложения и Redux. Благодаря тому, что у нас есть шаблонная reducer-функция, нам достаточно уведомить Redux лишь о ней. А всё остальное есть в нашем собственном, не зависящем от фреймворка, портируемом и исключительно функциональном коде!

Если мы теперь создадим входную точку нашего приложения — `index.js`, то сможем создать и экспортировать Store:

```
index.js
import makeStore from './src/store';

export const store = makeStore();
```

А раз уж мы его экспортировали, то можем теперь завести и Node REPL (например, с помощью `babel-node`), запросить файл `index.js` и взаимодействовать с приложением с помощью Store.

4.8. Настройка сервера Socket.io

Наше приложение будет работать в качестве сервера для другого браузерного приложения, имеющего пользовательский интерфейс для голосования и просмотра результатов. Нам нужно организовать взаимодействие клиентов с сервером, и наоборот.

Наше приложение только выиграет от внедрения общения в реальном времени, поскольку пользователям понравится сразу же наблюдать результаты своих действий и действий других. Для этой цели давайте воспользуемся WebSocket'ами. Точнее, возьмём

библиотеку Socket.io, предоставляющую хорошую абстракцию для работающих в браузерах WebSocket'ов. К тому же тут есть и несколько запасных механизмов для клиентов, не поддерживающих WebSocket'ы.

Добавляем Socket.io в проект:

```
npm install --save socket.io
```

Создаём файл `server.js`, экспортирующий функцию создания сервера Socket.io:

```
src/server.js
import Server from 'socket.io';

export default function startServer() {
  const io = new Server().attach(8090);
}
```

Этот код создает сервер Socket.io, а также поднимает на порте 8090 обычный HTTP-сервер. Порт выбран произвольно, он должен совпадать с портом, который позднее будет использоваться для связи с клиентами.

Теперь вызовем эту функцию из `index.js`, и сервер будет запущен с началом работы приложения:

```
index.js
import makeStore from './src/store';
import startServer from './src/server';
```

```
export const store = makeStore();
startServer();
```

Можно немного упростить процедуру запуска, добавив команду `start` в наш `package.json`:

```
package.json
"scripts": {
  "start": "babel-node index.js",
  "test": "mocha --compilers js:babel-core/register  --require
./test/test_helper.js  --recursive",
  "test:watch": "npm run test -- --watch"
},
```

Теперь после ввода следующей команды будет запускаться сервер и создаваться Redux-Store:

```
npm run start
```

Команда `babel-node` взята из ранее установленного нами пакета babel-cli. Она позволяет легко запускать Node-код с включённой поддержкой Babel-транспилирования. В целом, это не рекомендуется делать для боевых серверов, потому что производительность несколько снижается. Но зато хорошо подходит для наших учебных задач.

4.9. Трансляция Store из Redux Listener

Теперь у нас есть сервер Socket.io и контейнер Redux состояния, но они пока никак не интегрированы. Изменим это.

Сервер должен сообщать клиентам о текущем состоянии приложения (например, «за что сейчас голосуем?», «каков текущий результат?», «есть ли уже победитель?»). Это можно делать при каждом изменении с помощью передачи события из Socket.io всем подключённым клиентам.

А как узнать, что что-то изменилось? Для этого можно *подписаться* на Redux store, предоставив функцию, которая будет вызываться хранилищем при каждом применении action, когда состояние потенциально изменилось. По сути, это callback на изменения состояния внутри store.

Мы будем делать это в `startServer`, так что предоставим ему Redux store для начала:

```
index.js
import makeStore from './src/store';
import {startServer} from './src/server';

export const store = makeStore();
startServer(store);
```

Подпишем получателя событий (listener) на наше хранилище. Он считывает текущее состояние, превращает его в простой JavaScript-объект и передаёт его на сервер Socket.io в виде события `state`. В результате мы получаем JSON-сериализованный снэпшот состояния, рассылаемый на все активные подключения Socket.io.

```
src/server.js
import Server from 'socket.io';
```

```
export function startServer(store) {
  const io = new Server().attach(8090);

  store.subscribe(
    () => io.emit('state', store.getState().toJS())
  );
}
```

Теперь при каждом изменении мы передаём *полное* состояние всем клиентам. Но это может повлечь за собой серьёзный рост трафика. Можно предложить различные способы оптимизации (например, отправлять только актуальную часть состояния, отправлять дифы вместо снэпшотов, и т.д.). В нашей реализации не будем этого делать в целях сохранения простоты кода.

Помимо передачи снэпшота состояния было бы хорошо, если бы клиенты *немедленно* получали текущее состояние при подключении к серверу. Это позволит сразу синхронизировать состояние клиентских приложений с текущим состоянием сервера.

На сервере Socket.io мы можем слушать события `connection`, передаваемые клиентами при каждом подключении. В обработчике события мы можем сразу отдавать текущее состояние:

```
src/server.js
import Server from 'socket.io';

export function startServer(store) {
  const io = new Server().attach(8090);

  store.subscribe(
    () => io.emit('state', store.getState().toJS())
  );
```

```
  io.on('connection', (socket) => {
    socket.emit('state', store.getState().toJS());
  });

}
```

4.10. Получение Remote Redux Actions

Вдобавок к передаче клиентам состояния приложения, нам нужно уметь получать от них обновления: пользователи будут голосовать, а модуль управления голосованием будет обрабатывать события с помощью действия `NEXT`. Для этого достаточно напрямую скармливать в Redux store события `action`, генерируемые клиентами.

```
src/server.js
import Server from 'socket.io';

export function startServer(store) {
  const io = new Server().attach(8090);

  store.subscribe(
    () => io.emit('state', store.getState().toJS())
  );

  io.on('connection', (socket) => {
    socket.emit('state', store.getState().toJS());
    socket.on('action', store.dispatch.bind(store));
  });

}
```

Здесь мы уже выходим за рамки «стандартного Redux», потому что фактически принимаем в store удалённые (remote) actions. Но архитектура Redux вовсе не мешает нам: действия являются

JavaScript-объектами, которые можно легко посылать по сети, поэтому мы сразу получаем систему, в которой принимать участие в голосовании может любое количество клиентов. А это большой шаг!

Конечно, с точки зрения безопасности здесь есть ряд моментов, ведь мы позволяем любому клиенту, подключившемуся к Socket.io, отправлять любое действие в Redux store. Поэтому в реальных проектах нужно использовать что-то вроде файрвола, наподобие Vert.x Event Bus Bridge. Также файрвол нужно внедрять в приложения с механизмом аутентификации.

Теперь наш сервер работает следующим образом:

1. Клиент отправляет на сервер какое-то действие (action).
2. Сервер пересылает его в Redux store.
3. Store вызывает reducer, который исполняет логику, связанную с этим action.
4. Store обновляет состояние на основании возвращаемого reducer-ом значения.
5. Store исполняет соответствующий listener, подписанный сервером.
6. Сервер генерирует событие `state`.
7. Все подключённые клиенты — включая того, кто инициировал первоначальное действие — получают новое состояние.

Прежде, чем мы закончим работу над сервером, давайте загрузим в него тестовый набор записей, чтобы посмотреть, как работает система. Записи можно поместить в файл `entries.json`. Пусть

это будет список фильмов Дэнни Бойла.

```
entries.json
[
  "Shallow Grave",
  "Trainspotting",
  "A Life Less Ordinary",
  "The Beach",
  "28 Days Later",
  "Millions",
  "Sunshine",
  "Slumdog Millionaire",
  "127 Hours",
  "Trance",
  "Steve Jobs"
]
```

Далее просто загружаем список в `index.js`, а затем запускаем голосование с помощью действия `NEXT`:

```
index.js
import makeStore from './src/store';
import {startServer} from './src/server';

export const store = makeStore();
startServer(store);

store.dispatch({
  type: 'SET_ENTRIES',
  entries: require('./entries.json')
});
store.dispatch({type: 'NEXT'});
```

Теперь можно перейти к клиентскому приложению.

5. Клиентское приложение

Далее мы будем писать React-приложение, которое подключается к серверу и позволяет пользователям голосовать. И здесь мы тоже воспользуемся Redux. Собственно, это одно из наиболее распространённых его применений: в качестве движка в основании React-приложений. Мы уже познакомились с его работой, и скоро узнаем, как он совмещается с React и какое оказывает влияние на архитектуру. Рекомендую писать приложение с нуля, но можете скачать код с GitHub.

5.1. Настройка клиентского проекта

В первую очередь мы создадим свежий NPM-проект, как мы это делали в случае с сервером.

```
mkdir voting-client
cd voting-client
npm init -y
```

Теперь для нашего приложения нужна стартовая HTML-страница. Положим её в `dist/index.html`:

```
dist/index.html
<!DOCTYPE html>
<html>
<body>
  <div id="app"></div>
  <script src="bundle.js"></script>
</body>
</html>
```

Документ содержит лишь `<div>` с ID `app`, сюда мы и поместим

наше приложение. В ту же папку надо будет положить и файл `bundle.js`.

Создадим первый JavaScript-файл, который станет входной точкой приложения. Пока что можно просто поместить в него простое логирующее выражение:

```
src/index.js
console.log('I am alive!');
```

Для облегчения процесса создания приложения воспользуемся Webpack и его сервером разработки, добавив их к нашему проекту:

```
npm install --save-dev webpack webpack-dev-server
```

Если вы их пока не устанавливали, стоит установить эти же пакеты глобально, чтобы можно было удобно запускать всё необходимое из командной строки: `npm install -g webpack webpack-dev-server`.

Добавим файл конфигурации Webpack, соответствующий созданным ранее файлам, в корень проекта:

```
webpack.config.js
module.exports = {
  entry: [
    './src/index.js'
  ],
  output: {
    path: __dirname + '/dist',
    publicPath: '/',
```

```
    filename: 'bundle.js'
  },
  devServer: {
    contentBase: './dist'
  }
};
```

Он обнаружит нашу входную точку `index.js` и встроит всё необходимое в бандл `dist/bundle.js`. Папка `dist` будет базовой и для сервера разработки.

Теперь можно запустить webpack для создания `bundle.js`:

```
webpack
```

Далее запустим сервер, после чего тестовая страница станет доступна в localhost:8080 (включая логирующее выражение из `index.js`).

```
webpack-dev-server
```

Поскольку мы собрались использовать в клиентском коде React JSX синтаксис и ES6, то нам нужна еще пара инструментов. Babel умеет работать с ними обоими, поэтому подключим его и его Webpack-загрузчик:

```
npm install --save-dev babel-core babel-loader babel-preset-es2015
babel-preset-react
```

Включаем в `package.json` поддержку Babel'ем ES6/ES2015 и

React JSX, активируя только что установленные пресеты:

```
package.json
"babel": {
  "presets": ["es2015", "react"]
}
```

Теперь изменим конфигурационный файл Webpack, чтобы он мог найти `.jsx` и `.js` файлы и обработать их с помощью Babel:

```
webpack.config.js
module.exports = {
  entry: [
    './src/index.js'
  ],
  module: {
    loaders: [{
      test: /\.jsx?$/,
      exclude: /node_modules/,
      loader: 'babel'
    }]
  },
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  output: {
    path: __dirname + '/dist',
    publicPath: '/',
    filename: 'bundle.js'
  },
  devServer: {
    contentBase: './dist'
  }
};
```

Не будем тратить время на CSS. Если вы хотите сделать приложение красивее, то можете сами добавить в него стили. Либо можете воспользоваться стилями из этого коммита. В дополнение

к CSS-файлу будет добавлена Webpack-поддержка для подключения стилей (и автопрефиксов), а также компонент, немного улучшенный для визуализации.

5.1.1. Поддержка модульного тестирования

Для клиентского кода мы тоже будем писать модульные тесты. Для этого воспользуемся теми же библиотеками — Mocha и Chai:

```
npm install --save-dev mocha chai
```

Также будем тестировать и React-компоненты, для чего нам понадобится DOM. В качестве альтернативы можно предложить прогнать в настоящем веб-браузере тесты библиотекой наподобие Karma. Но это не является необходимостью для нас, поскольку мы можем обойтись средствами jsdom, реализацией DOM на чистом JavaScript внутри Node:

```
npm install --save-dev jsdom
```

Для последней версии jsdom требуется io.js или Node.js 4.0.0. Если вы пользуетесь более старой версией Node, то вам придётся установить и более старый jsdom:

```
npm install --save-dev jsdom@3
```

Также нам понадобится несколько строк настройки jsdom для использования React. В частности, создадим jsdom-версии

объектов `document` и `window`, предоставляемых браузером. Затем положим их в глобальный объект, чтобы React мог найти их, когда будет обращаться к `document` или `window`. Для этой настройки подготовим вспомогательный тестовый файл:

```
test/test_helper.js
import jsdom from 'jsdom';

const doc = jsdom.jsdom('<!doctype html><html><body></body>
</html>');
const win = doc.defaultView;

global.document = doc;
global.window = win;
```

Кроме того, нам нужно взять все свойства, содержащиеся в jsdom-объекте `window` (например, `navigator`), и добавить их в объект global в Node.js. Это делается для того, чтобы предоставляемые объектом `window` свойства можно было использовать без префикса `window.`, как это происходит в браузерном окружении. От этого зависит часть кода внутри React:

```
test/test_helper.js
import jsdom from 'jsdom';

const doc = jsdom.jsdom('<!doctype html><html><body></body>
</html>');
const win = doc.defaultView;

global.document = doc;
global.window = win;

Object.keys(window).forEach((key) => {
  if (!(key in global)) {
    global[key] = window[key];
  }
});
```

Также мы воспользуемся Immutable коллекциями, поэтому придётся прибегнуть к той же уловке, что и в случае с сервером, чтобы внедрить поддержку Chai. Установим оба пакета — immutable и chai-immutable:

```
npm install --save immutable
npm install --save-dev chai-immutable
```

Далее пропишем их в тестовом вспомогательном файле:

```
test/test_helper.js
import jsdom from 'jsdom';
import chai from 'chai';
import chaiImmutable from 'chai-immutable';

const doc = jsdom.jsdom('<!doctype html><html><body></body>
</html>');
const win = doc.defaultView;

global.document = doc;
global.window = win;

Object.keys(window).forEach((key) => {
  if (!(key in global)) {
    global[key] = window[key];
  }
});

chai.use(chaiImmutable);
```

Последний шаг перед запуском тестов: добавим в файл `package.json` команду для их запуска:

```
package.json
"scripts": {
```

```
  "test": "mocha --compilers js:babel-core/register --require
./test/test_helper.js \"test/**/*@(.js|.jsx)\""
},
```

Почти такую же команду мы использовали в `серверном package.json`. Разница лишь в спецификации тестового файла: на сервере мы использовали `--recursive`, но в этом случае не будут обнаруживаться `.jsx`-файлы. Для возмодности найти и `.js`, и `.jsx`-файлы используем glob.

Было бы удобно непрерывно прогонять тесты при любых изменениях в коде. Для этого можно добавить команду `test:watch`, идентичную применяемой на сервере:

```
package.json
"scripts": {
  "test": "mocha --compilers js:babel-core/register --require
./test/test_helper.js 'test/**/*.@(js|jsx)'",
  "test:watch": "npm run test -- --watch"
},
```

5.2. React и react-hot-loader

Инфраструктура Webpack и Babel готова, займемся React!

При построении React-приложений с помощью Redux и Immutable мы можем писать так называемые чистые компоненты (Pure Components, их ещё иногда называют Dumb Components). Идея та же, что и в основе чистых функций, должны соблюдаться два правила:

1. Чистый компонент получает все данные в виде свойств, как функция получает данные в виде аргументов. Не должно быть никаких побочных эффектов — чтения данных откуда либо, инициации сетевых запросов и т.д.
2. В целом у чистого компонента нет внутреннего состояния. Отрисовка зависит исключительно от входных свойств. Если дважды что-то отрисовать с помощью одного компонента, имеющего одни и те же свойства, то в результате мы получим один и тот же интерфейс. У компонента нет скрытого состояния, которое может повлиять на процесс отрисовки.

Использование чистых компонентов упрощает код, как и использование чистых функций: мы можем понять, что делает компонент, посмотрев на его входные данные и результат отрисовки. Больше нам ничего не нужно знать о компоненте. Тестировать его также не сложно, почти как и тестировать логику приложения но основе чистых функций.

Но если компонент не может обладать состоянием, то где оно *будет* находиться? В неизменяемой структуре данных внутри Redux store! Отделить состояние от кода пользовательского интерфейса — отличная идея. React-компоненты представляют собой всего лишь не имеющую состояния *проекцию* состояния на данный момент времени.

Но не будем забегать вперёд. Добавим React в наш проект:

```
npm install --save react react-dom
```

Также настроим react-hot-loader. Этот инструмент сильно ускорит процесс разработки благодаря перезагрузке кода без потери текущего состояния приложения.

```
npm install --save-dev react-hot-loader
```

Было бы глупо пренебрегать react-hot-loader, ведь наша архитектура только поощряет его использование. По сути, создание Redux и react-hot-loader — две части одной истории!

Для поддержки этого загрузчика сделаем несколько обновлений в `webpack.config.js`. Вот что получилось:

```
webpack.config.js
var webpack = require('webpack');

module.exports = {
  entry: [
    'webpack-dev-server/client?http://localhost:8080',
    'webpack/hot/only-dev-server',
    './src/index.js'
  ],
  module: {
    loaders: [{
      test: /\.jsx?$/,
      exclude: /node_modules/,
      loader: 'react-hot!babel'
    }]
  },
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  output: {
    path: __dirname + '/dist',
    publicPath: '/',
    filename: 'bundle.js'
  },
```

```
  devServer: {
    contentBase: './dist',
    hot: true
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ]
};
```

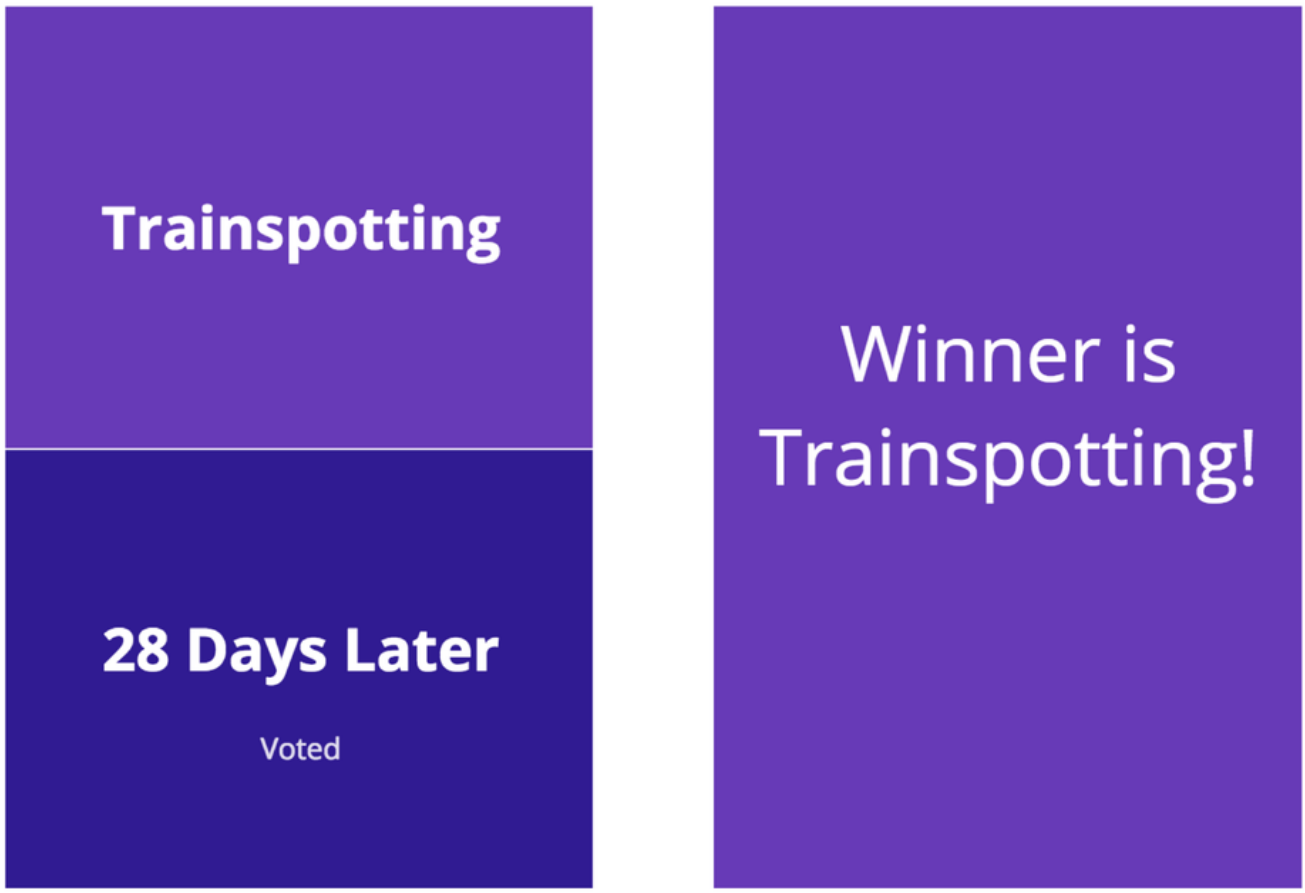
В секцию `entry` включены две новые вещи для входных точек нашего приложения: клиентская библиотека от Webpack сервера разработки и загрузчик модулей (hot module loader) Webpack. Благодаря этому мы сможем использовать инфраструктуру Webpack для горячей замены модулей. По умолчанию такая замена не поддерживается, поэтому в секции `plugins` придётся подгружать соответствующий плагин и активировать поддержку в секции `devServer`.

В секции `loaders` мы настраиваем загрузчик `react-hot`, чтобы он наряду с Babel мог работать с файлами .js и .jsx.

Теперь при запуске или рестарте сервера разработки мы увидим в консоли сообщение о включении поддержки горячей замены модулей (Hot Module Replacement).

5.3. Создание пользовательского интерфейса для экрана голосования

Этот экран будет очень простым: пока голосование не завершилось, всегда будут отображаться две кнопки, по одной для каждой из двух записей. А по завершении голосования будет показан победитель.



По большей части пока мы занимались разработкой через тестирование, при создании React-компонентов применим другой подход: сначала пишем компоненты, а затем тесты. Дело в том, что Webpack и react-hot-loader имеют ещё более короткий контур обратной связи, чем модульные тесты. Кроме того, при создании интерфейса нет ничего эффективнее, чем наблюдать его работу своими глазами.

Допустим, нам нужно создать компонент `Voting` и рендерить его в качестве входной точки приложения. Можно смонтировать его в div `#app`, который ранее был добавлен в `index.html`. И придётся переименовать `index.js` в `index.jsx`, ведь теперь он содержит

JSX-разметку:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import Voting from './components/Voting';

const pair = ['Trainspotting', '28 Days Later'];

ReactDOM.render(
  <Voting pair={pair} />,
  document.getElementById('app')
);
```

Компонент `Voting` получает пару записей в виде свойств. Пока что мы эту пару захардкодим, а позднее заменим реальными данными. Компонент чистый, поэтому ему не важно, откуда берутся данные.

Изменим имя стартового файла в `webpack.config.js`:

```
webpack.config.js
entry: [
  'webpack-dev-server/client?http://localhost:8080',
  'webpack/hot/only-dev-server',
  './src/index.jsx'
],
```

Теперь при запуске или рестарте webpack-dev-server мы увидим сообщение об отсутствии компонента `Voting`. Напишем его первую версию:

```
src/components/Voting.jsx
import React from 'react';
```

```
export default React.createClass({
  getPair: function() {
    return this.props.pair || [];
  },
  render: function() {
    return <div className="voting">
      {this.getPair().map(entry =>
        <button key={entry}>
          <h1>{entry}</h1>
        </button>
      )}
    </div>;
  }
});
```

Пара записей выводятся в виде кнопок, их можно увидеть в браузере. Попробуйте внести в код компонента какие-нибудь изменения, они *немедленно* появятся в браузере. Без рестартов и перезагрузок страницы. Это к вопросу о скорости обратной связи.

Если вы видите не то, что ожидаете, то проверьте выходные данные webpack-dev-server, а также лог браузера.

Теперь можно добавить первый модульный тест. Он будет расположен в файле `Voting_spec.jsx`:

```
test/components/Voting_spec.jsx
import Voting from '../../src/components/Voting';

describe('Voting', () => {

});
```

Для проверки отрисовки кнопок по свойству `pair`, нужно отрендерить компонент и проверить результат. Для этого воспользуемся вспомогательной функцией renderIntoDocument из

пакета тестовых утилит React, который сначала нужно установить:

```
npm install --save react-addons-test-utils
```

```
test/components/Voting_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument
} from 'react-addons-test-utils';
import Voting from '../../src/components/Voting';

describe('Voting', () => {

  it('renders a pair of buttons', () => {
    const component = renderIntoDocument(
      <Voting pair={["Trainspotting", "28 Days Later"]} />
    );
  });

});
```

После отрисовки компонента для поиска кнопок можно использовать другую вспомогательную функцию React — scryRenderedDOMComponentsWithTag. Их должно быть две, а что текстовое содержимое элементов должно совпадать с нашими двумя записями.

```
test/components/Voting_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithTag
} from 'react-addons-test-utils';
import Voting from '../../src/components/Voting';
import {expect} from 'chai';
```

```
describe('Voting', () => {

  it('renders a pair of buttons', () => {
    const component = renderIntoDocument(
      <Voting pair={["Trainspotting", "28 Days Later"]} />
    );
    const buttons = scryRenderedDOMComponentsWithTag(component,
'button');

    expect(buttons.length).to.equal(2);
    expect(buttons[0].textContent).to.equal('Trainspotting');
    expect(buttons[1].textContent).to.equal('28 Days Later');
  });

});
```

Запускаем тест и проверяем:

```
npm run test
```

При клике на любую кнопку компонент должен вызвать callback-функцию. Она должна быть передана компоненту в виде свойства, как и пара записей. Добавим в тест соответствующую проверку. Эмулируем клик с помощью объекта Simulate из тестовых утилит React:

```
test/components/Voting_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithTag,
  Simulate
} from 'react-addons-test-utils';
import Voting from '../../src/components/Voting';
import {expect} from 'chai';
```

```
describe('Voting', () => {

  // ...

  it('invokes callback when a button is clicked', () => {
    let votedWith;
    const vote = (entry) => votedWith = entry;

    const component = renderIntoDocument(
      <Voting pair={["Trainspotting", "28 Days Later"]}
              vote={vote}/>
    );
    const buttons = scryRenderedDOMComponentsWithTag(component,
'button');
    Simulate.click(buttons[0]);

    expect(votedWith).to.equal('Trainspotting');
  });

});
```

Написать этот тест не сложно. Для кнопок нам лишь нужен обработчик `onClick`, вызывающий `vote` с правильной записью:

```
src/components/Voting.jsx
import React from 'react';

export default React.createClass({
  getPair: function() {
    return this.props.pair || [];
  },
  render: function() {
    return <div className="voting">
      {this.getPair().map(entry =>
        <button key={entry}
                onClick={() => this.props.vote(entry)}>
          <h1>{entry}</h1>
        </button>
      )}
    </div>;
  }
});
```

Таким образом мы с помощью чистых компонентов будем управлять пользовательским вводом и действиями: компоненты не будут самостоятельно обрабатывать actions, а будут просто вызывать callback-и.

Здесь мы вернулись к разработке через тестирование. В ходе создания интерфейса мы будем ещё не раз переключаться с одного подхода на другой, в зависимости того, что будет полезнее в текущих обстоятельствах.

Когда пользователь проголосовал за какую-то позицию, не стоит позволять ему делать это повторно. Мы могли бы обработать эту ситуацию внутри состояния компонента, но поскольку стараемся сохранять компоненты чистыми, вынесем эту логику наружу. Компонент получит свойство `hasVoted`, и выбранный элемент мы пока захардкодим:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import Voting from './components/Voting';

const pair = ['Trainspotting', '28 Days Later'];

ReactDOM.render(
  <Voting pair={pair} hasVoted="Trainspotting" />,
  document.getElementById('app')
);
```

И допишем компонент голосования:

```
src/components/Voting.jsx
import React from 'react';

export default React.createClass({
  getPair: function() {
    return this.props.pair || [];
  },
  isDisabled: function() {
    return !!this.props.hasVoted;
  },
  render: function() {
    return <div className="voting">
      {this.getPair().map(entry =>
        <button key={entry}
                disabled={this.isDisabled()}
                onClick={() => this.props.vote(entry)}>
          <h1>{entry}</h1>
        </button>
      )}
    </div>;
  }
});
```

Добавим небольшой label на кнопку, который будет становиться видимым при получении свойства `hasVoted`. Сделаем вспомогательный метод `hasVotedFor`, который будет решать, нужно ли его отрисовывать:

```
src/components/Voting.jsx
import React from 'react';

export default React.createClass({
  getPair: function() {
    return this.props.pair || [];
  },
  isDisabled: function() {
    return !!this.props.hasVoted;
  },
  hasVotedFor: function(entry) {
    return this.props.hasVoted === entry;
  },
  render: function() {
```

```
    return <div className="voting">
      {this.getPair().map(entry =>
        <button key={entry}
                disabled={this.isDisabled()}
                onClick={() => this.props.vote(entry)}>
          <h1>{entry}</h1>
          {this.hasVotedFor(entry) ?
            <div className="label">Voted</div> :
            null}
        </button>
      )}
    </div>;
  }
});
```

Когда у нас появится финальный победитель, то отображаться будет только он. Для него мы сделаем другое свойство, значение которого также временно захардкодим:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import Voting from './components/Voting';

const pair = ['Trainspotting', '28 Days Later'];

ReactDOM.render(
  <Voting pair={pair} winner="Trainspotting" />,
  document.getElementById('app')
);
```

Можем обработать это в компоненте, отрисовывая div победителя или кнопки выбора в зависимости от значения свойства winner:

```
src/components/Voting.jsx
import React from 'react';

export default React.createClass({
  getPair: function() {
    return this.props.pair || [];
```

```
  },
  isDisabled: function() {
    return !!this.props.hasVoted;
  },
  hasVotedFor: function(entry) {
    return this.props.hasVoted === entry;
  },
  render: function() {
    return <div className="voting">
      {this.props.winner ?
        <div ref="winner">Winner is {this.props.winner}!</div> :
        this.getPair().map(entry =>
          <button key={entry}
                  disabled={this.isDisabled()}
                  onClick={() => this.props.vote(entry)}>
            <h1>{entry}</h1>
            {this.hasVotedFor(entry) ?
              <div className="label">Voted</div> :
              null}
          </button>
        )}
    </div>;
  }
});
```

Теперь мы получили нужную нам функциональность, но код отрисовки пока выглядит немного неопрятно. Лучше извлечь из него отдельные компоненты, чтобы компонент экрана голосования (vote screen) отрисовывал либо компонент победителя (winner), либо компонент голосования (vote). В случае компонента winner будет отрисовываться просто div:

```
src/components/Winner.jsx
import React from 'react';

export default React.createClass({
  render: function() {
    return <div className="winner">
      Winner is {this.props.winner}!
    </div>;
  }
});
```

Компонент голосования будет практически таким же, как и прежде, нужны лишь кнопки голосования:

```
src/components/Vote.jsx
import React from 'react';

export default React.createClass({
  getPair: function() {
    return this.props.pair || [];
  },
  isDisabled: function() {
    return !!this.props.hasVoted;
  },
  hasVotedFor: function(entry) {
    return this.props.hasVoted === entry;
  },
  render: function() {
    return <div className="voting">
      {this.getPair().map(entry =>
        <button key={entry}
                disabled={this.isDisabled()}
                onClick={() => this.props.vote(entry)}>
          <h1>{entry}</h1>
          {this.hasVotedFor(entry) ?
            <div className="label">Voted</div> :
            null}
        </button>
      )}
    </div>;
  }
});
```

А сам компонент голосования теперь просто принимает решение, какой из двух компонентов нужно отрисовать:

```
src/components/Voting.jsx
import React from 'react';
import Winner from './Winner';
import Vote from './Vote';
```

```
export default React.createClass({
  render: function() {
    return <div>
      {this.props.winner ?
        <Winner ref="winner" winner={this.props.winner} /> :
        <Vote {...this.props} />}
    </div>;
  }
});
```

Обратите внимание, что в компонент победителя добавлен ref. Мы будем использовать его в модульных тестах для получения необходимого DOM-элемента.

У нас готов чистый компонент голосования! Заметьте, мы до сих пор не реализовали *никакую* логику: есть только кнопки, которые пока ничего не делают, за исключением вызова callback-ов. Компоненты ответственны лишь за отрисовку интерфейса. Позднее мы добавим логику приложения, подключив интерфейс к Redux store.

Теперь напишем ещё несколько модульных тестов для проверки новой функциональности. Наличие свойства `hasVoted` должно приводить к отключению кнопок голосования:

```
test/components/Voting_spec.jsx
it('отключает кнопку, как только пользователь проголосует', () => {
  const component = renderIntoDocument(
    <Voting pair={["Trainspotting", "28 Days Later"]}
            hasVoted="Trainspotting" />
  );
  const buttons = scryRenderedDOMComponentsWithTag(component,
'button');

  expect(buttons.length).to.equal(2);
```

```
  expect(buttons[0].hasAttribute('disabled')).to.equal(true);
  expect(buttons[1].hasAttribute('disabled')).to.equal(true);
});
```

Label `Voted` появляется на той кнопке, чья запись совпадает со значением свойства `hasVoted`:

```
test/components/Voting_spec.jsx
it('добавляет label к записи, за которую проголосовали', () => {
  const component = renderIntoDocument(
    <Voting pair={["Trainspotting", "28 Days Later"]}
            hasVoted="Trainspotting" />
  );
  const buttons = scryRenderedDOMComponentsWithTag(component,
'button');

  expect(buttons[0].textContent).to.contain('Voted');
});
```

Когда у нас появляется победитель, то должны отрисовываться не кнопки, а элемент с ref'ом победителя:

```
test/components/Voting_spec.jsx
it('отрисовывает только победителя', () => {
  const component = renderIntoDocument(
    <Voting winner="Trainspotting" />
  );
  const buttons = scryRenderedDOMComponentsWithTag(component,
'button');
  expect(buttons.length).to.equal(0);

  const winner = ReactDOM.findDOMNode(component.refs.winner);
  expect(winner).to.be.ok;
  expect(winner.textContent).to.contain('Trainspotting');
});
```

Можно было бы написать тесты для каждого компонента в

отдельности, но я считаю, что в данном случае правильнее тестировать экран голосования в качестве «модуля». Мы тестируем внешнее поведение компонента, а тот факт, что внутри него есть более мелкие компоненты, это уже детали реализации.

5.4. Неизменяемые данные и чистый рендеринг (Pure Rendering)

Мы уже обсудили основные достоинства неизменяемых данных, но есть ещё одно, крайне практичное преимущество, связанное с их использованием вместе с React. Если в качестве свойств компонента мы будем использовать только неизменяемые данные, а сами компоненты напишем в соответствии с критериями чистоты, то заставим React применять более эффективную стратегию обнаружения изменений в свойствах.

Для этого применим PureRenderMixin из add-on-пакета. Если добавить mixin в компонент, то React станет по-другому проверять свойства (и состояние) компонента на наличие изменений. Сравнение будет не глубоким, а поверхностным, что гораздо быстрее.

Целесообразность этого решения заключается в том, что изменений в immutable структурах быть не может. Так что если свойства компонента есть неизменяемые данные, и они указывают на те же значения между отрисовками, то нет необходимости рендерить компонент заново!

Давайте напишем модульные тесты на этот случай.

Предполагается, что у нас чистый компонент, так что если дать ему изменяемый массив, а затем сделать в нём какое-то изменение, то компонент *не должен* перерисоваться:

```
test/components/Voting_spec.jsx
it('отрисовывается как чистый компонент', () => {
  const pair = ['Trainspotting', '28 Days Later'];
  const container = document.createElement('div');
  let component = ReactDOM.render(
    <Voting pair={pair} />,
    container
  );

  let firstButton = scryRenderedDOMComponentsWithTag(component,
'button')[0];
  expect(firstButton.textContent).to.equal('Trainspotting');

  pair[0] = 'Sunshine';
  component = ReactDOM.render(
    <Voting pair={pair} />,
    container
  );
  firstButton = scryRenderedDOMComponentsWithTag(component,
'button')[0];
  expect(firstButton.textContent).to.equal('Trainspotting');
});
```

Вместо `renderIntoDocument` мы вручную создаём родительский `<div>` и дважды отрисовываем в него, эмулируя перерисовку.

Нужно явно задать в свойствах **новый** неизменяемый список, чтобы изменения отразились в интерфейсе:

```
test/components/Voting_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithTag,
```

```
  Simulate
} from 'react-addons-test-utils';
import {List} from 'immutable';
import Voting from '../../src/components/Voting';
import {expect} from 'chai';

describe('Voting', () => {

  // ...

  it('обновляет DOM при изменении свойства', () => {
    const pair = List.of('Trainspotting', '28 Days Later');
    const container = document.createElement('div');
    let component = ReactDOM.render(
      <Voting pair={pair} />,
      container
    );

    let firstButton = scryRenderedDOMComponentsWithTag(component,
'button')[0];
    expect(firstButton.textContent).to.equal('Trainspotting');

    const newPair = pair.set(0, 'Sunshine');
    component = ReactDOM.render(
      <Voting pair={newPair} />,
      container
    );
    firstButton = scryRenderedDOMComponentsWithTag(component,
'button')[0];
    expect(firstButton.textContent).to.equal('Sunshine');
  });

});
```

Обычно я не заморачиваюсь написанием подобных тестов, а просто предполагаю использование PureRenderMixin. Но в нашем случае тесты просто помогают разобраться в происходящем. Здесь они демонстрируют, что компонент ведёт себя не так, как ожидается: обновление интерфейса происходит в **обоих** случаях. Это означает проведение глубоких проверок свойств, чего мы как раз и хотели избежать с помощью неизменяемых данных.

Всё встаёт на свои места после того, как мы включим PureRenderMixin в нашем компоненте. Сначала установим пакет:

```
npm install --save react-addons-pure-render-mixin
```

После его добавления в компоненты тесты начинают выполняться успешно:

```
src/components/Voting.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';
import Winner from './Winner';
import Vote from './Vote';

export default React.createClass({
  mixins: [PureRenderMixin],
  // ...
});

src/components/Vote.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

export default React.createClass({
  mixins: [PureRenderMixin],
  // ...
});

src/components/Winner.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

export default React.createClass({
  mixins: [PureRenderMixin],
  // ...
});
```

Строго говоря, мы начнём проходить тесты даже при простом включении PureRenderMixin в компоненте голосования, не

обращая внимания на остальные два компонента. Дело в том, что когда React не находит изменений в свойствах Voting, то он пропускает перерисовку всего поддерева компонента.

Но всё же правильнее будет последовательно использовать PureRenderMixin во всех компонентах. Во-первых, это подтвердит их чистоту, а во-вторых, их поведение не изменится даже после перегруппирования.

5.5. Создание пользовательского интерфейса для экрана результатов и обработка переходов (Routing Handling)

Закончили с экраном голосования, теперь перейдём к другому важному экрану нашего приложения: к экрану отображения результатов.

В качестве данных для отображения используются те же пары записей, что и на экране голосования, а также текущие счетчики голосов по каждой записи. Внизу добавляется маленькая кнопка, при нажатии которой мы переходим к голосованию по следующей паре.

Получается, что у нас есть два отдельных экрана, и в каждый момент времени должен отображаться один из них. Для выбора конкретного экрана для отображения можно использовать URL'ы. Назначим путь `#/` для отображения экрана голосования, а путь `#/results` — для отображения экрана результатов.

Подобные вещи легко выполняются с помощью библиотеки react-router, благодаря которой можно ассоциировать друг с другом разные компоненты и пути. Добавим её в наш проект:

```
npm install --save react-router@2.0.0
```

Теперь сконфигурируем пути. Для этого воспользуемся роутером (Router) из React-компонента `Route`, с помощью которого декларативно опишем таблицу соответствий. Пока что у нас есть лишь один путь:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Route} from 'react-router';
import App from './components/App';
import Voting from './components/Voting';

const pair = ['Trainspotting', '28 Days Later'];

const routes = <Route component={App}>
  <Route path="/" component={Voting} />
</Route>;

ReactDOM.render(
  <Voting pair={pair} />,
  document.getElementById('app')
);
```

У нас есть один путь, указывающий на компонент Voting. Также мы определили компонент *корневого* пути, который может использоваться всеми конкретными путями внутри него. Он указывает на компонент `App`, который скоро будет создан.

Задача компонента корневого пути заключается в отрисовке общей для всех разметки. Так должен выглядеть наш корневой компонент `App`:

```
src/components/App.jsx
import React from 'react';
import {List} from 'immutable';

const pair = List.of('Trainspotting', '28 Days Later');

export default React.createClass({
  render: function() {
    return React.cloneElement(this.props.children, {pair: pair});
  }
});
```

Этот компонент отрисовывает свои дочерние компоненты, передаваемые в свойстве `children`. Далее react-router подключает компоненты, определённые для текущего маршрута. Поскольку пока у нас есть только один маршрут для `Voting`, то на данный момент компонент всегда будет отрисовывать `Voting`.

Обратите внимание, что заглушка `pair` перемещёна из `index.jsx` в `App.jsx`. Для клонирования исходных компонентов с передачей кастомного свойства `pair` мы воспользуемся API cloneElement. Это временная мера, позднее можно будет убрать клонирующий вызов.

Выше мы говорили о том, что лучше использовать PureRenderMixin во всех компонентах. Исключением из этого правила является компонент App: из-за особенностей взаимодействия между роутером и React маршруты могут не

измениться. Возможно, в ближайшем будущем ситуация изменится.

Теперь вернёмся к `index.js`, из которого запустим сам роутер, чтобы они инициализировал наше приложение:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import App from './components/App';
import Voting from './components/Voting';

const routes = <Route component={App}>
  <Route path="/" component={Voting} />
</Route>;

ReactDOM.render(
  <Router history={hashHistory}>{routes}</Router>,
  document.getElementById('app')
);
```

Компонент `Router` из пакета `react-router` является корневым для нашего приложения, и он будет использовать механизм сохранения истории на базе `#hash` (в отличие от API для сохранения истории в HTML 5). Передадим ему нашу таблицу соответствий в виде дочернего компонента.

Теперь мы восстановили предыдущую функциональность нашего приложения: оно всего лишь отрисовывает компонент `Voting`. Но в это раз это делается с помощью роутера React, а значит мы легко можем добавлять новые пути. Сделаем это для экрана результатов, который будет обслуживаться новым компонентом

Results:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import App from './components/App';
import Voting from './components/Voting';
import Results from './components/Results';

const routes = <Route component={App}>
  <Route path="/results" component={Results} />
  <Route path="/" component={Voting} />
</Route>;

ReactDOM.render(
  <Router history={hashHistory}>{routes}</Router>,
  document.getElementById('app')
);
```

Здесь задали в компоненте `<Route>` для пути `/results` отрисовку компонента `results`. Все остальные пути покаведут к Voting.

Давайте создадим простую реализацию `Results` и посмотрим на работу роутинга:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

export default React.createClass({
  mixins: [PureRenderMixin],
  render: function() {
    return <div>Hello from results!</div>
  }
});
```

Если открыть в браузере localhost:8080/#/results, то вы увидите сообщение от компонента Results. Корневной маршрут должен отобразить кнопки голосования. С помощью кнопок «вперёд» и «назад» в браузере вы можете переключаться между путями, и отображаемый компонент будет меняться. Вот он, роутер в действии!

Больше в нашем приложении мы ничего не будем делать с помощью роутера React. Хотя у библиотеки куда больше возможностей, можете посмотреть её документацию.

Теперь, когда у нас есть временный компонент Results, давайте заставим его сделать что-то полезное. Пусть он отображает те же две записи, которые сейчас участвуют в голосовании в компоненте Voting:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

export default React.createClass({
  mixins: [PureRenderMixin],
  getPair: function() {
    return this.props.pair || [];
  },
  render: function() {
    return <div className="results">
      {this.getPair().map(entry =>
        <div key={entry} className="entry">
          <h1>{entry}</h1>
        </div>
      )}
    </div>;
  }
});
```

Раз это экран результатов, то нужно отобразить текущее распределение голосов, ведь именно это люди ожидают увидеть. Передадим в компонент из корневого компонента App временный результат голосования Map:

```
src/components/App.jsx
import React from 'react';
import {List, Map} from 'immutable';

const pair = List.of('Trainspotting', '28 Days Later');
const tally = Map({'Trainspotting': 5, '28 Days Later': 4});

export default React.createClass({
  render: function() {
    return React.cloneElement(this.props.children, {
      pair: pair,
      tally: tally
    });
  }
});
```

Теперь настроим компонент Results для отображения этих чисел:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

export default React.createClass({
  mixins: [PureRenderMixin],
  getPair: function() {
    return this.props.pair || [];
  },
  getVotes: function(entry) {
    if (this.props.tally && this.props.tally.has(entry)) {
      return this.props.tally.get(entry);
    }
    return 0;
  },
  render: function() {
```

```
    return <div className="results">
      {this.getPair().map(entry =>
        <div key={entry} className="entry">
          <h1>{entry}</h1>
          <div className="voteCount">
            {this.getVotes(entry)}
          </div>
        </div>
      )}
    </div>;
  }
});
```

А теперь давайте сменим передачу и добавим модульный тест для текущего поведения компонента Results, чтобы удостовериться, что позднее мы его не сломаем. Компонент должен отрисовывать div'ы для каждой записи, внутри которых отображать имена самих записей и текущее количество голосов. Если за запись никто не проголосовал, то пусть отображается ноль:

```
test/components/Results_spec.jsx
import React from 'react';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithClass
} from 'react-addons-test-utils';
import {List, Map} from 'immutable';
import Results from '../../src/components/Results';
import {expect} from 'chai';

describe('Results', () => {

  it('renders entries with vote counts or zero', () => {
    const pair = List.of('Trainspotting', '28 Days Later');
    const tally = Map({'Trainspotting': 5});
    const component = renderIntoDocument(
      <Results pair={pair} tally={tally} />
    );
    const entries = scryRenderedDOMComponentsWithClass(component,
'entry');
    const [train, days] = entries.map(e => e.textContent);
```

```
    expect(entries.length).to.equal(2);
    expect(train).to.contain('Trainspotting');
    expect(train).to.contain('5');
    expect(days).to.contain('28 Days Later');
    expect(days).to.contain('0');
  });

});
```

Теперь поговорим о кнопке «Next», используемой для перехода к следующему голосованию. С точки зрения компонента, в свойствах должна быть просто callback-функция. Она должна вызываться компонентом, когда внутри него нажимается кнопка «Next». Сформулируем достаточно простой модульный тест, весьма похожий на тот, что мы делали для кнопок голосования:

```
test/components/Results_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithClass,
  Simulate
} from 'react-addons-test-utils';
import {List, Map} from 'immutable';
import Results from '../../src/components/Results';
import {expect} from 'chai';


describe('Results', () => {

  // ...

  it('вызывает callback при нажатии кнопки Next', () => {
    let nextInvoked = false;
    const next = () => nextInvoked = true;

    const pair = List.of('Trainspotting', '28 Days Later');
    const component = renderIntoDocument(
      <Results pair={pair}
               tally={Map()}
               next={next}/>
```

```
    );
    Simulate.click(ReactDOM.findDOMNode(component.refs.next));

    expect(nextInvoked).to.equal(true);
  });

});
```

Реализация во многом схожа с кнопками голосования. Получилось немного проще, поскольку не нужно передавать аргументы:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

export default React.createClass({
  mixins: [PureRenderMixin],
  getPair: function() {
    return this.props.pair || [];
  },
  getVotes: function(entry) {
    if (this.props.tally && this.props.tally.has(entry)) {
      return this.props.tally.get(entry);
    }
    return 0;
  },
  render: function() {
    return <div className="results">
      <div className="tally">
        {this.getPair().map(entry =>
          <div key={entry} className="entry">
            <h1>{entry}</h1>
            <div class="voteCount">
              {this.getVotes(entry)}
            </div>
          </div>
        )}
      </div>
      <div className="management">
        <button ref="next"
                className="next"
                onClick={this.props.next}>
          Next
        </button>
```

```
      </div>
    </div>;
  }
});
```

Как и в случае с экраном голосования, на экране результатов должен отобразиться победитель:

```
test/components/Results_spec.jsx
it('отрисовывает финального победителя', () => {
  const component = renderIntoDocument(
    <Results winner="Trainspotting"
             pair={["Trainspotting", "28 Days Later"]}
             tally={Map()} />
  );
  const winner = ReactDOM.findDOMNode(component.refs.winner);
  expect(winner).to.be.ok;
  expect(winner.textContent).to.contain('Trainspotting');
});
```

Можно реализовать это с помощью повторного использования компонента Winner, уже разработанного для экрана голосования. Как только у нас определяется финальный победитель, то мы отрисовываем соответствующий компонент вместо стандартного экрана результатов:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';
import Winner from './Winner';

export default React.createClass({
  mixins: [PureRenderMixin],
  getPair: function() {
    return this.props.pair || [];
  },
  getVotes: function(entry) {
    if (this.props.tally && this.props.tally.has(entry)) {
```

```
      return this.props.tally.get(entry);
    }
    return 0;
  },
  render: function() {
    return this.props.winner ?
      <Winner ref="winner" winner={this.props.winner} /> :
      <div className="results">
        <div className="tally">
          {this.getPair().map(entry =>
            <div key={entry} className="entry">
              <h1>{entry}</h1>
              <div className="voteCount">
                {this.getVotes(entry)}
              </div>
            </div>
          )}
        </div>
        <div className="management">
          <button ref="next"
                  className="next"
                  onClick={this.props.next}>
            Next
          </button>
        </div>
      </div>;
  }
});
```

Этому компоненту также пошло бы на пользу разделение на более мелкие составляющие. Например, компонент Tally мог бы отображать пары записей. Если вам нравится эта идея, то смело рефакторьте!

И это почти весь интерфейс, необходимый нашему простому приложению. Пока что написанные нами компоненты ничего не делают, потому что не получают реальных данных или действий. Примечательно, как далеко нам удаётся зайти и без этого. Мы даже смогли внедрить в эти компоненты простые заглушки, чтобы сконцентрироваться на структуре интерфейса.

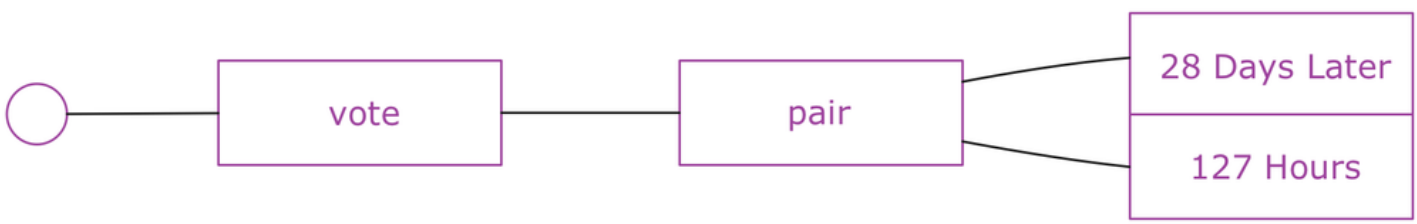
Теперь, когда мы завершили его создание, поговорим о том, как вдохнуть в него жизнь с помощью подключения Redux store к входным и выходным каналам.

5.6. Использование клиентского Redux Store

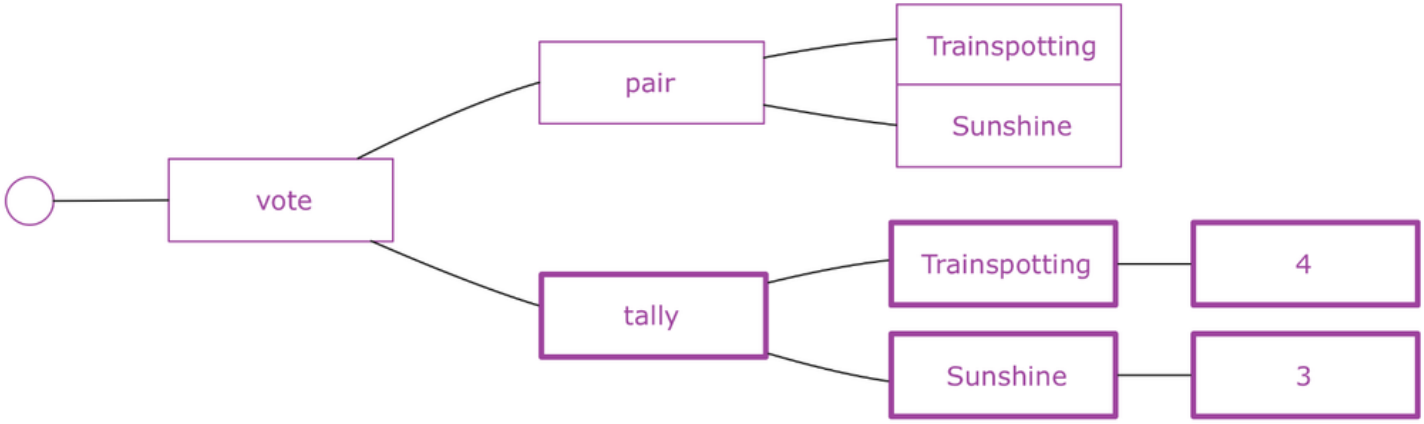
Redux был спроектирован для использования в качестве контейнера состояний приложений, имеющих пользовательский интерфейс. Наше приложение полностью подходит под этот критерий. Пока что мы использовали Redux только на сервере и выяснили, что он и там очень полезен! Теперь можно посмотреть, как он поведёт себя с React-приложением.

Как и в случае с сервером, давайте сначала продумаем возможные состояния нашего приложения. Отличий будет мало, и это не случайно.

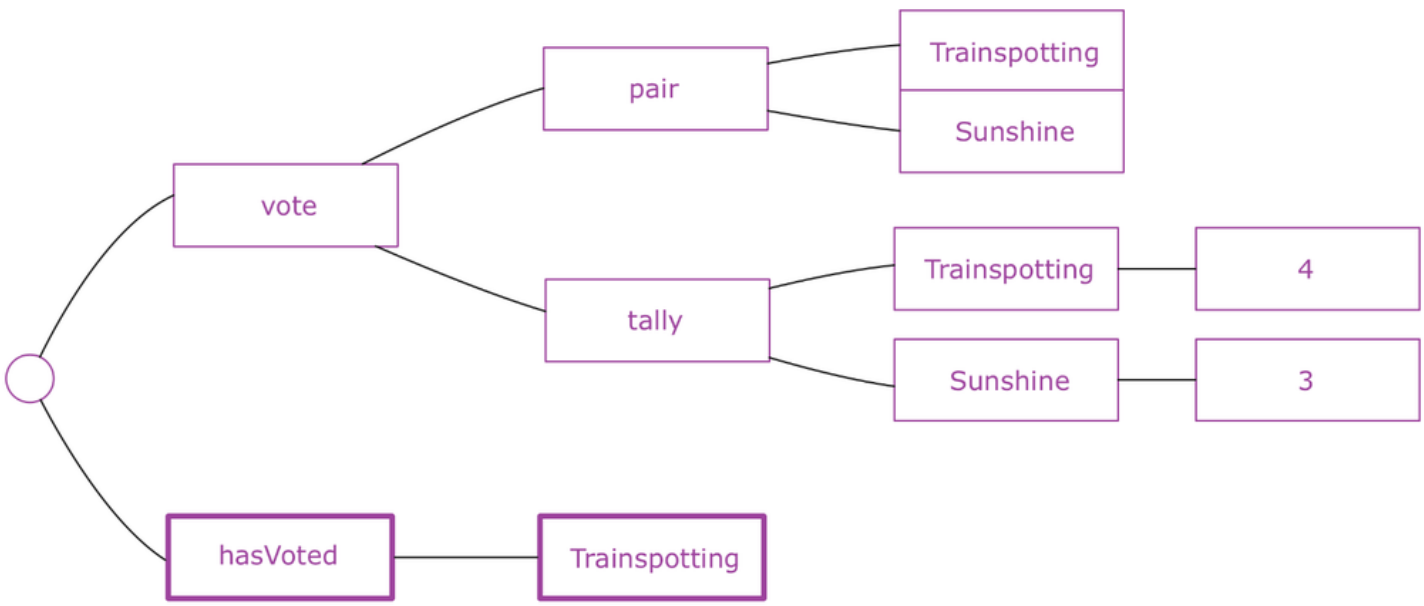
У нас есть интерфейс с двумя экранами. На обоих отображается пара записей, участвующих в голосовании. Имеет смысл сделать состояние vote с парой элементов для голосования:



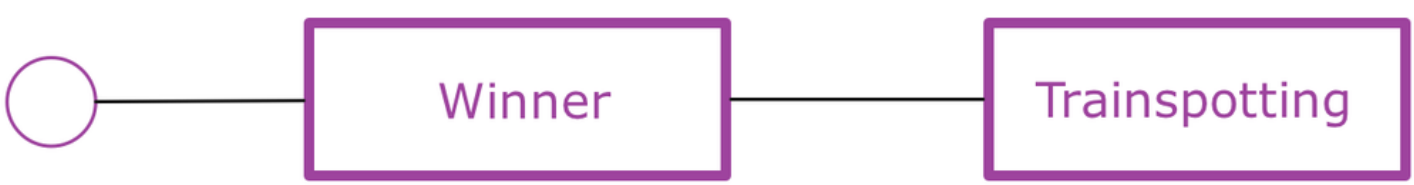
В то же состояние поместим экран результатов, куда выводятся текущее распределение голосов.



Компонент голосования (Voting) отрисовывается иначе, когда пользователь уже проголосовал в текущей паре. Это также должно отслеживаться состоянием:



Когда появляется финальный победитель, только он и должен присутствовать в состоянии:



Обратите внимание, что всё здесь является подмножеством состояний сервера, за исключением сущности `hasVoted`. Это

приводит нас к мыслям о реализации основной логики, действий (actions) и преобразователей (reducers), которые будут использоваться Redux store. Какими они должны быть?

Давайте рассмотрим это с точки зрения того, что может изменить состояние выполняющегося приложения. Один источник изменений состояния — действия пользователя. Сейчас в интерфейсе предусмотрено два возможных сценария взаимодействия:

- Пользователь кликает на одну из кнопок на экране голосования.
- Пользователь кликает на кнопку “Next” на экране результатов.

Кроме того, наш сервер настроен на отправку своего текущего состояния. Скоро мы напишем код для его получения. Это третий источник изменения состояния.

Можно начать с обновления состояния сервера, поскольку сделать это проще всего. Выше мы уже настраивали наш сервер, чтобы он генерировал событие `state`, чья полезная нагрузка представляет собой практически точную копию нарисованных нами клиентских деревьев состояний. Это не совпадение, именно таким мы его и разработали. С точки зрения нашего клиентского reducer-а, целесообразно иметь action, получающий от сервера снэпшот состояния и объединяющий его с клиентским состоянием. Этот action выглядел бы подобным образом:

```
{
  type: 'SET_STATE',
  state: {
```

```
    vote: {...}
  }
}
```

Посмотрим с помощью модульных тестов, как это работает. Получив действие, наподобие приведённого выше, reducer должен объединять его данные с текущим состоянием:

```
test/reducer_spec.js
import {List, Map, fromJS} from 'immutable';
import {expect} from 'chai';

import reducer from '../src/reducer';

describe('reducer', () => {

  it('handles SET_STATE', () => {
    const initialState = Map();
    const action = {
      type: 'SET_STATE',
      state: Map({
        vote: Map({
          pair: List.of('Trainspotting', '28 Days Later'),
          tally: Map({Trainspotting: 1})
        })
      })
    };
    const nextState = reducer(initialState, action);

    expect(nextState).to.equal(fromJS({
      vote: {
        pair: ['Trainspotting', '28 Days Later'],
        tally: {Trainspotting: 1}
      }
    }));
  });

});
```

Reducer должен уметь получать от сокета простую JS-структуру данных. Она должна быть преобразована в неизменяемую

структуру к моменту своего возвращения в виде следующего значения:

```
test/reducer_spec.js
it('обрабатывает SET_STATE с простой JS-нагрузкой', () => {
  const initialState = Map();
  const action = {
    type: 'SET_STATE',
    state: {
      vote: {
        pair: ['Trainspotting', '28 Days Later'],
        tally: {Trainspotting: 1}
      }
    }
  };
  const nextState = reducer(initialState, action);

  expect(nextState).to.equal(fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    }
  }));
});
```

Начальное состояние `undefined` также должно быть корректно инициализировано reducer-ом в виде неизменяемой структуры:

```
test/reducer_spec.js
it('обрабатывает SET_STATE без начального состояния', () => {
  const action = {
    type: 'SET_STATE',
    state: {
      vote: {
        pair: ['Trainspotting', '28 Days Later'],
        tally: {Trainspotting: 1}
      }
    }
  };
  const nextState = reducer(undefined, action);
```

```
  expect(nextState).to.equal(fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    }
  }));
});
```

Таковы наши технические условия. Давайте посмотрим, как их можно выполнить. У нас есть функция-reducer, экспортируемая reducer-модулем:

```
src/reducer.js
import {Map} from 'immutable';

export default function(state = Map(), action) {

  return state;
}
```

Reducer должен обработать action `SET_STATE`. С помощью функции merge из Map можно просто объединить новое состояние со старым в функции-обработчике. Это позволит нам пройти тесты!

```
src/reducer.js
import {Map} from 'immutable';

function setState(state, newState) {
  return state.merge(newState);
}

export default function(state = Map(), action) {
  switch (action.type) {
  case 'SET_STATE':
    return setState(state, action.state);
  }
```

```
  return state;
}
```

Обратите внимание, что мы не стали возиться с «основным» модулем, отделённым от reducer-модуля. Это связано с тем, что логика в преобразователе настолько простая, что о ней не нужно волноваться. Просто выполняем слияние, в то время как на сервере находится полная логика системы голосования. Если возникнет необходимость, можно будет позднее и на клиенте разделить функциональности.

У нас осталось ещё две причины изменения состояния, связанные с действиями пользователя: голосование и нажатие кнопки «Next». В обоих случаях подразумевается взаимодействие с сервером, поэтому мы вернёмся к этом чуть позднее, когда разберёмся с архитектурой подключения к серверу.

Пришло время добавить Redux в наш проект:

```
npm install --save redux
```

Хорошим местом для инициализации store является входная точка `index.jsx`. Создадим его с каким-нибудь состоянием, передав действие `SET_STATE` (это временное решение, пока у нас не появятся реальные данные):

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
```

```
import {createStore} from 'redux';
import reducer from './reducer';
import App from './components/App';
import Voting from './components/Voting';
import Results from './components/Results';

const store = createStore(reducer);
store.dispatch({
  type: 'SET_STATE',
  state: {
    vote: {
      pair: ['Sunshine', '28 Days Later'],
      tally: {Sunshine: 2}
    }
  }
});

const routes = <Route component={App}>
  <Route path="/results" component={Results} />
  <Route path="/" component={Voting} />
</Route>;

ReactDOM.render(
  <Router history={hashHistory}>{routes}</Router>,
  document.getElementById('app')
);
```

Store готов. Как нам теперь передать из него данные в React-компоненты?

5.7. Передача входных данных из Redux в React

В Redux Store содержится неизменяемое состояние приложения. У нас есть не имеющие состояния React-компоненты, принимающие неизменяемые данные на вход. Если мы сможем придумать способ надёжно передавать актуальные данные из store в компоненты, то будет замечательно. При сменах состояния React будет перерисовываться, а PureRenderMixin будет следить за тем, чтобы не перерисовывались те части интерфейса, которые не

должны.

Вместо самостоятельного написания кода синхронизации, можно использовать Redux React биндинги из пакета react-redux:

```
npm install --save react-redux
```

react-redux подключает наши чистые компоненты к Redux store с помощью:

- Отображения состояния из store во входные свойства компонента.
- Отображения actions в свойства callback-ов компонента.

Но прежде нам нужно обернуть наш верхнеуровневый компонент в компонент-провайдер (Provider) из react-redux. Он соединит наше дерево компонентов с Redux Store, что позволит нам позднее связать store с отдельными компонентами.

Поместим провайдера вокруг компонента-роутера. В результате провайдер станет наследником всех компонентов нашего приложения.

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import reducer from './reducer';
import App from './components/App';
import {VotingContainer} from './components/Voting';
```

```
import Results from './components/Results';

const store = createStore(reducer);
store.dispatch({
  type: 'SET_STATE',
  state: {
    vote: {
      pair: ['Sunshine', '28 Days Later'],
      tally: {Sunshine: 2}
    }
  }
});

const routes = <Route component={App}>
  <Route path="/results" component={Results} />
  <Route path="/" component={VotingContainer} />
</Route>;

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);

});
```

Теперь надо подумать о том, какие из компонентов нужно «подключить», чтобы из store поступали все необходимые данные. У нас есть пять компонентов, которые можно разделить на три категории:

- Корневой компонент `App` ни в чём не нуждается, потому что не использует данные.
- `Vote` и `Winner` используются только родительскими компонентами, предающими им все необходимые свойства. Они тоже не нуждаются в подключении.
- Остаются только те компоненты, которые использовались при задании путей: `Voting` и `Results`. Сейчас они получают от `App`

заглушки свойств. Вот их-то и надо подключить к store.

Начнём с компонента `Voting`. Возьмём из react-redux функцию connect, с помощью которой будем подключать компонент. Она берёт сопоставляющую функцию в качестве аргумента, а возвращает другую функцию, принимающую класс React-компонента:

```
connect(mapStateToProps)(SomeComponent);
```

Маппинг-функция осуществляет сопоставление состояния из Redux Store в свойства объекта. Затем эти свойства будут объединены со свойствами подключаемого компонента. В случае с Voting нам всего лишь нужно замапить `pair` и `winner` из Store:

```
src/components/Voting.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin'
import {connect} from 'react-redux';
import Winner from './Winner';
import Vote from './Vote';

const Voting = React.createClass({
  mixins: [PureRenderMixin],
  render: function() {
    return <div>
      {this.props.winner ?
        <Winner ref="winner" winner={this.props.winner} /> :
        <Vote {...this.props} />}
    </div>;
  }
});

function mapStateToProps(state) {
  return {
    pair: state.getIn(['vote', 'pair']),
    winner: state.get('winner')
```

```
  };
}

connect(mapStateToProps)(Voting);

export default Voting;
```

Это не совсем правильно. С точки зрения функционального подхода, функция `connect` на самом деле не должна изменять компонент `Voting`. Он остаётся чистым, не подключённым компонентом. Вместо этого `connect` *возвращает подключённую версию* `Voting`. А это означает, что наш текущий код по сути ничего не делает. Возьмём возвращаемое значение и назовём его `VotingContainer`:

```
src/components/Voting.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';
import {connect} from 'react-redux';
import Winner from './Winner';
import Vote from './Vote';

export const Voting = React.createClass({
  mixins: [PureRenderMixin],
  render: function() {
    return <div>
      {this.props.winner ?
        <Winner ref="winner" winner={this.props.winner} /> :
        <Vote {...this.props} />}
    </div>;
  }
});

function mapStateToProps(state) {
  return {
    pair: state.getIn(['vote', 'pair']),
    winner: state.get('winner')
  };
}
```

```
export const VotingContainer = connect(mapStateToProps)(Voting);
```

Теперь модуль экспортирует чистый компонент `Voting` и подключённый компонент `VotingContainer`. В документации react-redux первый называется «тупым» (dumb) компонентом, а второй — «умным» (smart). Лично я предпочитаю термины «чистый» и «подключённый». Называйте их, как удобнее, но нужно понимать, чем они отличаются:

- Чистый/тупой компонент полностью зависит от предоставляемых ему свойств. Представляет собой эквивалент чистой функции, только компонент.
- Подключённый/умный компонент оборачивает чистую версию в какую-то логику, которая позволяет синхронизироваться с изменяемым состоянием из Redux store. Логика берётся из react-redux.

Обновим нашу роутинг-таблицу, чтобы вместо `Voting` использовался `VotingContainer`. После этого экран голосования будет получать данные, которые мы положим в Redux-хранилище.

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import reducer from './reducer';
import App from './components/App';
import {VotingContainer} from './components/Voting';
import Results from './components/Results';
```

```
const store = createStore(reducer);
store.dispatch({
  type: 'SET_STATE',
  state: {
    vote: {
      pair: ['Sunshine', '28 Days Later'],
      tally: {Sunshine: 2}
    }
  }
});

const routes = <Route component={App}>
  <Route path="/results" component={Results} />
  <Route path="/" component={VotingContainer} />
</Route>;

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);
```

Изменим способ импортирования в модульном тесте для `Voting`, ведь нам больше не нужно использовать Voting в качестве экспортёра *по умолчанию*:

```
test/components/Voting_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithTag,
  Simulate
} from 'react-addons-test-utils';
import {List} from 'immutable';
import {Voting} from '../../src/components/Voting';
import {expect} from 'chai';
```

Больше ничего менять не требуется. Тесты написаны для *чистого* компонента Voting, который остаётся неизменным. Мы просто

добавили обёртку, чтобы подключить его к store.

Теперь то же самое провернём с экраном результатов, которому будем передавать атрибуты состояния `pair` и `winner`. Кроме того, для отображения результатов ему понадобится и `tally`:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';
import {connect} from 'react-redux';
import Winner from './Winner';

export const Results = React.createClass({
  mixins: [PureRenderMixin],
  getPair: function() {
    return this.props.pair || [];
  },
  getVotes: function(entry) {
    if (this.props.tally && this.props.tally.has(entry)) {
      return this.props.tally.get(entry);
    }
    return 0;
  },
  render: function() {
    return this.props.winner ?
      <Winner ref="winner" winner={this.props.winner} /> :
      <div className="results">
        <div className="tally">
          {this.getPair().map(entry =>
            <div key={entry} className="entry">
              <h1>{entry}</h1>
              <div className="voteCount">
                {this.getVotes(entry)}
              </div>
            </div>
          )}
        </div>
        <div className="management">
          <button ref="next"
                  className="next"
                  onClick={this.props.next}>
            Next
          </button>
```

```
      </div>
      </div>;
  }
});

function mapStateToProps(state) {
  return {
    pair: state.getIn(['vote', 'pair']),
    tally: state.getIn(['vote', 'tally']),
    winner: state.get('winner')
  }
}

export const ResultsContainer = connect(mapStateToProps)(Results);
```

В `index.jsx` изменим путь передачи результатов, вместо `Results` будет `ResultsContainer`:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import reducer from './reducer';
import App from './components/App';
import {VotingContainer} from './components/Voting';
import {ResultsContainer} from './components/Results';

const store = createStore(reducer);
store.dispatch({
  type: 'SET_STATE',
  state: {
    vote: {
      pair: ['Sunshine', '28 Days Later'],
      tally: {Sunshine: 2}
    }
  }
});

const routes = <Route component={App}>
  <Route path="/results" component={ResultsContainer} />
  <Route path="/" component={VotingContainer} />
</Route>;
```

```
ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);
```

Наконец, в тесте результатов обновим выражение импорта для `Results`:

```
test/components/Results_spec.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {
  renderIntoDocument,
  scryRenderedDOMComponentsWithClass,
  Simulate
} from 'react-addons-test-utils';
import {List, Map} from 'immutable';
import {Results} from '../../src/components/Results';
import {expect} from 'chai';
```

Таким вот образом можно подключать чистые React-компоненты к Redux-хранилищу, чтобы они могли получать оттуда нужные данные.

Для очень маленьких приложений, обладающих единственным корневым компонентом и не использующих роутинг, в большинстве случаев будет достаточно подключения корневого компонента. А затем уже корень делегирует эти данные в виде свойств для своих дочерних компонентов.

В приложениях с роутингом, вроде создаваемого нами, обычно лучше подключать каждый из компонентов роутера. Но каждый

компонент может быть подключён отдельно, поэтому вы вольны применять разные стратегии в зависимости от архитектуры приложения. На мой взгляд, имеет смысл во всех возможных случаях использовать обычные свойства, поскольку с ними проще понять, какие данные подаются на вход. К тому же вам не придётся разбираться с кодом «подключения».

Итак, мы теперь можем передавать в интерфейс данные из Redux. В `App.jsx` нам больше не нужны заглушки свойств, так что код упрощается:

```
src/components/App.jsx
import React from 'react';

export default React.createClass({
  render: function() {
    return this.props.children;
  }
});
```

5.8. Настройка клиента Socket.io

Поскольку в наш клиент представляет собой Redux-приложение, поговорим о способе подключения к серверному Redux-приложению. На данный момент они оба существуют в своих собственных мирах, никак не взаимодействуя друг с другом.

Сервер уже готов к принятию входящих socket-подключений и передачи им состояния голосования. А у клиента есть Redux-хранилище, в которое можно легко записать входные данные. Осталось только связать их.

Начнём с инфраструктуры. Нам нужно создать Socket.io-канал от браузера к серверу. Для этого воспользуемся библиотекой socket.io-client, являющейся клиентским аналогом библиотеки, которую мы использовали на сервере:

```
npm install --save socket.io-client
```

После импорта библиотеки мы получили функцию `io`, которую можно использовать для подключения к серверу Socket.io. Подключимся к порту 8090 (его мы использовали для сервера):

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import io from 'socket.io-client';
import reducer from './reducer';
import App from './components/App';
import {VotingContainer} from './components/Voting';
import {ResultsContainer} from './components/Results';

const store = createStore(reducer);
store.dispatch({
  type: 'SET_STATE',
  state: {
    vote: {
      pair: ['Sunshine', '28 Days Later'],
      tally: {Sunshine: 2}
    }
  }
});

const socket =
io(`${location.protocol}//${location.hostname}:8090`);

const routes = <Route component={App}>
```

```
  <Route path="/results" component={ResultsContainer} />
  <Route path="/" component={VotingContainer} />
</Route>;

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);
```

Проверьте, что сервер запущен, откройте в браузере клиентское приложение и просмотрите сетевой трафик. Должно быть установлено WebSocket-подключение, в которое отправляются контрольные сигналы Socket.io.

Во время разработки мы будем использовать на странице два Socket.io-подключения: одно наше, а второе для поддержки горячей Webpack-перезагрузки.

5.9. Получение actions с сервера

Получить входные данные из Socket.io канала достаточно просто. При первом подключении и каждом изменении сервер отправляет нам события `state`, достаточно их просто слушать. После получения подобного события передаём в наше хранилище действие `SET_STATE`. Для его обработки там уже есть reducer:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import io from 'socket.io-client';
```

```
import reducer from './reducer';
import App from './components/App';
import {VotingContainer} from './components/Voting';
import {ResultsContainer} from './components/Results';

const store = createStore(reducer);

const socket =
io(`${location.protocol}//${location.hostname}:8090`);
socket.on('state', state =>
  store.dispatch({type: 'SET_STATE', state})
);

const routes = <Route component={App}>
  <Route path="/results" component={ResultsContainer} />
  <Route path="/" component={VotingContainer} />
</Route>;

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);
```

Обратите внимание, что мы убрали заглушечную передачу `SET_STATE`. Она нам больше не нужна, потому что сервер начал передавать реальное состояние.

Взгляните на интерфейс — голосования или результатов: там должна отображаться первая пара записей, переданных с сервера. Подключение между клиентом и сервером установлено!

5.10. Передача actions от React-компонентов

Мы знаем, как передавать в интерфейс *входные* данные от Redux store. Давайте теперь поговорим о передаче из интерфейса *исходящих* действий.

Лучше всего начать с кнопок голосования. При создании интерфейса мы приняли, что компонент `Voting` будет получать свойство `vote`, значением которого является callback-функция. Компонент вызывает её, когда пользователь кликает на кнопки. Но мы пока не обеспечили поддержку этой функции, за исключением модульных тестов.

Что должно произойти, когда пользователь голосует за какую-нибудь запись? Очевидно, что его голос должен быть отправлен на сервер. Подробнее мы поговорим об этом ниже, но там тоже задействуется клиентская логика: компоненту должно назначаться свойство `hasVoted`, чтобы пользователь не мог дважды голосовать в рамках какой-либо пары.

Помимо `SET_STATE` у нас будет второй клиентский Redux action — `VOTE`.Он будет добавлять запись `hasVoted` в состояние:

```
test/reducer_spec.js
it('обрабатывает VOTE с помощью назначения hasVoted', () => {
  const state = fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    }
  });
  const action = {type: 'VOTE', entry: 'Trainspotting'};
  const nextState = reducer(state, action);

  expect(nextState).to.equal(fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    },
    hasVoted: 'Trainspotting'
```

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

Также имеет смысл *не принимать* это свойство в состояние, если действие VOTE по какой-то причине приходит с записью, которая в данный момент не участвует в голосовании:

```
test/reducer_spec.js
it('в случае неправильной записи не назначает hasVoted для VOTE', ()
=> {
  const state = fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    }
  });
  const action = {type: 'VOTE', entry: 'Sunshine'};
  const nextState = reducer(state, action);

  expect(nextState).to.equal(fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    }
  }));
});
```

Расширим логику reducer-а для обработки этого случая:

```
src/reducer.js
import {Map} from 'immutable';

function setState(state, newState) {
  return state.merge(newState);
}

function vote(state, entry) {
  const currentPair = state.getIn(['vote', 'pair']);
  if (currentPair && currentPair.includes(entry)) {
    return state.set('hasVoted', entry);
```

```
  } else {
    return state;
  }
}

export default function(state = Map(), action) {
  switch (action.type) {
  case 'SET_STATE':
    return setState(state, action.state);
  case 'VOTE':
    return vote(state, action.entry);
  }
  return state;
}
```

Свойство `hasVoted` не должно вечно оставаться в состоянии. При переходе к следующей паре оно должно переназначаться, чтобы пользователь мог голосовать дальше. Мы будем обрабатывать эту логику в `SET_STATE`, проверяя, содержится ли в паре голосования текущего состояния запись, за которую пользователь уже проголосовал. Если не содержится, то мы удаляем свойство `hasVoted`:

```
test/reducer_spec.js
it('если пара изменилась, то очищает hasVoted в SET_STATE', () => {
  const initialState = fromJS({
    vote: {
      pair: ['Trainspotting', '28 Days Later'],
      tally: {Trainspotting: 1}
    },
    hasVoted: 'Trainspotting'
  });
  const action = {
    type: 'SET_STATE',
    state: {
      vote: {
        pair: ['Sunshine', 'Slumdog Millionaire']
      }
    }
  };
  const nextState = reducer(initialState, action);
```

```
  expect(nextState).to.equal(fromJS({
    vote: {
      pair: ['Sunshine', 'Slumdog Millionaire']
    }
  }));
});
```

Это можно реализовать сочетанием функции `resetVote` и обработчика действия `SET_STATE`:

```
src/reducer.js
import {List, Map} from 'immutable';

function setState(state, newState) {
  return state.merge(newState);
}

function vote(state, entry) {
  const currentPair = state.getIn(['vote', 'pair']);
  if (currentPair && currentPair.includes(entry)) {
    return state.set('hasVoted', entry);
  } else {
    return state;
  }
}

function resetVote(state) {
  const hasVoted = state.get('hasVoted');
  const currentPair = state.getIn(['vote', 'pair'], List());
  if (hasVoted && !currentPair.includes(hasVoted)) {
    return state.remove('hasVoted');
  } else {
    return state;
  }
}

export default function(state = Map(), action) {
  switch (action.type) {
  case 'SET_STATE':
    return resetVote(setState(state, action.state));
  case 'VOTE':
    return vote(state, action.entry);
  }
```

```
  return state;
}
```

Эта логика определения актуальности свойства `hasVoted` для текущей пары имеет изъяны. Обратите внимание на упражнения ниже для улучшения логики.

Пришла пора подключить свойство `hasVoted` к свойствам `Voting`:

```
src/components/Voting.jsx
function mapStateToProps(state) {
  return {
    pair: state.getIn(['vote', 'pair']),
    hasVoted: state.get('hasVoted'),
    winner: state.get('winner')
  };
}
```

Нам ещё нужно как-то передать в `Voting vote` callback, который приведёт к обработке этого нового действия. `Voting` должен оставаться чистым и независимым от actions или Redux, поэтому задействуем функцию `connect` из react-redux.

react-redux можно использовать для подключения как *входных свойств, так и выходных действий*. Но сначала мы задействуем ещё одну ключевую идею Redux: *создатели действий (Action creators)*.

Как мы уже знаем, действия в Redux представляют собой простые объекты, обладающие (по соглашению) атрибутом `type` и прочими специфическими данными. Мы создавали эти действия по мере

надобности с помощью объектных литералов. Но предпочтительнее использовать маленькие фабричные функции наподобие этой:

```
function vote(entry) {
  return {type: 'VOTE', entry};
}
```

Такие функции ещё называют «создателями действий». Они являются чистыми функциями, которые всего лишь возвращают объекты действий. Но при этом таким образом инкапсулируют внутреннюю структуру объектов действий, чтобы она больше не была никак связана с остальной кодовой базой. С помощью создателей действий также удобно документировать все действия, которые могут быть переданы в ваше приложение. Было бы труднее собирать подобную информацию, будь она разбросана по всей кодовой базе в виде объектных литералов.

Создадим новый файл, определяющий создателей действий для двух наших уже существующих клиентских действия:

```
src/action_creators.js
export function setState(state) {
  return {
    type: 'SET_STATE',
    state
  };
}

export function vote(entry) {
  return {
    type: 'VOTE',
    entry
  };
}
```

Для этих функций очень легко писать модульные тесты. Но обычно я этого не делаю, если создатель действия только возвращает объект. Но если хотите, то можете добавить.

Теперь в файле `index.jsx` в Socket.io-обработчике события мы можем использовать создателя действия `setState`:

```
src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import io from 'socket.io-client';
import reducer from './reducer';
import {setState} from './action_creators';
import App from './components/App';
import {VotingContainer} from './components/Voting';
import {ResultsContainer} from './components/Results';

const store = createStore(reducer);

const socket =
io(`${location.protocol}//${location.hostname}:8090`);
socket.on('state', state =>
  store.dispatch(setState(state))
);

const routes = <Route component={App}>
  <Route path="/results" component={ResultsContainer} />
  <Route path="/" component={VotingContainer} />
</Route>;

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);
```

Ещё одно преимущество создателей действий заключается том, как react-redux подключает их к React-компонентам. У нас есть callback-свойство `vote` в `Voting` и создатель действия vote. Имена одинаковые, как и сигнатуры функции: один аргумент, являющийся записью, за которую проголосовали. Так что мы можем просто передать создателя действия в функцию `connect` из react-redux в качестве второго аргумента:

```
src/components/Voting.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';
import {connect} from 'react-redux';
import Winner from './Winner';
import Vote from './Vote';
import * as actionCreators from '../action_creators';

export const Voting = React.createClass({
  mixins: [PureRenderMixin],
  render: function() {
    return <div>
      {this.props.winner ?
        <Winner ref="winner" winner={this.props.winner} /> :
        <Vote {...this.props} />}
    </div>;
  }
});

function mapStateToProps(state) {
  return {
    pair: state.getIn(['vote', 'pair']),
    hasVoted: state.get('hasVoted'),
    winner: state.get('winner')
  };
}

export const VotingContainer = connect(
  mapStateToProps,
  actionCreators
)(Voting);
```

В результате свойство `vote` будет передано в `Voting`. Это свойство представляет собой функцию, создающую действие с помощью создателя vote, и отправляющую это действие в Redux Store. То есть действие будет отправляться при клике на кнопку голосования! Можете сразу проверить это в браузере: после нажатия на кнопку она деактивируется.

5.11. Отправка действий на сервер с помощью Redux Middleware

Теперь нам нужно решить последний вопрос — получения сервером результатов пользовательских действий. Это должно происходить при голосовании и нажатии на кнопку “Next” на экране результатов.

Начнём с голосования. Что у нас уже есть?

- Когда пользователь голосует, создаётся действие `VOTE` и отправляется в клиентский Redux store.
- Действия `VOTE` обрабатываются клиентским reducer-ом в виде назначения свойства `hasVoted`.
- Сервер готов к принятию actions от клиентов посредством Socket.io-событий `action`. Все полученные действия будут передаваться в серверный Redux Store.
- Действия `VOTE` обрабатываются серверным reducer-ом регистрированием голоса и обновлением результатов.

Похоже, у нас есть почти всё, что нужно. Не хватает только

отправки на сервер клиентских действий `VOTE`, чтобы обработать их в *обоих* Redux stores. Этим мы и займёмся.

С чего начать? В Redux нет ничего подходящего для этого, поскольку в число его основных задач не входит поддержка распределённых систем наподобие нашей. Так что будем сами решать, как нам организовать отправку действий на сервер.

Redux предоставляет шаблонный способ подцепления к actions, отправляемым в redux store — Middleware.

Middleware (посредник) — это функция, которая вызывается при передаче действия ещё до того, как это действие попадёт в reducer и store. Middleware можно использовать в разных целях, от логгирования и обработки исключений до модифицирования действий, кэширования результатов и изменения способа и времени попадания действия в store. Мы же воспользуемся этими функциями для отправки клиентских actions на сервер.

Обратите внимание на разницу между middleware и listeners:

- Первые вызываются до того, как действие попадает в store, поэтому они могут повлиять на него.
- Вторые вызываются после выполнения действия, и уже никак не могут повлиять на его судьбу.

Разные инструменты для разных задач.

Создадим remote action middleware, благодаря которому с

помощью Socket.io-подключения действие будет отправлено не только в первоначальный store, но и в удалённый.

Настроим каркас нашего middleware. Это функция, которая берёт Redux store и возвращает другую функцию, принимающую callback «next». Эта другая функция возвращает *третью*, принимающую Redux action. Именно эта последняя функция и отражает реализацию middleware:

```
src/remote_action_middleware.js
export default store => next => action => {

}
```

Вы можете счесть предыдущий код немного странным, но это более конкретный способ записи:

```
export default function(store) {
  return function(next) {
    return function(action) {

    }
  }
}
```

Подобный способ вкладывания одноаргументных функций называется каррированием. В этом случае мы можем легко сконфигурировать посредника: если бы все аргументы содержались в одной функции (`function(store, next, action) { }`), то нам пришлось бы передавать их при каждом использовании посредника. А благодаря каррированию мы можем единожды вызвать первую функцию и получить возвращаемое

значение, которое «помнит», какой `store` нужно использовать.

То же самое относится и к аргументу `next`. Это callback, который должен вызываться middleware по завершении работы, когда нужно передать `action` в store (или следующему middleware):

```
src/remote_action_middleware.js
export default store => next => action => {
  return next(action);
}
```

Посредник может решить *не вызывать* `next`, если сочтёт нужным задержать действие. Тогда оно уже никогда не попадёт в reducer или store.

Давайте что-нибудь залоггируем в middleware, чтобы узнать, когда оно вызывается:

```
src/remote_action_middleware.js
export default store => next => action => {
  console.log('in middleware', action);
  return next(action);
}
```

Если добавить это middleware к нашему Redux store, то все действия будут логироваться. Активировать middleware можно с помощью Redux функции `applyMiddleware`. Она берёт middleware, которое мы хотим зарегистрировать, и возвращает функцию, которая, в свою очередь, берёт функцию `createStore`. Затем эта вторая функция создаёт для нас store с уже включённым

в него middleware:

```
src/components/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import {Router, Route, hashHistory} from 'react-router';
import {createStore, applyMiddleware} from 'redux';
import {Provider} from 'react-redux';
import io from 'socket.io-client';
import reducer from './reducer';
import {setState} from './action_creators';
import remoteActionMiddleware from './remote_action_middleware';
import App from './components/App';
import {VotingContainer} from './components/Voting';
import {ResultsContainer} from './components/Results';

const createStoreWithMiddleware = applyMiddleware(
  remoteActionMiddleware
)(createStore);
const store = createStoreWithMiddleware(reducer);

const socket =
io(`${location.protocol}//${location.hostname}:8090`);
socket.on('state', state =>
  store.dispatch(setState(state))
);

const routes = <Route component={App}>
  <Route path="/results" component={ResultsContainer} />
  <Route path="/" component={VotingContainer} />
</Route>;

ReactDOM.render(
  <Provider store={store}>
    <Router history={hashHistory}>{routes}</Router>
  </Provider>,
  document.getElementById('app')
);
```

Это еще один хороший пример использования каррирования. И его очень активно используют API Redux.

Теперь, если мы перезагрузим приложение, то увидим, что middleware логгирует все происходящие actions: сначала исходное `SET_STATE`, а в ходе голосования — `VOTE`.

Middleware должно отправлять полученное действие в Socket.io-подключение и передавать следующему middleware. Но для начала нужно предоставить ему это подключение. Оно уже есть у нас в `index.jsx`, осталось только дать middleware доступ. Это легко осуществить с помощью ещё одного каррирования в определении middleware. Внешняя функция должна брать сокет Socket.io:

```
src/remote_action_middleware.js
export default socket => store => next => action => {
  console.log('in middleware', action);
  return next(action);
}
```

```
src/index.jsx
const socket =
io(`${location.protocol}//${location.hostname}:8090`);
socket.on('state', state =>
  store.dispatch(setState(state))
);

const createStoreWithMiddleware = applyMiddleware(
  remoteActionMiddleware(socket)
)(createStore);
const store = createStoreWithMiddleware(reducer);
```

Обратите внимание, что нам нужно поменять местами инициализацию сокета и store: сокет должен создаваться первым, поскольку он нам понадобится в ходе инициализации store.

Осталось только, чтобы middleware сгенерировало событие `action`:

```
src/remote_action_middleware.js
export default socket => store => next => action => {
  socket.emit('action', action);
  return next(action);
}
```

Вот и всё! Теперь при клике на одну из кнопок голосования вы увидите в том же окне браузера обновление текущих результатов. То же самое произойдёт и в других браузерах, в которых будет запущено приложение. Голос зарегистрирован!

Но тут есть одна проблема: когда мы получаем с сервера обновление состояния и передаём действие `SET_STATE`, оно также попадает на сервер. И хотя он ничего не делает с этим действием, но всё же его получатель запросов срабатывает, генерируя новое `SET_STATE`. Получается бесконечный цикл.

Middleware удалённого action не должен отправлять на сервер каждое действие. Некоторые из них, вроде SET_STATE, должны обрабатываться локально, на клиенте. Пусть посредник отправляет на сервер только конкретные действия, содержащие свойство `{meta: {remote: true}}`:

(этот подход взят из примера rafScheduler из документации middleware)

```
src/remote_action_middleware.js
export default socket => store => next => action => {
  if (action.meta && action.meta.remote) {
    socket.emit('action', action);
  }
  return next(action);
}
```

Создатель действия должен назначать это свойство для `VOTE`, и *не должен* для `SET_STATE`:

```
src/action_creators.js
export function setState(state) {
  return {
    type: 'SET_STATE',
    state
  };
}

export function vote(entry) {
  return {
    meta: {remote: true},
    type: 'VOTE',
    entry
  };
}
```

Резюмируем происходящее:

1. Пользователь кликает кнопку голосования. Передаётся действие VOTE.
2. Middleware удалённого действия отправляет action через Socket.io-подключение.
3. Действие обрабатывается клиентским Redux store, в результате чего назначается локальное состояние `hasVote`.

4. Когда сообщение приходит на сервер, то серверный Redux store обрабатывает action и обновляет голос в текущих результатах.
5. Получатель запросов в серверном store транслирует снэпшот состояния всем подключённым клиентам.
6. В Redux store каждого из подключённых клиентов передаются действия `SET_STATE`.
7. Они обрабатываются хранилищами на основе обновленного сервером состояния.

Для завершения нашего приложения осталось только заставить работать кнопку “Next”. На сервере уже есть необходимая логика, как и в модуле голосования. Осталось только соединить их друг с другом.

Создатель действия для `NEXT` должен создавать удалённое действие правильного типа:

```
src/action_creator.js
export function setState(state) {
  return {
    type: 'SET_STATE',
    state
  };
}

export function vote(entry) {
  return {
    meta: {remote: true},
    type: 'VOTE',
    entry
  };
}

export function next() {
  return {
    meta: {remote: true},
```

```
    type: 'NEXT'
  };
}
```

Создатели действий подключаются в виде свойств к компоненту `ResultsContainer`:

```
src/components/Results.jsx
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';
import {connect} from 'react-redux';
import Winner from './Winner';
import * as actionCreators from '../action_creators';

export const Results = React.createClass({
  mixins: [PureRenderMixin],
  getPair: function() {
    return this.props.pair || [];
  },
  getVotes: function(entry) {
    if (this.props.tally && this.props.tally.has(entry)) {
      return this.props.tally.get(entry);
    }
    return 0;
  },
  render: function() {
    return this.props.winner ?
      <Winner ref="winner" winner={this.props.winner} /> :
      <div className="results">
        <div className="tally">
          {this.getPair().map(entry =>
            <div key={entry} className="entry">
              <h1>{entry}</h1>
              <div className="voteCount">
                {this.getVotes(entry)}
              </div>
            </div>
          )}
        </div>
        <div className="management">
          <button ref="next"
                  className="next"
                  onClick={this.props.next}>
            Next
```

```
          </button>
        </div>
      </div>;
  }
});

function mapStateToProps(state) {
  return {
    pair: state.getIn(['vote', 'pair']),
    tally: state.getIn(['vote', 'tally']),
    winner: state.get('winner')
  }
}

export const ResultsContainer = connect(
  mapStateToProps,
  actionCreators
)(Results);
```

Ну… вот и всё! Теперь наше приложение полностью готово и функционирует. Попробуйте открыть экран результатов на компьютере и экран голосования на мобильном устройстве. Вы увидите, что после выполнения действия на одном устройстве результат сразу отображается на другом. Волшебное зрелище. Нажмите на кнопку «Next» на экране результатов и посмотрите, как продвигается голосование на другом устройстве.

6. Упражнения

Если вы хотите улучшить это приложение, а заодно получше познакомиться с архитектурой Redux, то можете выполнить несколько упражнений. Под каждым из них есть ссылка на одно из возможных решений.

1. Неправильное предотвращение голосования
Если запись не входит текущую пару, то сервер не должен

позволять голосовать за неё. Добавьте провальный модульный тест для иллюстрации проблемы, и исправьте её.

Моё решение.

2. Улучшение сброса состояния голосования
Если в новой паре нет записи, которая уже участвовала в голосовании, то клиент сбрасывает свойство hasVoted. Но тут есть проблема: в последних раундах голосования возникают ситуации, когда в двух следующих друг за другом парах есть одинаковая запись, и тогда свойство не сбрасывается. В последнем раунде пользователь не может проголосовать, потому что кнопки неактивны.

Модифицируйте систему так, чтобы она создавала уникальный идентификатор для каждого раунда, а состояние голосования отслеживало бы этот идентификатор.

Подсказка: Можно отслеживать счётчик раундов на сервере. Когда пользователь отдаёт голос, сохраняйте на клиенте номер раунда. При обновлении общего состояния сбрасывайте состояние голосования, если на сервере изменился номер раунда.

Моё решение на сервере и на клиенте.

3. Предотвращение повторного голосования
В течение одного раунда пользователь всё ещё может проголосовать несколько раз. Достаточно обновить страницу, и состояние голосования теряется. Исправьте это.

Подсказка: Можно генерировать уникальные идентификаторы для каждого пользователя, чтобы сервер мог знать, кто и за что проголосовал. Так что если пользователь снова проголосует, то его предыдущий голос в этом раунде обнуляется. Тогда вы можете не блокировать кнопки голосования, это позволит пользователям менять своё мнение в течение раунда.

Моё решение на сервере и на клиенте.

4. Перезапуск голосования
Создайте на экране голосования кнопку, позволяющую пользователю запустить голосование с самого начала.

Подсказка: Вам нужно отслеживать в состоянии порядок следования записей, чтобы при сбросе вернуться к началу.

Моё решение на сервере и на клиенте.

5. Отображение состояния подключения сокета
При неустойчивой связи подключение к Socket.io может прерываться. Добавьте визуальный индикатор, который сообщал бы пользователю об отсутствии с сервером.

Подсказка: Прослушивайте сетевые сообщения от Socket.io и передавайте действия, которые будут класть в Redux-хранилище состояние подключения.

Моё решение.

Бонусный вызов: переход в одноранговый режим (Peer to Peer) Сам я это сделать не пробовал, но тема интересная, если вы любите исследовать. Модифицируйте логику системы таким образом, чтобы вместо отдельных реализаций reducer-ов на клиенте и сервере применялся reducer с полной логикой, исполняемый на каждом клиенте. Передавайте все действия каждому участнику, чтобы все видели одно и то же.

Как можно удостовериться, что каждый клиент получает все передаваемые действия, да ещё и в нужном порядке? Нужен ли будет в этом случае сервер? Сможете ли вы перейти на полноценный P2P с помощью WebRTC? (или Socket.io P2P)

Похожие публикации

Как выбрать язык программирования?

gobwas • 17 декабря 2015 в 12:57

10 основных ошибок при разработке на Node.js 45

ПЕРЕВОД

ZaValera • 16 апреля 2015 в 18:42

Fast app resume для Windows Phone 8 11

Atreides07 • 7 марта 2014 в 11:49

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке 14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада 4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе 25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации 2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

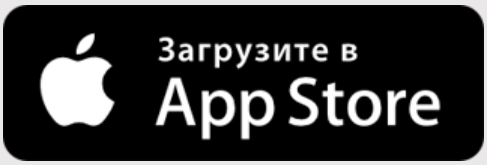
Стивен Хокинг и его работа — что дал ученый человечеству? 8

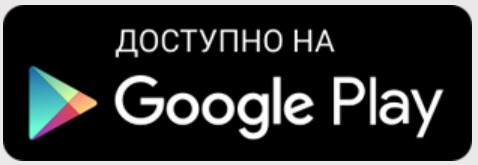
marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения 17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия