Урок 7

Концепция Flux на примере использования Redux

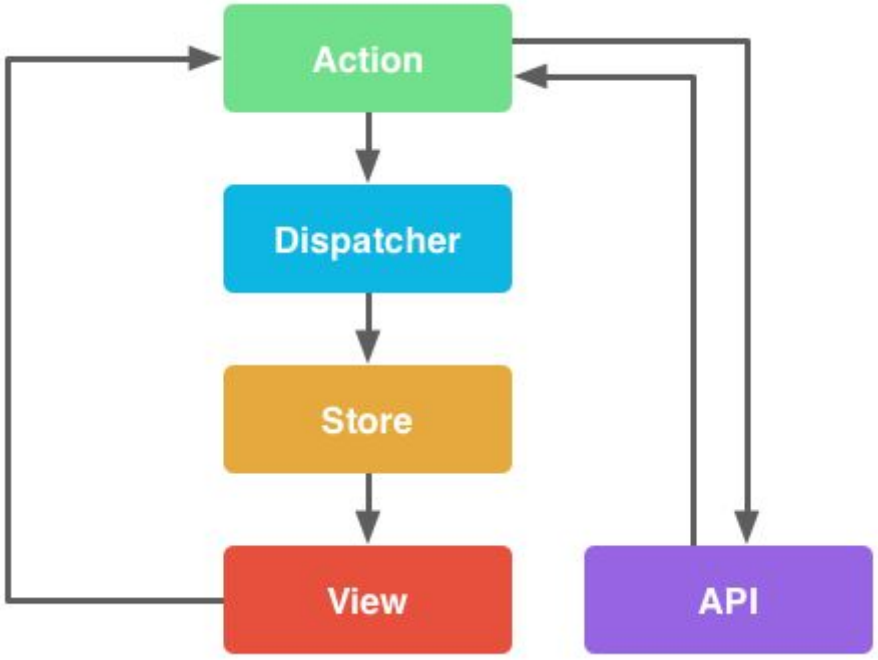
Установка redux в приложение

Что такое Flux

Flux — это архитектура, которую команда Facebook использует при работе с React. Это не фреймворк, или библиотека, это новый архитектурный подход, который дополняет React и принцип однонаправленного потока данных.
Тем не менее, Facebook предоставляет репозиторий, который содержит реализацию Dispatcher. Диспетчер играет роль глобального посредника в шаблоне «Издатель-подписчик» (Pub/sub) и рассылает полезную нагрузку зарегистрированным обработчикам.Типичная реализация архитектуры Flux может использовать эту библиотеку вместе с классом EventEmitter из NodeJS, чтобы построить событийно-ориентированную систему, которая поможет управлять состоянием приложения.
Flux легче всего объяснить, исходя из составляющих его компонентов:

- Actions / Действия — хелперы, упрощающие передачу данных Диспетчеру;
- Dispatcher / Диспетчер — принимает Действия и рассылает нагрузку зарегистрированным обработчикам;
- Stores / Хранилища — контейнеры для состояния приложения и бизнес-логики в обработчиках, зарегистрированных в Диспетчере;
- ControllerViews / Представления — React-компоненты, которые собирают состояние хранилищ и передают его дочерним компонентам через свойства.

Схематично это изобразить можно следующим образом:



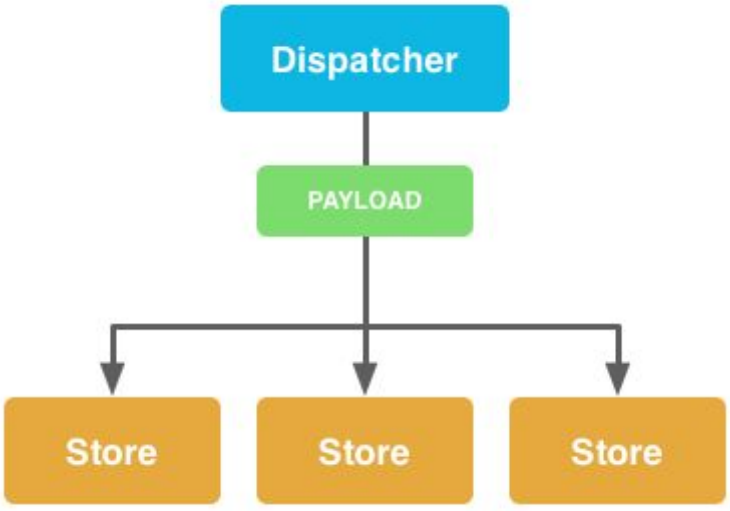
FluxDispatcher

Диспетчер — это менеджер всего этого процесса. Это центральный узел приложения. Диспетчер получает на вход действия и рассылает эти действия (и связанные с ними данные) зарегистрированным обработчикам.
Диспетчер рассылает данные всем зарегистрированным в нём обработчикам и позволяет вызывать обработчики в определённом порядке, даже ожидать обновлений перед тем, как продолжить работу. Есть только один Диспетчер, и он действует как центральный узел всего вашего приложения.
Пример синтаксиса:

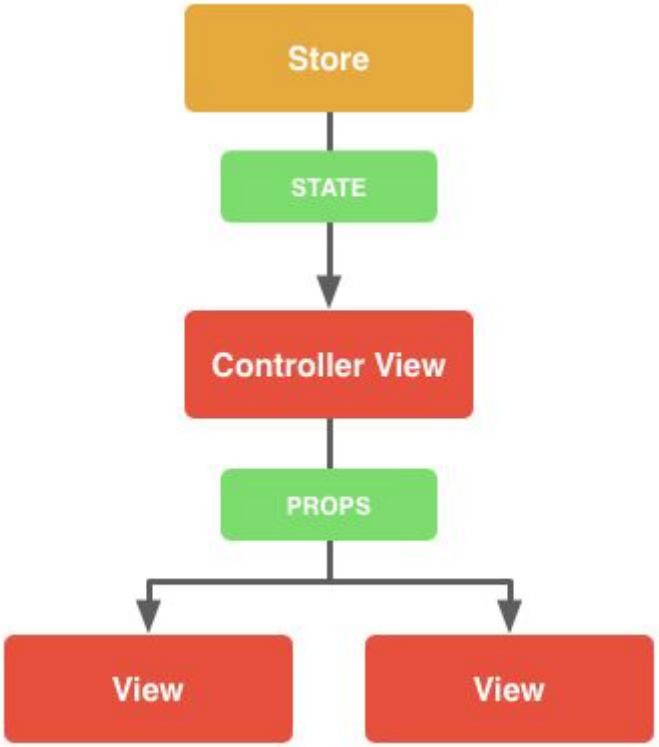
```
var Dispatcher = require('flux').Dispatcher;
varAppDispatcher = new Dispatcher();

AppDispatcher.handleViewAction = function(action) {
this.dispatch({
   source: 'VIEW_ACTION',
   action: action
 });
}
module.exports = AppDispatcher;
```

В примере создаётся экземпляр Диспетчера и метод handleViewAction. Эта абстракция полезна, если вы собираетесь разделять действия, созданные в интерфейсе и действия, пришедшие от сервера / API. Этот метод вызывает метод dispatch, который уже рассылает данные action всем зарегистрированным в нём обработчикам. Это действие затем может быть обработано Хранилищами, в результате чего состояние приложения будет обновлено. Схематично это изобразить можно так:



Объекты представлены в виде Представлений. Представления — это всего лишь React-компоненты, которые подписаны на событие «change» и получают состояние приложения из Хранилищ. Далее они передают эти данные дочерним компонентам через свойства.



Пример синтаксиса:

```
/** @jsxReact.DOM */
var React = require('react');
varShoesStore = require('../stores/ShoeStore');
// Метод для получения состояния приложения из хранилища
functiongetAppState() {
return {
    shoes: ShoeStore.getShoes()
  };
}
// Создаём React-компонент
varShoeStoreApp = React.createClass({
// Используем метод getAppState, чтобы установить начальное состояние
getInitialState: function() {
returngetAppState();
  },
// Подписываемся на обновления
componentDidMount: function() {
ShoeStore.addChangeListener(this._onChange);
  },
// Отписываемся от обновлений
componentWillUnmount: function() {
ShoesStore.removeChangeListener(this._onChange);
  },
  render: function() {
return (
<ShoeStoreshoes={this.state.shoes} />
);
  },
// Обновляем состояние Представления в ответ на событие "change"
_onChange: function() {
this.setState(getAppState());
}
});
module.exports = ShoeStoreApp;
```

Состояние приложения хранится в Хранилищах, поэтому используются интерфейс Хранилищ, чтобы получить эти данные, а затем обновить состояние компонентов.

EventEmitter

EventEmitter(EE) представляет собой основной объект, реализующий работу с событиями в Node.js. Для того чтобы воспользоваться EventEmitter, достаточно подключить встроенный модуль «events» и взять из него соответствующее свойство (создадим ee.js) :

```
let EventEmitter = require('events').EventEmitter;
```

После можно создавать новый объект:

```
let server = new EventEmitter;
```

EventEmitter содержит методы для работы с событиями.

1) Подписка:

```
server.on('request', function(request) {
  request.approved = true;
});
```

on – имя события, function – обработчик. Подписчиков можно вызвать много, и все они будут вызваны в том же порядке, в котором назначены.

2) emit:

```
server.emit('request', {from: "Client"});
server.emit('request', {from: "Another Client"});
```

Генерирует события и передаёт данные. Эти данные попадают в функцию обработчика.

Если браузерные обработчики срабатывают в произвольном порядке, то Node-обработчики точно в том порядке, в котором были назначены. Если есть какие-то обработчики, то назначая следующие, я точно уверен, он сработает после предыдущих.

Ещё одно отличие в том, что в браузере невозможно получить список обработчиков, в которых назначен определенный элемент. А в Node.js это сделать легко: emitter.listeners(eventName) возвращает все обработчики на данное событие. А emitter.listenerCount(eventName) позволяет получить их общее количество.

Store

Store-хранилища в Flux управляют состоянием определённых частей предметной области вашего приложения. На более высоком уровне это означает, что Хранилища хранят данные, методы получения этих данных и зарегистрированные в Диспетчере обработчики Действий. Пример синтаксиса хранилища:

```
varAppDispatcher = require('../dispatcher/AppDispatcher');
varShoeConstants = require('../constants/ShoeConstants');
varEventEmitter = require('events').EventEmitter;
var merge = require('react/lib/merge');
// Внутренний объект для хранения shoes
var _shoes = {};
// Метод для загрузки shoes из данных Действия
functionloadShoes(data) {
  _shoes = data.shoes;
}
// Добавить возможности EventEmitter из Node
varShoeStore = merge(EventEmitter.prototype, {
// Вернуть все shoes
getShoes: function() {
return _shoes;
  },
emitChange: function() {
this.emit('change');
  },
addChangeListener: function(callback) {
this.on('change', callback);
  },
removeChangeListener: function(callback) {
this.removeListener('change', callback);
  }
});
// Зарегистрировать обработчик в Диспетчере
AppDispatcher.register(function(payload) {
var action = payload.action;
var text;
// Обработать Действие в зависимости от его типа
switch(action.actionType) {
caseShoeConstants.LOAD_SHOES:
// Вызвать внутренний метод на основании полученного Действия
loadShoes(action.data);
break;
default:
returntrue;
  }
// Если Действие было обработано, создать событие "change"
ShoeStore.emitChange();
returntrue;
});
module.exports = ShoeStore;
```

Для создания хранилища в ES6 будем использовать createStore().

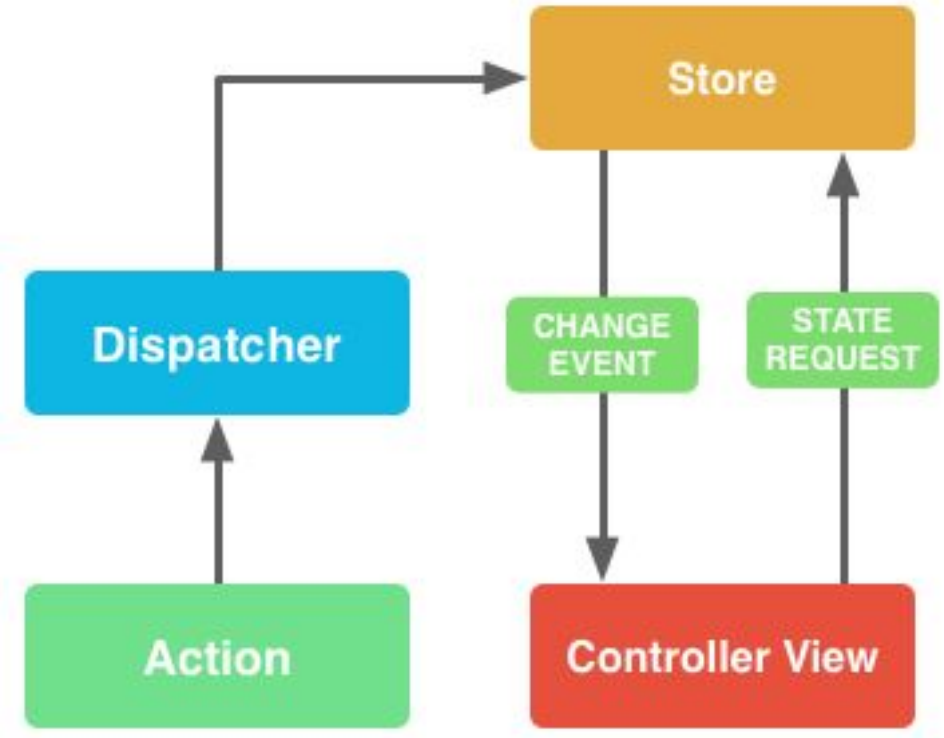
```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp)
```

При необходимости можно указать начальное состояние в качестве второго аргумента createStore()

```
let store = createStore(todoApp, window.STATE_FROM_SERVER)
```

К хранилищу добавлены возможности EventEmitter из NodeJS. Это позволяет хранилищам слушать и рассылать события, что, в свою очередь, позволяет компонентам представления обновляться, отталкиваясь от этих событий. Так как наше представление слушает событие «change», создаваемое Хранилищами, оно узнаёт о том, что состояние приложения изменилось, и пора получить (и отобразить) актуальное состояние.

Выполнена регистрация обработчика в AppDispatcher с помощью его метода register. Это означает, что теперь Хранилище слушает оповещения от AppDispatcher. Исходя из полученных данных, оператор switch решает, можем ли мы обработать Действие. Если действие было обработано, создаётся событие «change», и Представления, подписавшиеся на это событие, реагируют на него обновлением своего состояния:



Представление использует метод getShoes интерфейса Хранилища для того, чтобы получить все shoes из внутреннего объекта shoes и передать эти данные в компоненты.

Что такое Redux и для чего он нам нужен

Redux — это инструмент управления как состоянием данных, так и состоянием интерфейса в JavaScript-приложениях. Он подходит для одностраничных приложений, в которых управление состоянием может со временем становиться сложным. Redux не связан с каким-то определ`нным фреймворком, и хотя разрабатывался для React, может использоваться с Angular или jQuery.

Все данные в React связаны с компонентами. При этом не очевидно, как два компонента, не связанные отношением родитель-ребенок, будут взаимодействовать между собой.

В React не рекомендуется реализовывать прямое взаимодействие компонент-компонент. Для взаимодействия между двумя компонентами, не имеющими связи родитель-ребенок, вы можете настроить глобальную систему событий. Flux — это один из способов достижения этого.

Позволяет вам создавать приложения, которые ведут себя одинаково в различных окружениях (клиент, сервер и нативные приложения), а также просто тестируются. Кроме того, это обеспечивает большой опыт отладки, например, редактирование кода в реальном времени в сочетании с time traveling.

Использовать Redux можно с любой другой view-библиотекой. Это крошечная библиотека (2kB, включая зависимости).

Для установки:

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

Для связки с React понадобится подключить следующие инструменты:

```
npm install --save react-redux
npm install --save-dev redux-devtools
```

Предполагается, что вы используете пакетный менеджер npm вместе со сборщиком модулей, типа Webpack или Browserify, использующие CommonJS модули.

Структура проекта:

Приведён пример классического разделения структуры проекта

1. actions отвечает за хранение всех action creators проекта;
2. components — все react компоненты;
3. constants — константы. Важно, что в эту папку мы собираем не только константы наподобие ENTER_KEY = 13, но и различные названия actions, например:

```
export const FETCH_ACCOUNT = 'FETCH_ACCOUNT';
export const RECEIVE_ACCOUNT = 'RECEIVE_ACCOUNT';
```

4. containers — специфичная директория. Состоит из страничных компонентов, которые при необходимости (например, недостатке данных) вызывают action creators и отправляют компоненты на рендеринг.
5. reducers — ничего интересного, просто управление стейтом приложения. Здесь же хранится “корневой” редьюсер, объединяющий все остальные (с помощью combineReducers);
6. store — конфиг стора, его инициализация;
7. index.js — точка входа в наше приложение. Инициализирует react, описывает роутинг, создает стор и т.п.;

С такой структурой удобно работать и начинать разработку проекта. Далее мы можем выбирать определенную модель приложения и продумать необходимые модули.

Reducers

Для подключения в командной строке необходимо прописать следующее:

```
npm install --save reduce-reducers
```

Пример использования:

```
const reducer = reduceReducers(
  (prev, curr) => ({...prev, A: prev.A + curr}),
  (prev, curr) => ({...prev, B: prev.B * curr}),
);
expect(reducer({ A: 1, B: 2 }, 3)).to.deep.equal({ A: 4, B: 6 });
expect(reducer({ A: 5, B: 8 }, 13)).to.deep.equal({ A: 18, B: 104 });
```

Изначально reducers созданы для объединения нескольких Redux, которые соответствуют различным действиям. Технически можно выполнять это с любыми reducers.

Redux EventEmitter и Store

Redux предлагает хранить все состояния приложения в одном месте, называемом «store» («хранилище»). Компоненты «отправляют» изменение состояния в хранилище, а не напрямую другим компонентам. Компоненты, которые должны быть в курсе этих изменений, «подписываются» на хранилище.

Хранилище может рассматриваться как «посредник» во всех изменениях состояния в приложении. С Redux компоненты не связываются друг с другом напрямую, все изменения должны пройти через единственный источник истины, через хранилище.

Практика

Задача 1. Пример использование Redux

В примере приведен код Redux, распространяющийся на многие задачки:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Redux basic example</title>
    <script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
  </head>
  <body>
    <div>
      <p>
        Clicked: <span id="value">0</span> times
        <button id="increment">+</button>
        <button id="decrement">-</button>
```

```
      <button id="incrementIfOdd">Increment if odd</button>
      <button id="incrementAsync">Increment async</button>
    </p>
  </div>
  <script>
    function counter(state, action) {
      if (typeof state === 'undefined') {
        return 0
      }
      switch (action.type) {
        case 'INCREMENT':
          return state + 1
        case 'DECREMENT':
          return state - 1
        default:
          return state
      }
    }
    var store = Redux.createStore(counter)
    var valueEl = document.getElementById('value')
    function render() {
      valueEl.innerHTML = store.getState().toString()
    }
    render()
    store.subscribe(render)
    document.getElementById('increment')
      .addEventListener('click', function () {
        store.dispatch({ type: 'INCREMENT' })
      })
    document.getElementById('decrement')
      .addEventListener('click', function () {
        store.dispatch({ type: 'DECREMENT' })
      })
    document.getElementById('incrementIfOdd')
      .addEventListener('click', function () {
        if (store.getState() % 2 !== 0) {
          store.dispatch({ type: 'INCREMENT' })}
      })
    document.getElementById('incrementAsync')
      .addEventListener('click', function () {
        setTimeout(function () {
          store.dispatch({ type: 'INCREMENT' })
        }, 1000) })
  </script>
 </body>
</html>
```

Домашнее задание

Необходимо для Вашего react приложения настроить для работы readux.

1. Для сохранения состояния необходимо реализовать хранилище (store), которое представляет собой комбинацию редьюсеров (reducers), объединенных методов combineReducers. Магазин создаётся при помощи функции createStore(reducer, state, middleware) (см. проект с урока).
2. Для разделения ответственности по обработке частей состояния, необходимо для каждой части состояния создать свой редьюсер. Каждый редьюсер - это функция, которая принимает 2 аргумента: state и action, где state - текущее состояние, которое требуется изменить, action - действие, которое диктует правила изменения этого состояния. При этом результатом работы функции редьюсера - новое состояние, которое эта функция и возвращает. Не забывайте главное правило - состояния иммутабельно, поэтому мы должны всегда порождать новое состояния на основании старого. При этом необходимо производить клонирование тех элементов, которые являются ссылочными, например, массивы, обычно для них хватает вызова функции slice(0). Эти редьюсеры должны быть скомбинированы при создании магазина в один редьюсер. При этом так же необходимо в редьюсре задать для состояния начальный вид, так как изначально state - undefined
3. Для всех требуемых действий необходимо создать соответствующие функции - actions, который возвращают либо объект, который будет передан диспетчеру, либо функцию (в случае асинхронной обработки), которая принимает входной параметр dispatch - функция для асинхронной диспетчеризации.
4. Для увязки нашего приложения с состоянием хранилища, необходимо на главной странице, где отрисовывается основной элемент нашего приложения, обернуть все роуты в элемент - Provider и задать его свойство store = { сюда мы помещаем экземпляр нашего хранилища из п. 2 }
5. При работе с компонентами нашего приложения, которые будет изменять или обращаться к состоянию, мы должны использовать функцию connect из react-redux пакета. Эта функция накладывается на компонент в качестве декоратора (аля @connect()). Первым и самым важным ее параметром является функция, которая принимает наше хранилище (store) и должна вернуть объект. Этот объект, а точнее его свойства, будут записаны в this.props нашего компонента. В том числе в this.props.dispatch будет записана функция диспетчеризации. Вся работа по изменению состояния ведется через нее и вызовы соответствующих действий, который порождают требуемый для диспетчеризации объект (объект или функцию). Более подробно можно почитать по ссылкам.
6. Если нужно, то также можно сделать свои middleware и подключить их в процесс обработки изменения состояния

Задание:

1. Настроить приложение в соответствии с п.1 - п.6 руководства по использованию redux
2. Реализовать страницу ленты блогов с применением redux (http://jsonplaceholder.typicode.com/posts)
3. Для пользователей нашего блога, требуется загружать их данные из интернета по ссылке (jsonplaceholder.typicode.com/users)
4. *При клике на пользователя должен отображаться список его постов (http://jsonplaceholder.typicode.com/posts?userId=1 у которых userId равен id пользователя). Попробуйте реализовать это в виде split view (то есть слева пользователи справа их посты)
5. *При клике на каждый пост должны показываться комментарии к посту (http://jsonplaceholder.typicode.com/posts/1/comments)

Если не хотите или не получается сделать вариант с загрузкой из интернета, сделайте json-файлы, которые будут загружаться в state нашего приложения вместо загрузки (либо сразу, то есть в начальном состоянии, либо при помощи считывания при обработке action в редьюсере).

Для работы Вам потребуются следующие npm пакеты:

redux - https://www.npmjs.com/package/redux(непосредственно redux с его функциями createStore, combineReduxers и applyMiddleware)

react-redux - https://www.npmjs.com/package/react-redux(добавляем элемент Provider в наше приложение, а также функцию connect)

redux-thunk - https://www.npmjs.com/package/redux-thunk(middleware для возможности диспетчиризации функций)

redux-logger - https://www.npmjs.com/package/redux-logger(middleware для логирования состояний)

redux-promise-middleware -https://www.npmjs.com/package/redux-promise-middleware(middleware для создания асинхронных диспетчеризируемых функций)

идет в секцию пакетов devDependencies

babel-transform-decorators-legacy
-https://www.npmjs.com/package/babel-plugin-transform-decorators-legacy (для возможности обрабатывать декораторы при сборке babel)

Задачи со * повышенной сложности.

Дополнительные материалы

1. https://facebook.github.io/react/index.html
2. https://www.gitbook.com/book/maxfarseer/react-course-ru/details
3. https://learn.javascript.ru/screencast/webpack
4. https://habrahabr.ru/post/269831/
5. http://redux.js.org/docs/basics/
6. https://github.com/reactjs/react-redux
7. https://www.gitbook.com/@maxfarseer
8. https://www.youtube.com/playlist?list=PLoYCgNOIyGADILc3iUJzygCqC8Tt3bRXt

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://facebook.github.io/react/index.html
2. https://www.gitbook.com/book/maxfarseer/react-course-ru/details
3. https://learn.javascript.ru/screencast/webpack