

# Структура урока:

---

- Обзор
- Проблемы управления состоянием
- Что такое Flux?
- Поток данных во Flux
- Итоги

## Обзор

Привет 🖐️😌, друг, и добро пожаловать на первый урок курса по Redux от Lectrum. Нашей целью является изучить что такое состояние приложения, зачем оно нам надо и как с ним работать, исследовать почему управление состоянием приложения это то о чем вы должны задуматься, так же мы изучим решение долгосрочных проблем в отношении `Flux`. В итоге, мы проведем пробный полет через экосистему `Redux` что бы приготовить свои руки к следующему этапу этого курса, в котором мы совершим погружение в глубины `Redux` 💪.

## Проблемы управления состоянием

Давайте начнем по порядку. Что такое состояние приложения? Состояние приложения — это форма данных, которая используется чтобы описать логику приложения которая должна быть отображена в UI 🖥️.

Давайте представим себе легкий пример — `book-reader` 📖 web-приложения с базовым функционалом изменяющим текущую страницу. Как только приложение загрузится первоначально, оно будет иметь несколько основных сущностей:

- Значение `current page` ;
- Значение `total pages` ;
- Значений `title` книги.

🔍 Заметка:

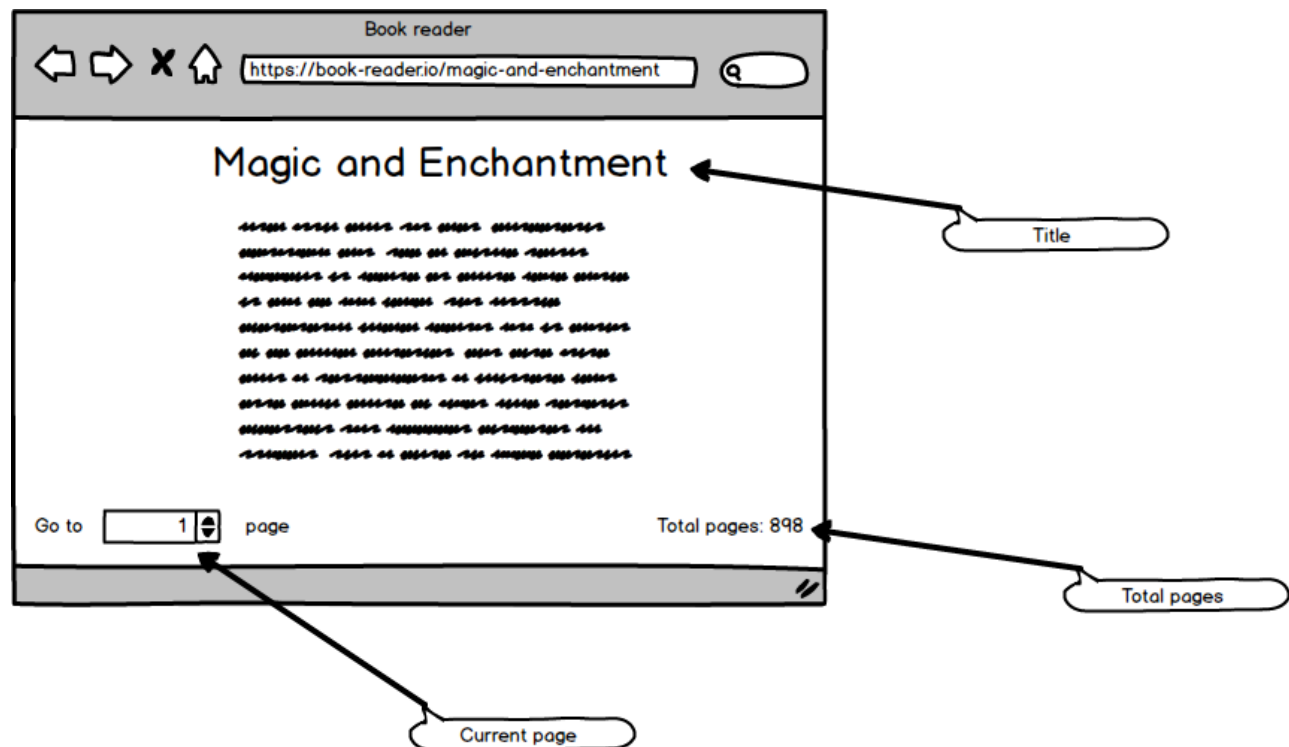
В реальном мире существует гораздо больше управляемых состояний. Этих трех значений достаточно для нашей цели.

С момента инициализации приложения `current value` будет `1`. Значение `total pages` будет относиться к заглавной странице книги, которую читает пользователь.

Давайте посмотрим что происходит с состоянием приложения, когда пользователь переворачивает страницу. `Оскар` здесь чтобы помочь нам в наших исследованиях.

Вы спрашиваете кто это такой, `Оскар`? `Оскар` это талантливый разработчик из Скандинавии, который изучает создание волшебных заклинаний, как сварить волшебное зелье, а так же другие интересные оккультические трюки. И между делом, `Оскар` является создателем `book-reader` 📖 web- приложения. И конечно же, `Оскар` обожает программирование.

Вот так ведет себя состояние `book-reader` приложения, когда `Оскар` меняет `current page` `Magic and Enchantment` книги, которую он очень любит читать:



- Как только приложение загружено — `current page` равно `1` и первая страничка отображается в UI;
- `Оскар` хочет выполнить операцию `CHANGE_PAGE` по нажатию на элемент `<select>` (после `Go to`) и выбрать нужную страницу (скажем `5`);
- Операция `CHANGE_PAGE` выполняется немедленно;
- Состояние приложения отвечающее за эту операцию меняет значение `current page` на `5`;
- Состояние приложения информирует `view`, о том что состояние было изменено;
- В результате `view` вызывает обновление UI.

Все, что указано выше является описанием состояния приложения на протяжении его жизни. В каждом моменте всего состояния приложения данные должны быть сохранёнными где-либо, защищены, быть стабильными, быть доступными и легко управляемыми 🛡️.

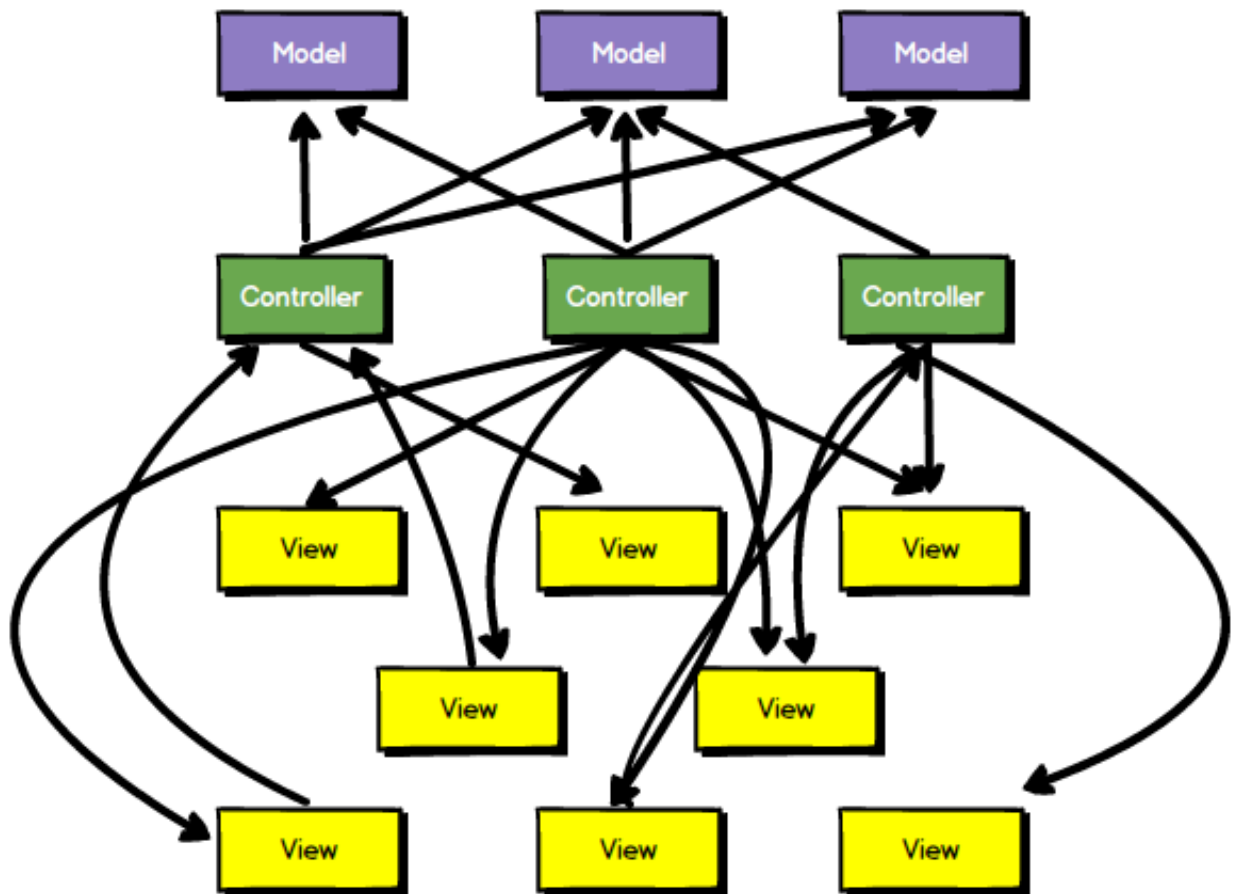
Однако, в реальном мире приложения намного более запутанные, и необходимость управления более сложными соединениями и отношениями состояний становится более, намного более, трудной 😞.

Традиционный подход `mvc` работает неплохо, пока не появляются вторая, третья и более моделей. Подключение их логики к одному или более контролерам, которые в свою очередь присоединяются к другим моделям или видам и так далее и тому подобное...

---

## The MVC problem

---



---

В тот же момент станет почти невозможным тестировать, поддерживать, расширять или даже понять что происходит внутри приложения. И это еще цветочки. Настоящий ад начнётся, когда количество контроллеров станет расти 😓.

Эта проблема начинает возникать за долго до этого. Более того растущая день за днём серьёзность проблемы, становится еще более пугающей чем была раньше, беря во внимание безумный темп эволюции интернет технологий.

Хорошего решения не было, пока 6-го мая, 2014 Facebook не вынес на публику что-то, что изменило целую индустрию.

Этим чем-то был Flux — способ управления моделями данных 🙌.

## Что такое Flux?

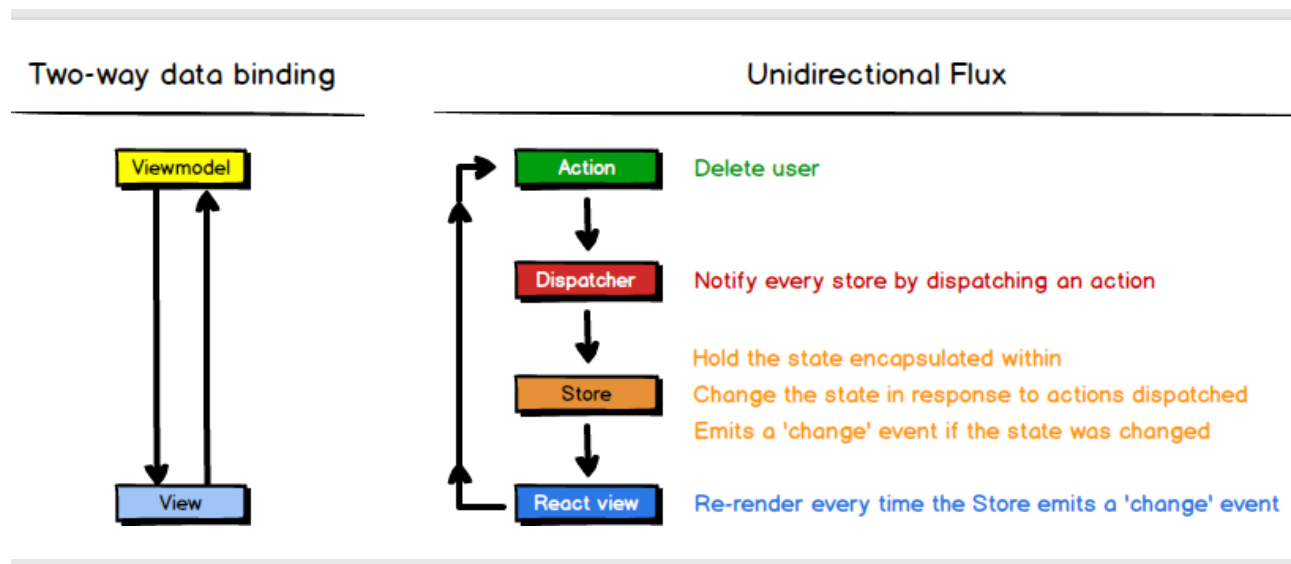
Проблема MVC проста: существует возможность воплотить в жизнь двунаправленную привязку из любой точки приложения свободно и без ограничений.

🔍 Заметка:

Двунаправленная привязка это когда две сущности привязаны ссылкой друг к другу и каждая сущность должна обновить данные, всякий раз когда обновляется одна из сущностей.

Более того, приложение может использовать другие шаблоны в одно и тоже время с MVC, без каких-либо ограничений! Что приведет к ужасной логической связи 😞.

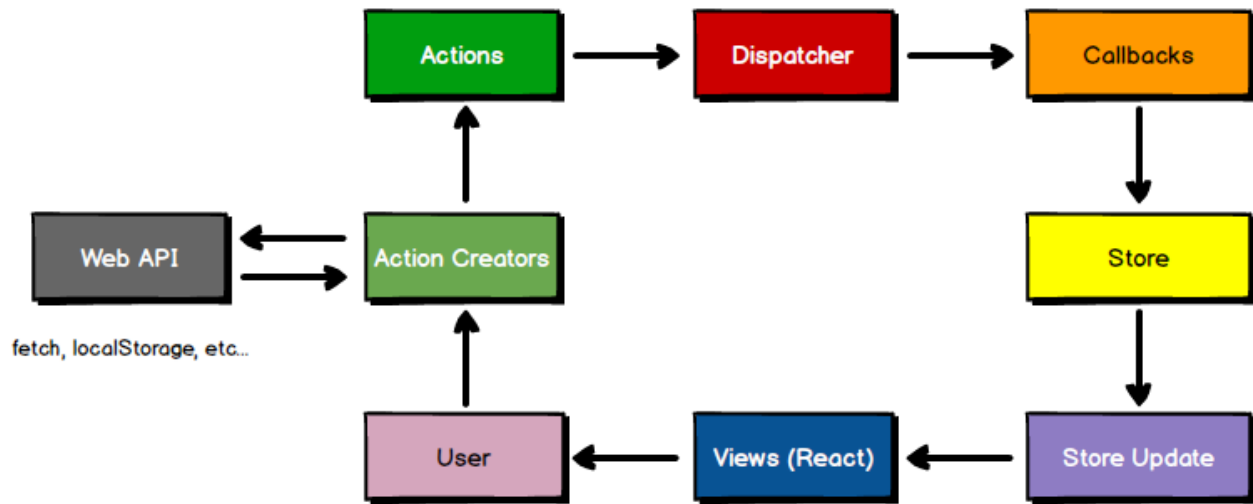
Инженеры Facebook 👨‍💻 задавались вопросом, как создать шаблон управления данными, чтобы решить проблему MVC. Так что в конечном счете они поняли, что ограничения могут прийти на выручку.



Иногда хорошее решение предотвращения чего-то плохого — ограничение возникшей проблемы.

Flux — это архитектура приложения, которая использует строгое ограничение в виде unidirectional data flow (или one way data flow). Подход one way data flow описывает схему, когда изменение данных любое может быть представлено только в одном направлении. Не существует коммуникации two-way в подходе one way data flow. И это основа Flux.

# Data lifecycle in Flux



В `Flux` любое событие, которое вызывается пользователем из UI представляется как специальная сущность называемая `action`. `Action` — это простой JavaScript объект, который имеет обязательное свойство `type` и другой набор свойств, которые описывают данные относящиеся к типичному `action`. Свойство `type` помогает определить какой `action` был вызван. Это либо `LOG_IN action`, либо `SUBMIT_FORM action` или что-либо другое.

К примеру, в `Spellbook`, который мы недавно исследовали вместе с `Оскар`ом, `action.type` будет равно `CHANGE_PAGE`, потому что `action.type` — это буквально описание того, что `Оскар`у необходимо сделать. Свойство `type` обязательное, но определение остальных свойств объекта `action` полностью зависит от вашего решения.

Так же существует неофициальный стандарт `action` - `FSA` (`Flux Standard Action`).

Этот стандарт принуждает следовать единственному принципу сохранения `единую форму` для каждого `action`:

Действие **ДОЛЖНО**:

- Быть `обычным объектом JavaScript`;
- иметь свойство `type`.

Действие **МОЖЕТ**:

- иметь свойство `error`;
- иметь свойство `payload`;
- иметь свойство `meta`.

В действие НЕ ДОЛЖНЫ быть включены другие свойства отличные от `type`, `payload`, `error`, и `meta`.

✅ Хорошая практика:

```
{
  type: 'LOG_IN',
  payload: {
    userID: 'asdfu124AS&F'
  },
  meta: {
    premiumUser: true
  }
}
```

✅ Так же хорошая практика:

```
{
  type: 'FORM_SUBMIT_FAIL',
  payload: new Error(),
  error: true
}
```

❌ Плохая практика:

```
{
  description: 'FETCH_DATA_FROM_REMOTE_SERVER',
  data: [/* data.... */]
  additional: { /* additional information */ }
}
```

👉 Совет:

Навсегда запомните, свойство `type` нужно держать в объекте действия, и быть всегда последовательными в стилистике написания кода 👉. Следование `FSA` является хорошим стартом, но не обязательным.

Существует множество полезных библиотек следующих `FSA`, потому что это дает возможность совершенствовать инфраструктуру проекта следуя простым принципам. `FSA` так же может быть применен в `Redux`, но об этом позже.

Существует два способа создания `action`:

- используя простой объект;

- используя специальную функцию `action creator` которая `возвращает обычный action object`.

`action creator` называется так легко потому что буквально означает `creates an action`.

```
const logIn = (user) => ({
  type: 'LOG_IN',
  payload: user
});
```

Функция `logIn` – является `action creator`, потому что возвращает `(creates)` `action`.

👉 Совет:

Вы можете определять действия, используя любой подход, который лучше вам подходит —объект или создатель действия. Так же создатель действия предоставляет удобную обертку вокруг объекта действия, что помогает упорядочить ваш код, а так же может представлять интерес для пытливых разработчиков.

`Flux` может быть использован с любым инструментом UI или даже вообще без него! Однако по своей сути `Flux` был спроектирован специально для `React`.

Так же, `Flux` сам по себе является не совсем библиотекой или инструментом 🤔.

`Flux` – `архитектурный шаблон приложения` разработанный инженерами `Facebook`.

Существует множество `реализаций`. И у некоторых из них смешные имена 😊:

- [Facebook's Flux](#)
- [Flummox](#)
- [Alt](#)
- [Fluxxor](#)
- [Flux This](#) 🤔
- [MartyJS](#)
- [NuclearJS](#)
- [Dolorean](#)
- [Reflux](#)
- [Fluxy](#)

И много других 😊.

Ситуация становится запутанной в связи с тем, что существует также `имплементация Flux`, разработанная `facebook` 🤔.

Да, такие как:

- Шаблон `Flux`, разработанный `Facebook` которому все могут следовать чтобы создать свою собственную реализацию;
- И библиотека `flux` – реализация шаблона `Flux`, так же была разработана `Facebook`.

Да, это запутанно, но так же важно и различать эти два понятия. Не стоит волноваться, просто запомните две вещи:

- `Flux` – это архитектурый шаблон приложения;
- Существует много воплощений шаблона `Flux`. И вы даже сможете создать ваше собственное!

! Важно:

Запомните: Flux – это архитектурый шаблон приложения, который может быть воплощён в жизнь любым разработчиком. Не более, не менее.

## Поток данных во Flux

Однажды `action` которое `created` с помощью `action creator`, приняло роль `dispatcher`.

`Dispatcher` является сущностью `singleton` в `Flux`. `Dispatcher` отвечает за получение объекта `action` и отправку его в поток данных `Flux` используя метод `dispatcher.dispatch(action)`.

Единственный `dispatcher` для проекта `Flux` 🏆.

Взгляните на реализацию `dispatcher` в `flux` от Facebook:

```
// app/dispatcher/index.js

import { Dispatcher } from 'flux';

export default new Dispatcher();
```

В коде ниже показано, как `action` может создаваться используя создатель действия `createPage`, в нашем `book-reader` приложении, которое мы исследовали вместе с Оскаром:

```
// app/actions/index.js

export const changePage = (page) => ({
  type: 'CHANGE_PAGE',
  payload: page
});
```

Армия обработчиков данных `Flux` появилась на поле битвы – `stores`. Да, `stores`, это множественное число.



Может быть много `stores` в приложении `Flux`, и каждый из `store` заботится об личной части `state`. Каждый `store` содержит часть `state` приложения внутри себя и как-то меняет его в соответствии с `отправленными действиями`.

Как только `действие` `отправлено` с помощью `dispatcher`, каждое `store` выполняет проверку, нужно ли обрабатывать этот `action` или же нет. Выполняя проверку, `хранилища` использует `action.type`, которое мы обсудили выше. По этому свойство `type` является `обязательным` – оно помогает `хранилищам` выделить один `action` среди других и решить есть ли необходимость в обновлении состояния.

`Dispatcher` предоставляет метод `register()` для `регистрации` `хранилища` которое будет уведомлено про все `отправленные действия` 🖨.

Так и только так `store` может узнать, что что-то произошло. Не существует других способов изменить управление данными `store`. Данные внутри `store` являются приватными и могут изменяться только самим `store` и то только после соответствующих `action` которые `отправляются` с помощью `dispatcher`.

Это раскрывает красивую и надежную композицию `one-way data flow`, когда данные возможно изменить только в `один` способ – используя `отправку действия`.

Здесь приведен пример того как `Flux store` может быть использован в нашем приложении `book-reader`, которое так любит `Оскар`:

```
// app/stores/index.js

import { EventEmitter } from 'events';
import dispatcher from '../dispatcher';

export default new class MagicBookStore extends EventEmitter {
  constructor () {
    super();

    this.state = {
      title:      'Magic and Enchantment',
      totalPages: 898,
      currentPage: '1'
    };

    dispatcher.register((action) => {
      switch (action.type) {
        case 'CHANGE_PAGE':
          this.changePage(action.payload);
          break;
      }
    });
  }
};
```

```

    }

    subscribe (callback) {
        this.on('change', callback);
    }

    unsubscribe (callback) {
        this.removeListener('change', callback);
    }

    update () {
        this.emit('change');
    }

    getInitialState () {
        return this.state;
    }

    getCurrentPage () {
        return this.state.currentPage;
    }

    changePage (newPage) {
        this.state.currentPage = newPage;
        this.update();
    }
}();

```

Когда `store` определяет, что `action`, который был `dispatched` в сфере своих обязанностей (использует проверку `action.type`, объявленную в `switch`, из примера выше), `store` выполняет логику, определенную разработчиком, для того чтобы вычислить новое состояние.

После этого `store` используя `EventEmitter` из `Node.js` модуля `events` запускает событие `change` и уведомляет UI о необходимости выполнить `re-render` в связи с изменением данных.

Давайте посмотрим на реализацию компонента `Book` который соединён с `MagicBookStore`:

```

// app/components/Book/index.js

// Core
import React, { Component } from 'react';

// Flux

```

```

import { changePage } from '../actions'
import dispatcher from '../dispatcher';
import MagicBookStore from '../stores';

export default class Book extends Component {
  constructor () {
    super();

    this.onChange = ::this._onChange;
    this.changePage = ::this._changePage;
  }

  state = MagicBookStore.getInitialState();

  componentDidMount () {
    MagicBookStore.subscribe(this.onChange);
  }

  componentWillUnmount () {
    MagicBookStore.unsubscribe(this.onChange);
  }

  _onChange () {
    this.setState(() => ({
      currentPage: MagicBookStore.getCurrentPage()
    }));
  }

  _changePage (event) {
    dispatcher.dispatch(changePage(event.target.value));
  };

  render () {
    const { currentPage, totalPages, title } = this.state;

    const pagesToSelect = [...Array(totalPages).keys()].map((page) => (
      <option key = { page }>{ page }</option>
    ));

    return (
      <section>
        <h1>{ title }</h1>
        Go to
        <select value = { currentPage } onChange = { this.changePage
      >
        { pagesToSelect }

```

```

        </select>
        page
        { /* current page content */ }
        <p>Total pages: { totalPages }</p>
    </section>

    );
}
}

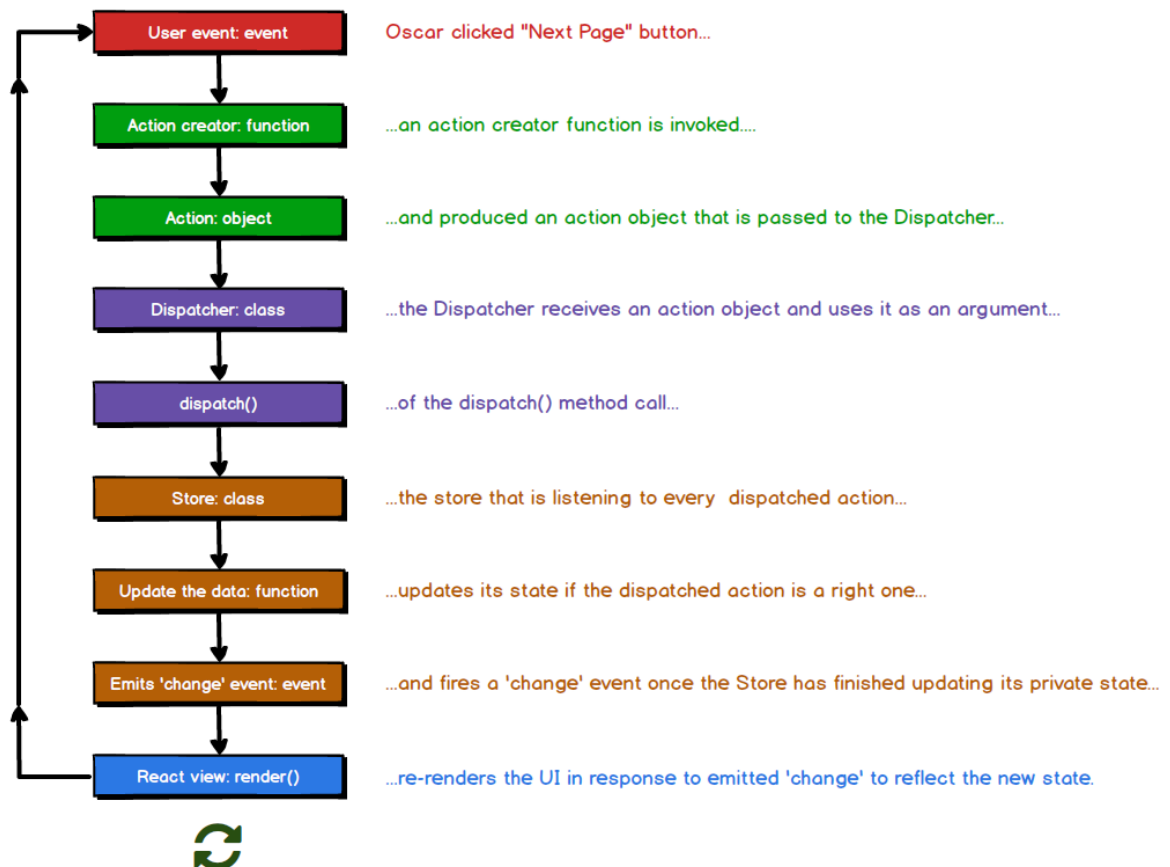
```

Сразу после метода `MagicBookStore.changePage()` вызывается событие `change` запущенное `EventEmitter`, выполняется метод `_onChange()` компонента `Book` и запрашивается новое обновление `state`, которое вызовет `re-render` компонента со свежими данными переданными в его состояние.

Это просто и читабельно!

Вот полная схема жизненного цикла `the one-way data flow` в `Flux`, которая происходит как только `Оскар` переключает следующую страницу в его электронной книге:

## Full Flux flow



Each new user action triggers a new one-way data flow lifecycle

---

Теперь у нас есть все детали про `Flux`, и мы готовы продвигаться, но перед тем как двигаться дальше, давайте сделаем быстрый обзор по экосистеме `Redux`, так как зёрна знаний посеяны в наших умах 🧠:

- В `Redux` единое `Store`, где всё `состояние` приложения находится внутри одного объекта;
- `Actions` `dispatched` с помощью `Store`;
- Прямые изменения состояния `запрещены`;
- Логика обновления состояния управляется специальной простой функцией `reducer`;
- Сильно вдохновлено идеями функционального программирования;
- `Redux` не зависит от `EventEmitter`;
- Так же предоставляет мощные фишки такие как `middleware` и `enchancers`;
- Может быть более `строгим` чем `Flux` но в результате дает больше стабильности.

И даже более того! Давайте вспомним фишки шаблона `Flux` которые мы только что изучили, для того чтобы подготовиться для более глубокого погружения в кротовую нору `Redux` 🚀!

## Итоги

`Flux` — это архитектура приложения, которую использует `Facebook` для построения клиентских приложений. Она дополняет составные представления `React` используя `unidirectional data flow`. Это больше чем шаблон, чем фреймворк, и вы можете начать использовать `Flux` немедленно, без большого количества нового кода 🧑🏻.


`Flux` может быть применен к любым инструментам, которые используются для создания фрагментов UI, хотя он лучше всего подходит к декларативной концепции `React` ⚙️.



Мы так же пробежимся по ключевым игрокам `Flux`:

- `actions`;
- `dispatcher`;
- `stores`;
- `React views`.

Здесь много новых вещей, и это может вас запутать. Возможно, будет полезным подумать про `Flux` и `React` как будто бы они приятели, которые взаимодействуют друг с другом. К примеру можно представить такой диалог:

- `React`: Эй, действие `changePage`, кто-то (`Оскар`) выбрал новую страницу в продолжении 🧑🏻;
- `Action`: Оу, спасибо, `React`! Я зарегистрирую `action creator` в `dispatcher`, так чтобы `dispatcher` мог позаботиться об уведомлении всех заинтересованных `stores` 🧑🏻;
- `Dispatcher`: Оу, дай ка, я посмотрю кого ты выбрал для того чтобы позаботится о `new`

`page`. Ага! Похоже, что `MagicBookStore` зарегистрировал обратное действие для `CHANGE_PAGE` `action` вместе со мной, так что нужно его уведомить ;

- `store`: Привет, `dispatcher`! Спасибо за обновление! Я обновлю свои данные теми, которые ты отправил. Потом я запущу событие для компонентов `React` которые нуждаются в нём ;
- `React`: Ооо, новые блестящие данные от `store`! Обновлю ка я UI чтобы отобразить это .

Это диалог четырёх ключевых игроков `Flux unidirectional data flow`.

Итак, `Flux`:

- Управляет состоянием приложения;
- Предназначенный для `React`, но работает с `любым view движком`;
- Менее строгий но менее надежный по сравнению с `Redux`;
- В его основе `единый dispatcher`, и `множество stores`;
- Обновление состояния приложения обрабатываются каждым `Store`;
- Предвестник к `Redux`.

Спасибо за чтение, и увидимся на следующем уроке .

Если у вас есть идеи как можно улучшить урок, пожалуйста, поделитесь с нами `hello@electrum.io`. Ваш отзыв очень важен для нас!