

## 2. Знакомство с Redux

---



- Обзор
- Три принципа Redux
- Actions
- Store
- Reducers
- Middleware
- Enhancers
- Redux против Flux
- Итоги

### Обзор

Привет, и добро пожаловать, дорогой друг 🙌😊! В этом уроке, нашей целью является ознакомление с `Redux`. `Redux` был создан `Деном Абрамовым`, и представляет собой эволюцию `Flux`.

Беря во внимание тот факт, что мы уже фундаментально знакомы с `Flux` из предыдущего урока, в этом уроке мы используем полученные знания и будем много ссылаться и сравнивать `Flux` чтобы лучше понять идею `Flux` и `Redux`.

В этом уроке:


-  это символ означает `схожесть` между `Flux` и `Redux`,
-  а этот символ означает `различие`.

Мне очень радостно от того, что я могу поделиться с вами фактами о `Redux`, так что давайте начнем 🏃!




### Три принципа Redux

`Redux` — это предсказуемый контейнер состояния 📦 для приложений JavaScript. Первый релиз `Redux 0.2.0` был выпущен для всеобщего обозрения `2-го Июня, 2015`.


`Redux` построен на основе `трёх фундаментальный принципов` 🦄:

-  В отличие от `Flux`, состояние всего вашего приложения – является `единым неизменяемым объектом`. И это не только концептуально проще чем `multi-store` модель `Flux` – владение состоянием как единым неизменяемым объектом помогает отладке 🐛, поддерживает `SSR` 🖥️ и создает `time-travel` ⌚ фишки, такие как `undo/redo`



нереально легкие для воплощения в жизнь.

-  Так же как и в `Flux`, единственный способ изменения состояния является `emit an action`, который описывает `намерение пользователя` .
-  так же отличием от `Flux` – является управление изменением состояния с помощью новой отдельной сущности `reducers`, который является простой, чистой функцией.

## Actions

Так же как в `Flux` , в `Redux` события происходящие в приложении называются `actions`. `Actions` – это обычный JavaScript объект который описывает события.

 Важно:


Так же как и в `Flux` , каждое `действие` должно иметь обязательное уникальное свойство `type` то, что поможет отделить одно `действие` от другого. Остальные же свойства `действий` абсолютно на ваше усмотрение. Также стандарты `FSA` могут быть применены и в приложении `Redux`, что, между прочим, может быть хорошей практикой .

Свойство `action.type` должно иметь `serializable` значение. Тип `string` может послужить правильным выбором.

 Заметка:

В JavaScript сериализуемые сущности это сущности, которые могут быть конвертированы в форму JSON, и переконвертированы в их начальную форму обратно сохраняя функциональность, свойства, данные и наследственные связи.

Сущности такие как `Symbols`, `Functions` и `Promises`, так же как `Sets` и `Maps` не могут быть безопасно сериализованы без дополнительных инструментов. Хотя технически все еще возможно, использовать их типы и значения в свойстве `action.type`, хотя очень не рекомендовано. Тип `String` лучше всего подходит для этого.

Так же, как и в `Flux` , `actions` можно создавать с помощью функций, которые называются `action creators`. Типичный `action creator` имеет такое же имя как и `action.type`.

Так же, полезно извлекать свойство `action.type` в отдельную `константу` для повторного её использования в других частях приложения. Этот трюк помогает избежать общих ошибок, которые может допустить разработчик пока набирает текст.

Беря приложение `Оскара` `book-reader` из предыдущего урока, вот как может выглядеть `changePage` `action creator` вместе с извлеченным в `константу` свойством `action.type`:

```
// app/actions/types.js

export const CHANGE_PAGE = 'CHANGE_PAGE';
```

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

import { CHANGE_PAGE } from './types';

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

Используя константу `CHANGE_PAGE` таким способом, мы объявляем **единый источник правды** для константы **типа действия** всего приложения. Это становится очень удобным особенно, когда ваше приложение увеличивается, и вы начинаете привязывать логику к `reducers` (`reducers` тема этого урока) где константы так же показывают свою полезность 🍷.

## Store

`Store` в `Redux` – центральный актер на сцене .

Вы можете создать `store` вызвав функцию `createStore`, которая была предоставлена главным API пакета `redux`.

🔍 Заметка:

Иногда, функция `createStore()` так же относится к `store creator`.

Вы передаете ваш `root reducer` в качестве первого аргумента функции `createStore`.

```
// app/store/index.js

import { createStore } from 'redux';
import reducer from '../reducers';

export default createStore(reducer);
```

А это второй возможный аргумент который вы можете передать в `createStore` в том случае если вам нужно **сохранить** состояние вашего приложения для каких-либо целей.

```
// app/store/index.js

import { createStore } from 'redux';
import reducer from '../reducers';

const preloadedState = localStorage.getItem('__@preloadedState');

export default createStore(reducer, preloadedState);
```

Поступая таким образом, вы можете восстановить ранее сереализованное и сохранённое в `localStorage` (или другое хранилище) `state` приложения.

К примеру, представьте себе что `Оскар` читает свою книгу `Magic and Enchantment` на странице `345` и внезапно, его вызывает `Committee of Sorcerer Developers` для решения каких-то срочных вопросов 🧙! `Оскар` бросает все, трансформируется в "Великого Грифона" и улетает прочь чтобы помочь своим друзьям. Все прошло благополучно, `Оскар` спас Королевство и получил в дар поцелуй от Прекрасной Принцессы 🧚. В данном случае `Оскар` сохранил предыдущее состояние приложения `book-reader` в `localStorage` и теперь он может просто `восстановить` предварительно сохраненную сессию извлекая её из `localStorage` и передавая её функции `createStore()` в качестве второго аргумент. Юху 🙌!

Вы так же можете использовать трюк `preloaded state`, если вы создаете видео игру и хотите сохранять прогресс её пользователей все время, что бы иметь возможность продолжить с момента где вы закончили играть в последний раз 🎮.

🔍 Заметка:

Вы так же можете `обновлять` состояние от `сервера` в `universal/isomorphic` приложениях.

Это тот момент, который контрастирует с `Flux` ❌. Потому что в `Flux`, `stores` содержит в себе и данные состояния, и логику манипуляций состоянием.

В то время как `Redux` `store` награжден `Single Responsibility Principle` потому что единственный `store` содержит полный объект состояния, и `reducers` управляет изменением этого состояния.

`Store` – это объект, и его API очень прост. Он может:

- `запускать` действия:

```
store.dispatch( action: Object ): void => null
```

- `подписываться` и `отписываться` от `слушателя`:

```
Store.subscribe( listener: function ): function => unsubscribe
```

- перейти к текущему состоянию:

```
Store.getState(): Object => state
```

- заменить текущий reducer новым:

```
Store.replaceReducer( nextReducer: function ): void => null
```

Функция `dispatch` используется для запуска объекта `action`.

Функция `subscribe` является более низкоуровневой API и более вероятно, что в реальной разработке, вместо того чтобы использовать её напрямую, привязки `React` возможно будут лучшим выбором, но больше об этом вы узнаете в следующем уроке. Так как в данный момент, нашей целью является изучение `store.subscribe()` идиоматический функциональный принцип.

`subscribe` получает функцию обратного вызова в качестве аргумента, который может быть представлен как `listener`. Функция обратного вызова `listener` будет вызываться каждый раз когда меняется состояние. Возвращённое значение из `subscribe(listener)` является функция `unsubscribe`. Которая может быть использована для отписки от предыдущей подписки.

Функция `getState` всего лишь возвращает весь объект состояния приложения.

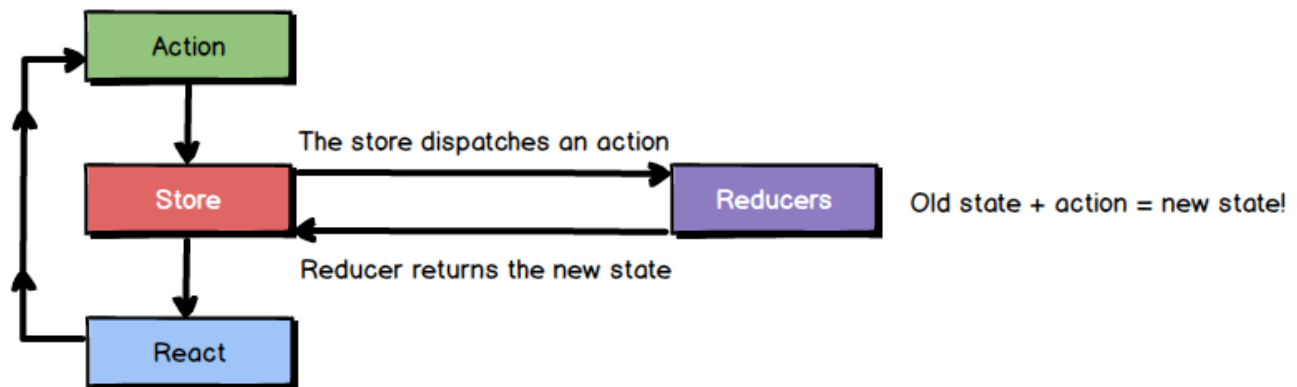
Функция `replaceReducer` продвинутый API, который может быть использован для поддержки фишки `hot reloading`. Так же, эта функция может подойти во время разработки более продвинутых фишек `code-splitting`.

🔍 Заметка:

`Hot reloading` — это когда во время разработки вы можете немедленно увидеть изменения в вашем браузере без потери текущего состояния клиентом.

# Redux flow

---

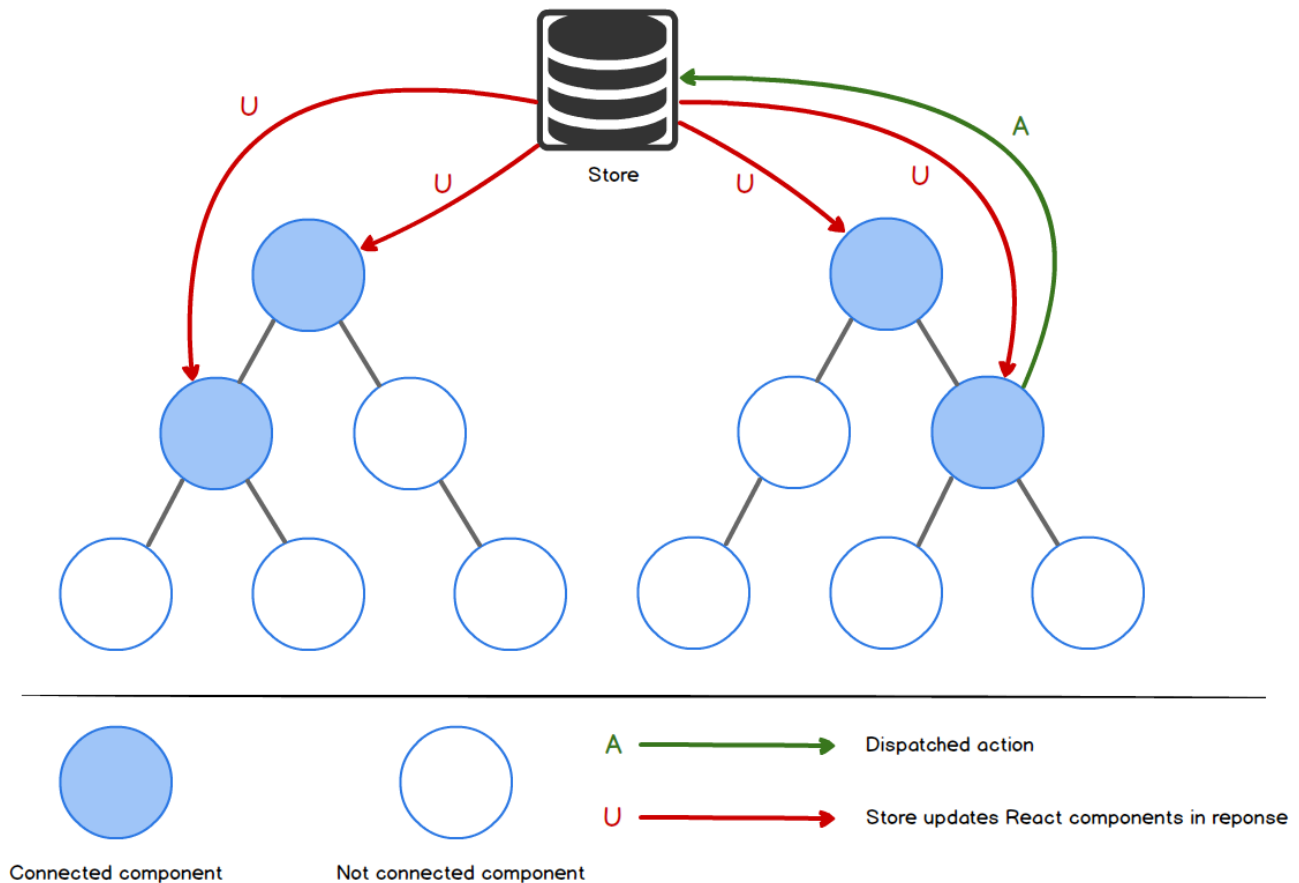


Наиболее захватывающий момент это то, что `Store` не предоставляет никаких API для прямого изменения данных состояния. Это означает, что единственный способ изменения состояния это `запуск действия`. Каждое `действие` управляется при помощи подходящей функции `reducer`, которая в свою очередь запустит вычисления и вернет новое состояние приложения. Эти фишки доказывают что `неизменяемость` природна для `Redux`, `store` – не может изменяться напрямую 🚫.

`Redux` навязывает содержание всех `state` в едином централизованном объекте. Это делает приложение более доступным для понимания и аннулирует необходимость управления взаимодействия между многочисленными `stores`, как во `Flux`. Более того, `Redux` благоприятный для построения приложения `isomorphic/server-side-rendered` и это чудесно 🙌.

---

## Redux store



Представьте себе, что каждый круг это `React component`. Что если у нас есть компоненты `React` в разных частях приложения, которым нужно управлять и работать с одними и теми же частями состояния приложения 🤔?

Если синие компоненты работают с одними и теми же частями данных, то как тогда они будут взаимодействовать гарантируя то, что данные всегда будут оставаться синхронизированными? Ответ прост – централизованное хранилище Redux 🎯.

Вы можете думать про `store` как о локальной клиентской базе данных 🗄️. Как только у вас появится хранилище, компонент `React` может запустить действие, которое обновит единственный `store`. Так что как только ваш компонент присоединится к `store`, и когда `store` обновится, он немедленно уведомит о том, что состояние приложения изменилось и выполнит `re-render` чтобы отобразить новое состояние в UI.

## Reducers

Чтобы изменить состояние, вы запускаете действие, которое в конечном счете регулируется `reducer`.

`Reducers` – одна из новых концепций представленных в `Redux` <sup>NEW</sup>. `Reducers` – чистые функции, которые берут текущее `состояние` и `запущенное действие` в качестве аргументов, потом вычисляют и возвращают новое `состояние`. Приложение `Redux` может использовать множество функций `reducer`, и каждая из них отвечает за свою долю состояний. С того момента как `reducers` стали `чистыми функциями`, `побочные эффекты` не могут быть произведены с помощью `reducers`.

🔍 Заметка:

В программировании, `side-effect` это когда функция изменяет состояние чего-либо, что находится за её границами. К примеру, функция которая отправляет `fetch` запрос создает `side-effect`. Чистые функции не должны иметь `побочных эффектов`. `Reducers` это `чистые функции`, и потому – не должны отправлять `fetch` запросов. Мы будем обсуждать `pure function` более углубленно на следующем уроке.

Вид `reducer`:

```
(state, action) => state;
```

Вот как функция `reudcer` будет выглядеть в приложении `Оскара` `book-reader`:

```
// app/reducers/magicBook/index.js

import { CHANGE_PAGE } from '../actions/types';

const initialState = {
  title:      'Magic and Enchantment',
  totalPages: 898,
  currentPage: '1'
};

export default (state = initialState, action) => {
  switch (action.type) {
    case CHANGE_PAGE:
      return Object.assign({}, state, { currentPage: action.payload });

    default:
      return state;
  }
};
```



В предыдущем примере мы использовали константу `CHANGE_PAGE` в `action creator`. Заметьте как константа `CHANGE_PAGE` вторично использовалась `reducer` 'ом. Вот где проявляется настоящая мощь подхода извлечения констант <sup>100</sup>, и его полезность возрастает вместе с ростом приложения.

🔍 Заметка:

Конструкция `switch` является опциональной. Вы можете составлять определение `action.type` удобным для вас способом. Конструкция `switch` здесь всего лишь шаблон.

Каждый раз когда `changePage` действие запускается, функция `reducer` выходит на сцену. Он принимает текущее состояние и запущенное действие в качестве аргументов, и выполняет проверку действия которое было запущено, это то единственное о чем должен заботиться `reducer`. Если это так, то новое состояние вычисляется и возвращается в `store`. Если не было совпадений в `action.type`, то состояние возвращается без изменений.

! Важно:

Функция `reader` всегда должна вернуть состояние, даже если вообще не было совпадений в `action.type`.

Существует возможность совместить множество `reducers` в единственный `root reducer`, который может быть использован как аргумент для вызова `createStore`.

`Redux` предоставляет функцию помощника `combineReducers` из своего главного API.

Вид будет следующий:

```
combineReducers(reducers: Object): function => rootReducer
```

К примеру, если у нас есть информация `user` в состоянии `book-reader` приложения Оскара 📖, то вот как мы можем использовать `combineReducers` что бы вычислить единственный `reducer` из множества:

```
// app/reducers/user/index.js

import { UPDATE_USER } from '../../actions/types';

const initialState = {
  firstName: 'Oscar',
  lastName: 'Egilsson'
};

export default (state = initialState, { type, payload }) => {
  switch (type) {
```

```

    case UPDATE_USER:
      const { firstName, lastName } = payload;

      return Object.assign({}, state, { firstName, lastName });

    default:
      return state;
  }
};

```

#### 🔍 Заметка:

Попытайтесь сосредоточиться на синтаксисе `destructuring assignment`, который используется в этом `reducer` чтобы сделать код более сжатым и менее повторяющимся.

```

// app/reducers/index.js

import { combineReducers } from 'redux';
import user from './user';
import magicBook from './magicBook';

export default combineReducers({
  user,
  magicBook,
});

```

И используйте наш `root reducer` как аргумент для `createStore`:

```

// app/store/index.js

import { createStore } from 'redux';
import reducer from '../reducers';

export default createStore(reducer);

```

В нашем случае, следующий объект `state` возникнет в соответствии с вызовом `createStore(reducer)`:

```
{
  user: {
    firstName: 'Oscar',
    lastName: 'Egilsson'
  },
  magicBook: {
    title: 'Magic and Enchantment',
    totalPages: 898,
    currentPage: 1
  }
}
```

В этом случае `store` и `root reducer` соединены .

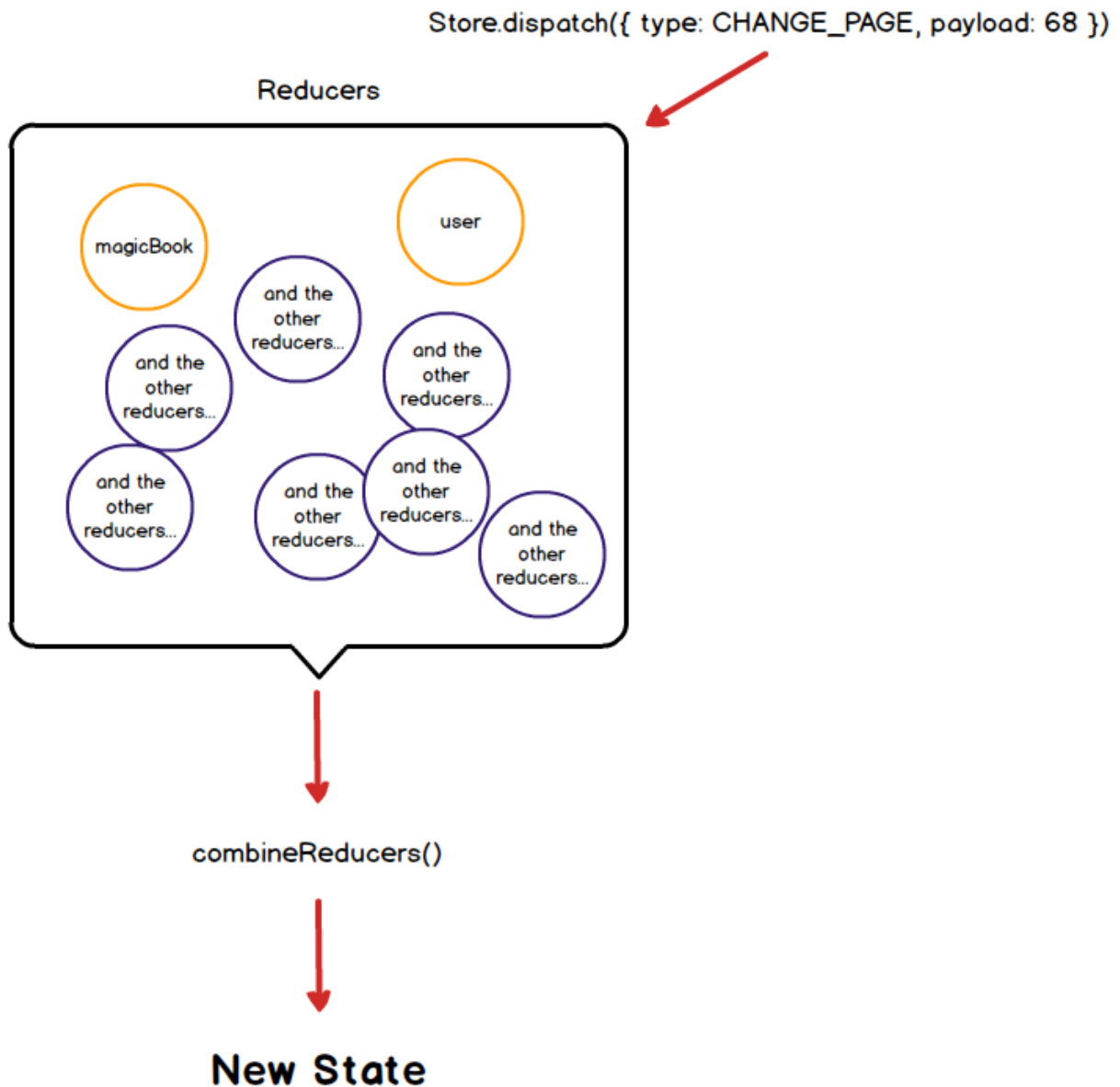
Когда создается `store`, `Redux` вызывает `reducers` и использует их возвращенные значения как `initial state` приложения. Однако, вам может быть интересно, есть ли у нас множество `reducers`, который из них вызывается, когда действие запускается 🤔?

Ответ – `все`.

---

## All Reducers are called on each dispatch

---



---

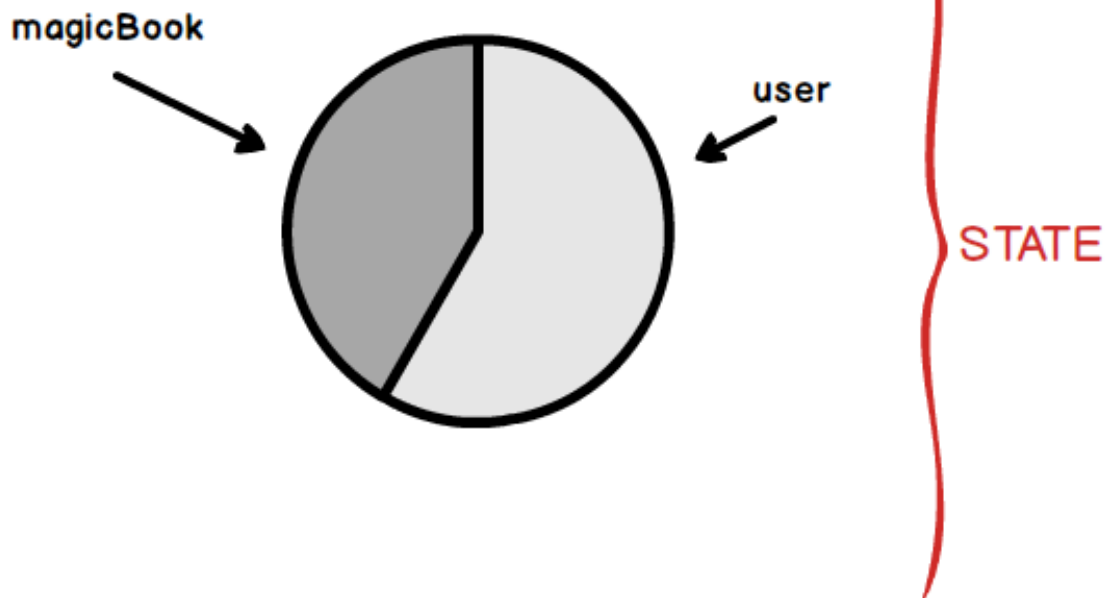
Каждый единичный `reducers` вызывается, когда действие запускается. Конструкция `switch` внутри каждого `reducer` смотрит на `action.type` чтобы определить нужно ли ему что-то сделать. Вот почему важно чтобы все `reducers` возвращали не тронутый `state` как `default`, если нет совпадений, `switch case` пропускает `action.type`.

Вот как получается возможным для каждого `reducer` управлять своей собственной частью состояния.

---

## Each Reducer = a 'slice' of State

---



👉 Совет:

Вам не обязательно использовать `combineReducers` в вашем приложении – это всего лишь полезная функция, не более того! И она не управляет всеми возможными сценариями. Абсолютно возможно написать логику `reducer` без её использования, и это вполне нормально, когда необходимо писать пользовательскую логику для `reducer` в случаях когда `combineReducers` не справляется.

## Forbidden in Reducers:

---

- ☹ **Mutate arguments**
- ☹ **Perform side effects**
- ☹ **Call non-pure functions**

Вернемся к приложению `Оскара` `book-reader`, вот как компонент `Book` будет выглядеть в `Redux`:

```
// app/components/Book/index.js

// Core
import React, { Component } from 'react';
import { changePage } from '../../actions';
import Store from '../../store';

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

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

  state = Store.getState().magicBook;

  componentDidMount () {
    this.unsubscribe = Store.subscribe(this.onChange);
  }

  componentWillUnmount () {
    this.unsubscribe();
  }

  _onChange () {
    this.setState(() => Store.getState());
  }

  _changePage (event) {
    Store.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>
      </section>
    );
  }
}
```

```

    Go to
    <select value = { currentPage } onChange = { this.changePage
  }>

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

  );
}
}

```

Заметьте **линия кода 16**: **состояние компонента** **Book** инициализируется **Store.getState().magicBook**. Это потому что мы расширяем **Redux state** со вторым **user reducer** и совмещаем **user** и **magicBook reducers** с помощью функции **combineReducers**. В результате получаем **Redux state** объект с двумя свойствами:

- **magicBook**: часть **state**, который содержит **magic book data**;
- **user**: часть **state**, который содержит **user data**.

Вот почему свойство доступа **.magicBook** объявлено в конце конструкции **Store.getState().magicBook** на **линии кода 16**.

 Заметка:

Этот пример воплощает в жизнь не обработанное использование **Redux** непосредственно в компоненте **React**. Да, в один момент данный подход может показаться немного неловким, но мы изучим более продвинутые и элегантные методы соединения **React** с **Redux** в следующих уроках.

Помните 🙌: **reducers** должны быть **pure functions**. Эти детали означают, что они не должны производить какие-либо **side-effects**. Если у вас есть **pure function**, вызов её с теми же аргументами всегда возвращает один и тот же результат 100.

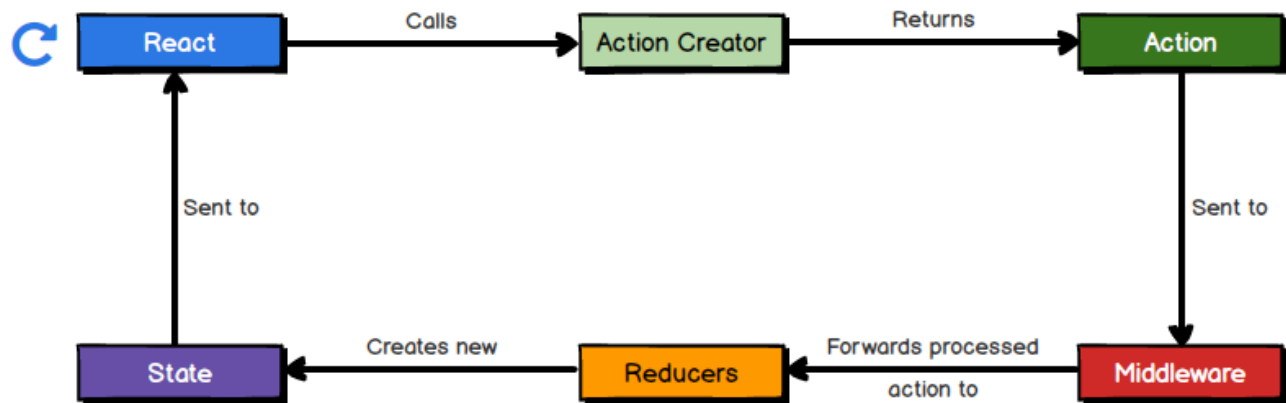
## Middleware

**Middleware** – это мощное понятие **Redux**. Вы можете подумать про **middleware** как про систему **plug-in**. 💪

**Middleware** может отвечать на **действия** уникальным для неё способом. **Middleware** вводятся в **store** во время его создания, перехватывает каждое **запущенное действие** и обрабатывает его до или после того как отправит его в **reducer**.

В основном, `middleware` может научить ваш `store` делать абсолютно новые вещи. К примеру, понимать `action`, возвращать значения `promise` вместо `object`, а потом запустить подходящие действия, в то время как `promise` `resolves` или `rejects`. Так же возможно ввести функциональность сбора аналитики сразу в `store` и даже больше!

## Redux life-cycle with middleware



Once the **middleware** intercepts an **action**, you can:

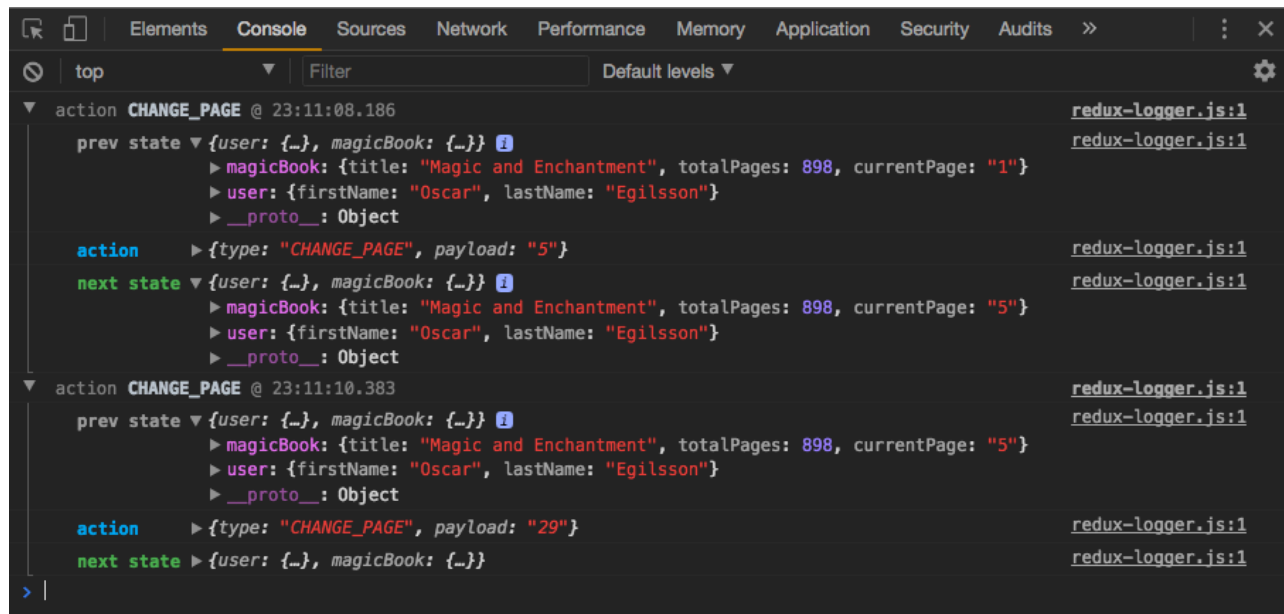
- log, modify, transform or cancel it;
- send fetch requests inside the middleware (analytic statistic for instance);
- pass it to the next middleware in the chain (if there are any);
- finally, delegate it to the reducer if there are no more middleware.

Возможно, самое обыкновенное использование `middleware` в разработке это [redux-logger](#).

`redux-logger` `middleware` вполне легкий, но адски полезный. Каждый раз когда действие является `запущено`, `redux-logger` `middleware` вызывает `console.log` с:

- состояние приложения перед тем как действие будет запущено ;
- запущенное действие само собой;
- состояние приложения после того как действие обработано в `reducers`, новое состояние вычисляется и возвращается.





Ухты, да это же просто здорово! 🤔

Круто то, что `middleware` это и есть API, это так просто. Внедряя новый `middleware` в ваше приложение, вы можете использовать вспомогательную функцию `applyMiddleware()`, которая поставляется вместе с `Redux`.

```
// app/store/index.js

import { createStore, applyMiddleware } from 'redux';
import { createLogger } from 'redux-logger';
import reducer from '../reducers';

const logger = createLogger();

export default createStore(reducer, applyMiddleware(logger));
```

Следующий синтаксис может быть применен в том случае, если вам понадобятся множественные `middleware` в вашем приложении:

```
const middleware = [firstMiddleware, secondMiddleware];

export default createStore(reducer, applyMiddleware(...middleware));
```

Так что это оно! Просто передайте `middleware`, который вы захотите в `applyMiddleware()` как аргумент.

Однако, как это может быть возможным, если второй аргумент функции `createStore()` `preloaded state`, спросите вы 🤔? Ответ прост: `Redux` достаточно умен, чтобы понять, что если вы не хотите использовать возможную фишку `preloaded state` – вы просто её пропускаете, и `Redux` делает всю грязную работу за вас.

```
createStore(reducer, applyMiddleware(...middleware));

// Works too:

createStore(reducer, preloadedState, applyMiddleware(...middleware));
```

– Похоже на магию 🪄! – Так оно и есть – говорит `Оскар` мечтательно – ...но, знаете, я вам покажу еще больше магической магии. Давайте создадим наш собственный `logger middleware` 🧙!

```
// app/store/index.js

import { createStore, applyMiddleware } from 'redux';
import reducer from '../reducers';

const logger = (store) => (next) => (action) => {
  console.log('Previous state:', store.getState());
  next(action);
  console.log('Action:', action);
  console.log('Next state:', store.getState());
};

export default createStore(reducer, applyMiddleware(logger));
```

Тут мы видим вывод нашего вновь вызванного `conjured-up` пользовательского `middleware`:

---

```
Elements Console Sources Network Performance Memory Application Security Audits >>
top Filter Default levels
Previous state: {user: {_, magicBook: {_}} index.js:8
  ▶ magicBook: {title: "Magic and Enchantment", totalPages: 898, currentPage: "1"}
  ▶ user: {firstName: "Oscar", lastName: "Egilsson"}
  ▶ __proto__: Object
Action: {type: "CHANGE_PAGE", payload: "21"} index.js:10
Next state: {user: {_, magicBook: {_}} index.js:11
  ▶ magicBook: {title: "Magic and Enchantment", totalPages: 898, currentPage: "21"}
  ▶ user: {firstName: "Oscar", lastName: "Egilsson"}
  ▶ __proto__: Object
Previous state: {user: {_, magicBook: {_}} index.js:8
  ▶ magicBook: {title: "Magic and Enchantment", totalPages: 898, currentPage: "21"}
  ▶ user: {firstName: "Oscar", lastName: "Egilsson"}
  ▶ __proto__: Object
Action: {type: "CHANGE_PAGE", payload: "11"} index.js:10
Next state: {user: {_, magicBook: {_}} index.js:11
  ▶ magicBook: {title: "Magic and Enchantment", totalPages: 898, currentPage: "11"}
  ▶ user: {firstName: "Oscar", lastName: "Egilsson"}
  ▶ __proto__: Object
>
```

Функция `logger` имплементирует пользовательский `logger middleware`. У нас есть функция с параметром, которая возвращается в функцию с новым параметром, что само по себе уже является посредником:

- первая функция ссылается на `store` в аргументе, таким образом делая возможным использование `store.getState()` в теле `middleware`;
- вторая функция, предоставляет параметр `next`, который является функцией, которая будучи вызванной с аргументом `action`, передает `action` в `next` посреднику по цепочке или к `reducers` если нет других посредников;
- третья функция содержит ссылку на текущий обрабатываемый объект `action`;
- и наконец-то четвертая — это сам посредник. Форма его тела и логика полностью зависит от вас! Вас ограничивает только ваше воображение.

😬...и да, с начала это может взорвать вам мозг и выглядит пугающим. Однако, не волнуйтесь, это все требует времени, чтобы приспособиться к ряду вложенных функций.

Немного практики и вы сможете создать ваш собственный `even-more-magical` посредственник, как это ежедневно делает Оскар 🧙.

## Enhancers

`Enhancers` всего лишь функция `higher-order`, которая используется для усиления дефолтного поведения `store`. В основном, `enhancer` расширяет функциональные возможности хранилища. Он берет `createStore()` функцию и возвращает, усиленную (или функционально расширенную) `createStore()`.

## 🔍 Заметка:

Функция `higher-order` это функция, которая берет одну или несколько функций как аргументы, и/или возвращает функцию как её результат. Мы обсудим эти `higher-order` функции более углубленно на следующих уроках.

В контрасте к `middleware`, `enhancers` обеспечивают дополнительные фишки для вашего приложения `from the above`, в то время как `middleware` делает `from the inside`. Типичный `enhancer` – является `superstructure`, в то время как типичный `middleware` – является `plug-in`.

К примеру, `enhancers` может расширять функциональность `store` таким путем:

- Добавление автоматического постоянства `state` между сессиями пользователей;
- Синхронизируя `state` между вкладками в новом браузере;
- Интеграция мощного `Redux developer tools`.

Чтобы ввести `Redux devtools` в ваше приложение сделайте три шага:

1. Установите [Redux DevTools](#) из `Chrome Web Store`;
2. Обновите `store creator` следующим способом:

```
// app/store/index.js

import { createStore, applyMiddleware, compose } from 'redux';
import reducer from '../reducers';

const logger = (store) => (next) => (action) => {
  console.log('Previous state:', store.getState());
  next(action);
  console.log('Action:', action);
  console.log('Next state:', store.getState());
};

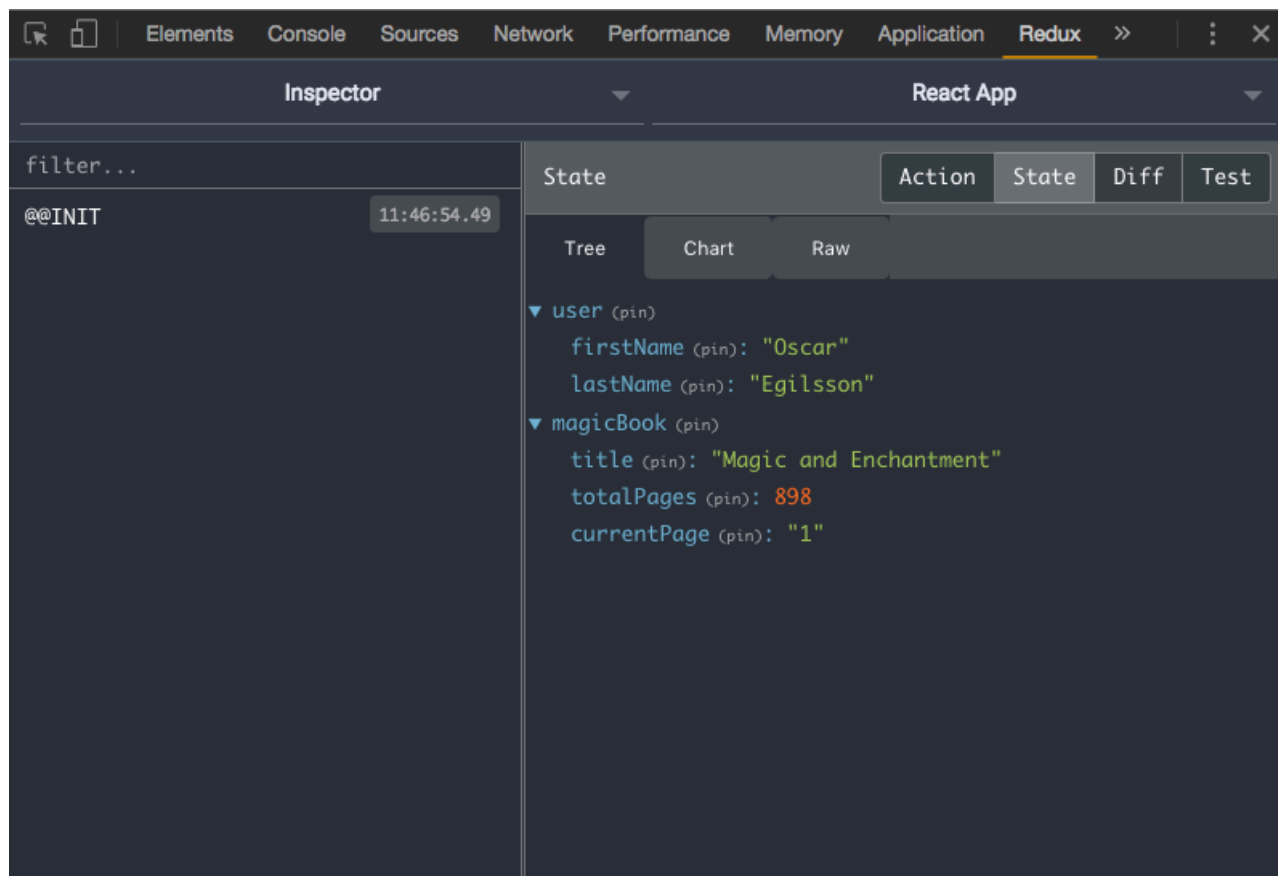
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ||
compose;

export default createStore(
  reducer,
  composeEnhancers(applyMiddleware(logger))
);
```

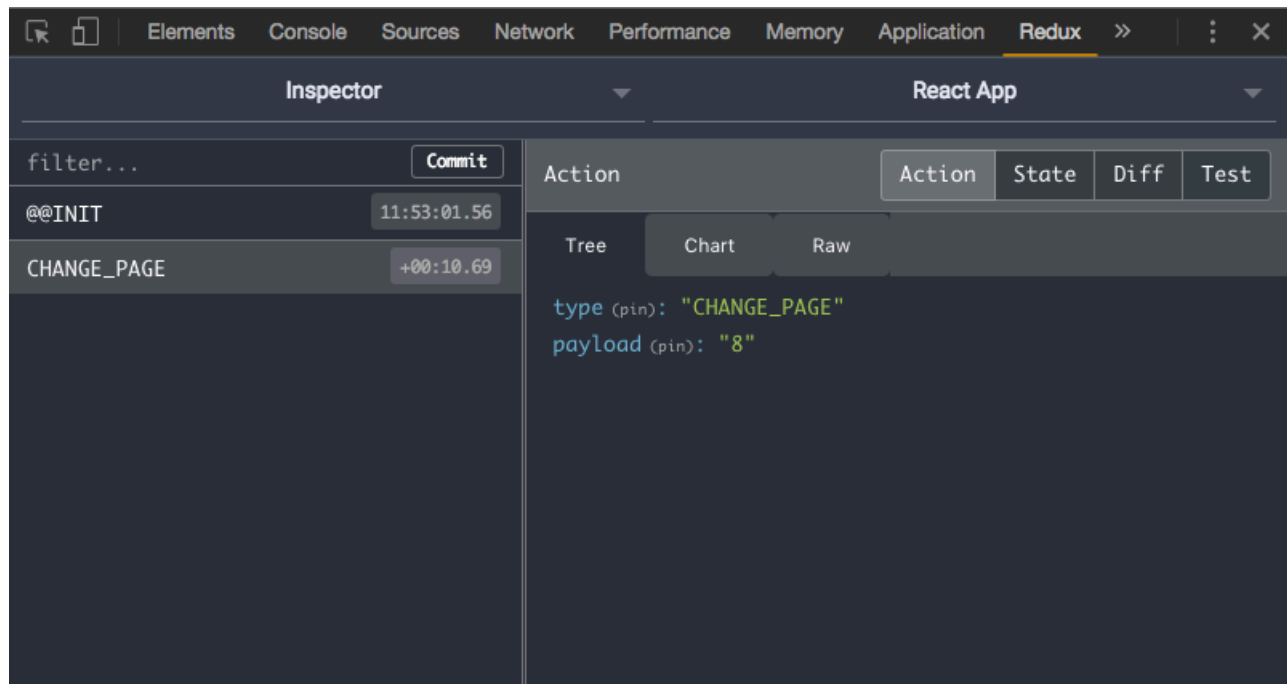
## 🔍 Заметка:

Если функция `compose` вас пугает – не волнуйтесь, это всего лишь функциональная программная утилита. Это одна из тем, которую мы будем обсуждать на следующих уроках.

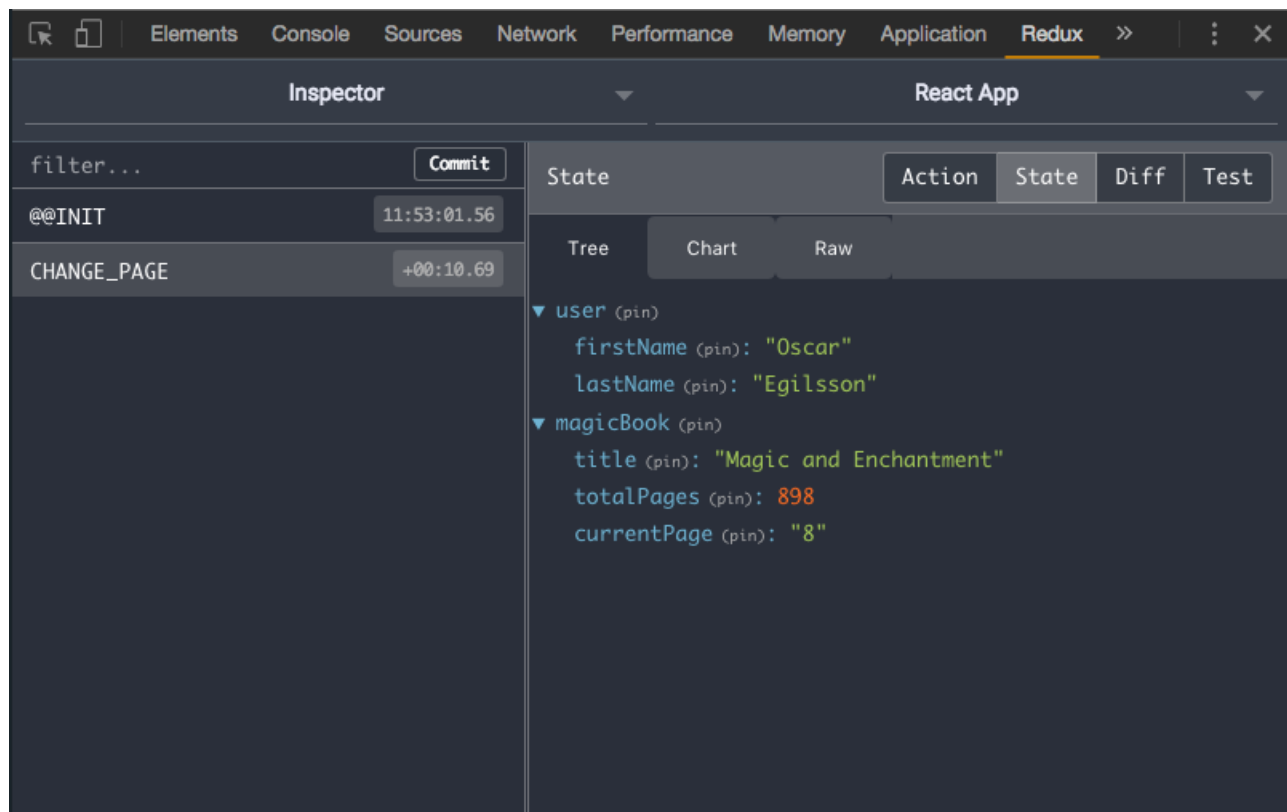
И Вуаля! `Redux DevTools` готов к использованию! Как только все правильно завершено, возникнет новая вкладка `chrome dev tools`, представляющая текущее состояние удобным способом:



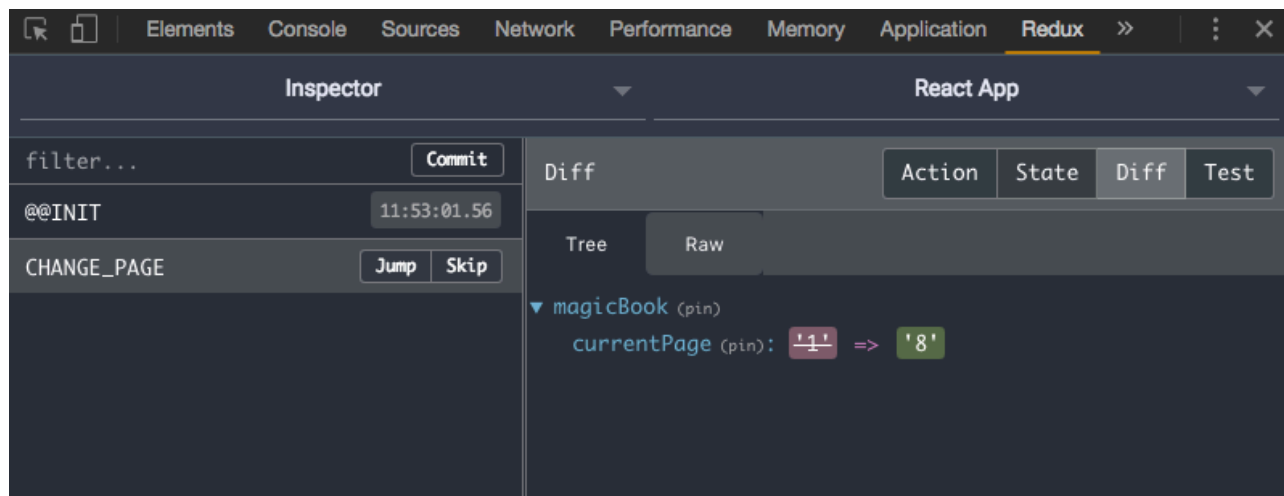
Так же, всякий раз когда действие запускается, оно немедленно мониторится `Redux DevTools`. Вы можете увидеть абсолютно все объекты действия для каждого запущенного действия:



Как и новое состояние обновленное в ответ на каждое действие:



И даже diff из previous state и new state:



Когда Ден Абрамов впервые продемонстрировал Redux developer tools с time travel debugging и hot reloading фишками на React Europe Conference, люди просто ахнули. Time travel debugging – это очень мощный способ, чтобы увидеть как состояние вашего приложения постоянно меняется 🕒.

Вы можете:

- сделать шаг назад и вернуть время, когда вы отлаживали и видели каждое изменение состояния и как это происходило;
- Откатить изменения состояния в прошлое и видеть как это оказало эффект на конечное состояние, в текущем времени;
- Вы даже можете выключить индивидуальное actions, которые возникали чтобы увидеть как состояние вашего приложения будет выглядеть как будто определенные action никогда не случались 🤖.

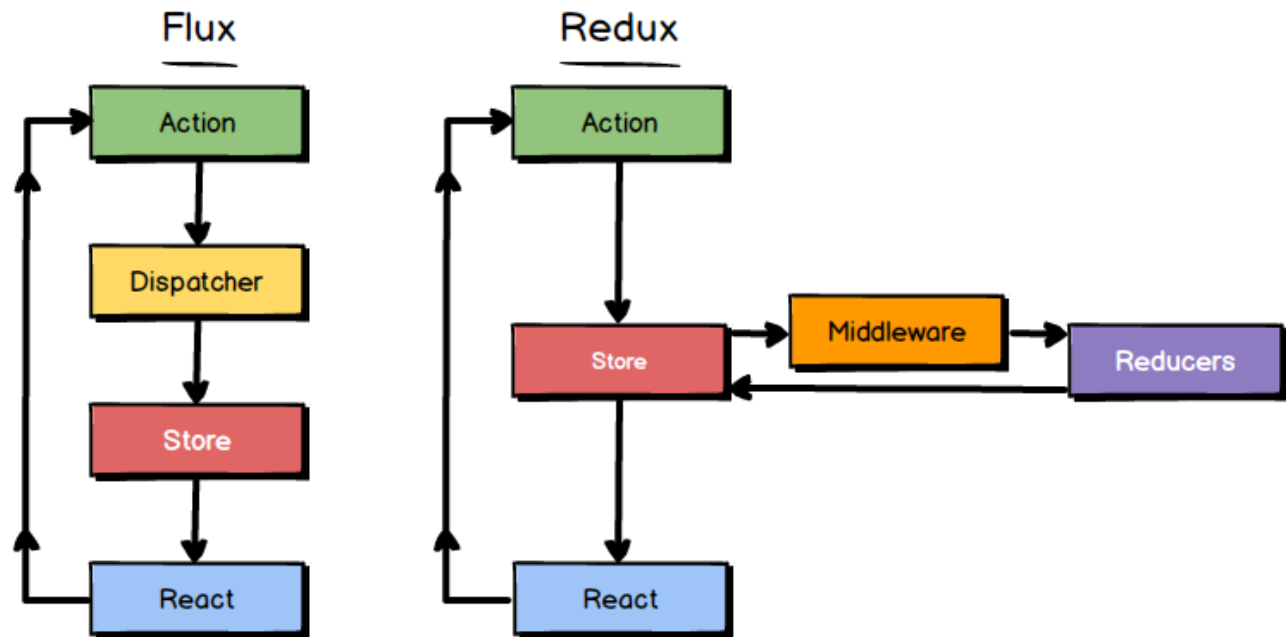
Довольно таки невероятно 🦄!

## Redux против Flux

Так как мы уже знаем, Flux имеет три ключевые концепции: actions, dispatcher и stores. Когда action запускается, dispatcher уведомляет stores. Так что Flux использует единственный dispatcher чтобы соединить actions со множественными stores. Когда состояние store обновляется, store использует EventEmitter чтобы обновить React view.

В сравнении, Redux вообще не имеет dispatcher. Функция dispatch предоставляется единственным store, так что Redux вообще не нужен dispatcher. Для окончательных изменений, Redux зависит от чистой функции, которая называется reducers.

## Flux vs Redux



Хотя `Redux` следует таким же принципам, как и `Flux`, мы можем увидеть тонкую разницу между ними в сравнении:



Redux	Общее	Flux
One-way data flow философия	✓	One-way data flow философия
Не привязан к виду	✓	Не привязан к виду
Изменение состояния вызывается dispatching an action	✓	Изменение состояния вызывается dispatching an action
Необязательно action creators	✓	Необязательно action creators
Действие отправляется единичным store	✗	Действие отправляется единичным dispatcher
state живет внутри store	✓	state живет внутри store
state является immutable	✗	state является mutable
Единичный store с иерархическими reducers	✗	Несколько единообразных отключённых stores
Изменения state управляются reducers	✗	Изменения state управляются stores
Не зависит от EventEmitter	✗	Зависит от EventEmitter
Мощная middleware и enhancers система	✗	Не увеличивает функциональность
Легковесный: 6 кВ	✓	Легковесный: 3 кВ

Так что же лучше использовать Flux или Redux?

Хорошо, принимая во внимание невероятно богатый plug-in и другие вспомогательные инструменты, которые были разработаны за последние годы, Redux это хороший выбор во всех возможных сценариях. Flux это потрясающий шаблон, но все еще имеет некоторые масштабные нерешенные проблемы, с того времени как поддержка приложения могла быстро выйти из-под контроля из-за природы множественных store и отсутствия поддержки plug-in .

Однако, когда бы то вы не были сбиты с толку этим сложным выбором, вы можете обратиться к данному совету:

'...если вы не уверены нужно ли вам это – значит вам это не нужно'

Пит Хант, инженер команды разработки `React` из `Facebook`.

## Итоги

`Redux` похож на `Flux` но в тоже время у них много отличий. Давайте пройдемся по примеру беседы `React` и `Redux`, если бы они были друзьями:

- `React`: Эй, `changePage` действие, кто-то нажал кнопку `Go to` чтобы перейти к другой странице книги;
- `Action`: Спасибо `React`! Я запускаю действие чтобы заботливые `reducers` могли обновить состояние;
- `Reducer`: Оу, спасибо действие. Я вижу, что ты передал мне текущее состояние и действие для представления. Я сделаю новую копию состояния и верну её;
- `Store`: Спасибо за обновление состояния, мистер `reducer`. Я уведомя `React`;
- `React`: Ооо! Какие блестящие новые данные ты передал мне хранилище! Пожалуй я обновлю UI чтобы это отобразить.

Вот эта – беседа постоянно повторяется в ответ на действия пользователя.

Ключевыми игроками `Redux` являются:

- `Store` – объект в котором живет `state`, главный провайдер `Redux API`;
- `State` – изменяется в соответствии с `action`;
- `Action` – описание следующего изменения `state`;
- `Action creator` – функция, которая возвращает `action`;
- `Reducer` – управляет `action`, вычисляет и возвращает новый `state`;
- `Middleware` – обеспечивает любой вид мощной функциональности, которая может быть вставлена во внутрь `store`;
- `Enhancers` – `superstructure` которая увеличивает функциональность `store`.

Так что закругляясь мы можем сказать, что `Redux`:

- Управляет состоянием приложения;
- Предназначенный для `React`, но работает с любым типом `view` движков;
- Более строгий но Более надежный по сравнению с `Flux`;
- `Single immutable store` в его основе, которое предоставляет `dispatch`, `subscribe`, `getState` и `replaceReducer` API;
- `Reducers` часть основного жизненного цикла;
- Обновление `state` приложения управляется каждым `reducer`;
- Эволюционировавший `Flux`.

Спасибо что остаётесь с нами! Увидимся на следующем уроке, где мы обсудим более углубленно связь между `React` и `Redux` ⚡!

Если у вас есть какие либо идеи как мы можем улучшить наши уроки, пожалуйста, поделитесь с нами вашими мыслями [hello@lectrum.io](mailto:hello@lectrum.io). Ваши отзывы очень важны для нас!