3. Redux и React

- Обзор
- Container vs. presentational компонент
- Присоединение Redux к React
- Компоненты высшего порядка
- Маршрутизация в Redux
- Итоги

Обзор

Привет, друг 🙌🏻! С этого момента, мы уже знаем основы `Flux` и `Redux`. Однако, в предыдущих примерах приложения `Оскара` `book-reader`, мы использовали `Redux` прямо в компоненте `Book` и это сработало, но действительно ли нам нужно постоянно вручную `подписывать` каждый компонент к `состоянию` 😐?

К счастью нет, существует более элегантное решение! На этом уроке, наша цель выяснить:

- Различие между компонентами `container` и `presentational` ;
- Как использовать привязку `react-redux` для `присоединения` `Redux` к `React` коротко и элегантно;
- Изучить что такое 'higher-order' компоненты и для чего они нам нужны;
- Основы маршрутизации в `Redux`.

Так что первая тема у нас: какие отличия между компонентами `container` и `presentational` 🤔?

Давайте выясним 🕵🏻!

Контейнер vs. презентационный компонент

Чтобы понять как `React` присоединяется к `Redux`, важно понять оба типа компонентов `React`. В рамках `Redux`, компоненты `React` могут быть разделены на `container` 📦 и `presentational` 🖥. Однако, существуют такие же общие названия как: `Fat` и `Skinny`, `Smart` и `Dumb`, `Stateful` и `Stateless`, и так далее и тому подобное. Названия разные, но идея общая. Давайте рассмотрим компоненты `container` и `presentational` в сравнении:

Container 📦	Presentational 🖥️
Заботиться о `how things work`	Заботиться о `how things look`
Непосредственно связан с `Redux`	Не связан с `Redux`
Обеспечивает данные и поведение для `presentational` компонента или другого `container`	Получает `данные state` и `action to dispatch` исключительно с помощью `props`
не имеет разметки DOM и стилей	Имеет разметку DOM и собственные стили
Может содержать оба `components` или `presentational` компонент	Может содержать оба `components` или `presentational` компонент

`Containers` всего лишь компоненты `React`, но их использование специфическое. `Container components` содержит необходимую логику для получения `Redux` `state` и `actions`, а так же передачи их к `presentational components` с помощью `props`.

Это простое разделение поможет сделать ваши компоненты `React` легко трестируемыми, а так же легкими для повторно использования.

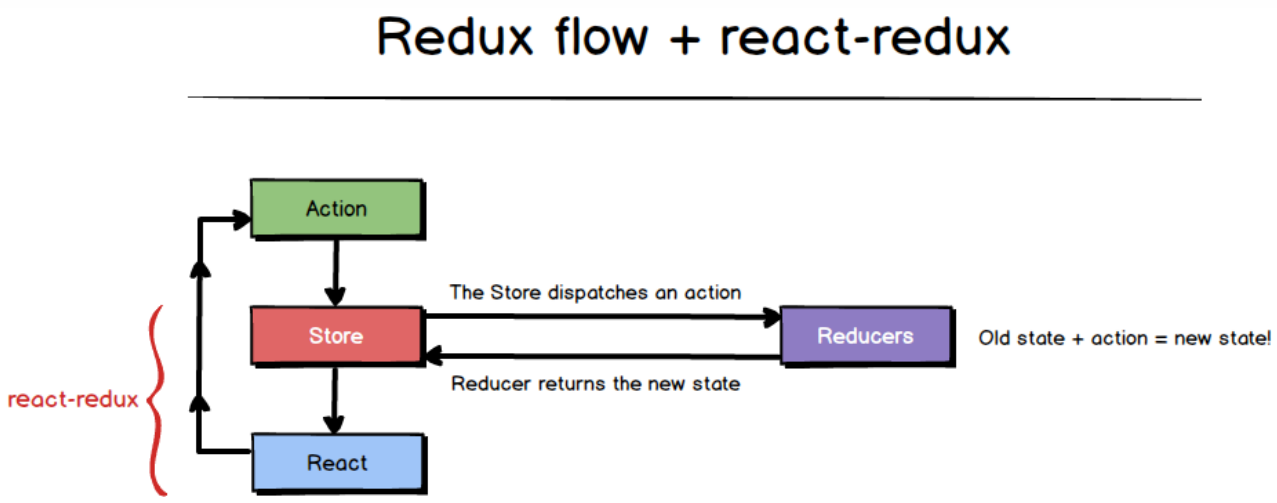
🔍 Заметка:

> Когда вложенность `presentational components` увеличивается до нескольких уровней в глубину, может быть внедрен `container component`, для того чтобы минимизировать повторяющиеся свойства, которые передаются вниз снова и снова. Как и когда лучше это сделать, специфично для каждого проекта и должно быть согласованно с вашей командой разработчиков.

Понимая это мы можем сейчас получить точное видение того что компоненты `React` могут быть `соединены` с `Redux` используя специальную связь, которая называется `react-redux`.

Присоединение Redux к React

`React-redux` – это библиотека-компаньон, которая `присоединит` наши компоненты `React` `container` к `Redux` `store`.



`React-redux` это особая библиотека потому, что `Redux` не легко присоединить к `React`. По скольку `Redux` – это всего лишь способ управления состоянием, вы можете его использовать с любой библиотекой. Вы можете написать приложение `Redux` с `React`, `Angular`, `Ember`, `Vue` или вообще без какой-либо библиотеки, используя обыкновенный `JavaScript` 👨. К примеру, так же существует `ng-redux`, для того чтобы написать приложение `Angular` с `Redux` 👰.

Из-за `immutable` природы `store`, которая означает, что каждое обновление состояния полностью возвращает 'новый' объект состояния. Если объект `old state` не ссылается на объект `current state`, тогда мы знаем, что состояние изменилось 🔬.

Это значит, что `react-redux` может просто сделать сравнении по ссылке для определения, когда нужно уведомить `React` об изменении состояния. Используется метод жизненного цикла `React` `shouldComponentUpdate` для того, чтобы быстро отвергнуть любые ненужные вычисления если ничего не изменилось ⚖️.

Однако, кроме этого пакет `react-redux` за кулисами включает в себя разнообразие сложных оптимизаций производительности, которые могут положится только на `immutable` состояния. Так что хорошие новости в том, что вы можете свободно получить улучшение производительности, благодаря не изменяемости `Redux` по дефолту. Это одно из больших преимуществ работы связки `react` + `redux` + `react-redux`. Было бы на много больше задач, которые нужно обдумать и правильно обработать, но вместо нас `react-redux` позаботится об этом ☑️.

Так что если вы чуть-чуть расстроились делая немного лишней работы поддерживая состояние `не изменяемым`, помните: `это окупается автоматически улучшенной производительностью` 🐵.

`React-redux` связывает ваш компонент `React` вместе с `Redux`, и оно состоит из `двух ключевых элементов`:

- Компонент `Provider`;

- И функция `connect()`.

Компонент `Provider`, который полностью оборачивает ваше приложение, делает `store` доступным для того, чтобы `присоединить` его к `React` используя функцию `connect()`.

```
// app/index.js

// Core
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';

// App
import Book from './components/Book';

ReactDOM.render(
    <Provider store = { store }>
        <Book />
    </Provider>,
    document.getElementById('root')
);
```

Теоретически, вам не нужно использовать компонент `Provider` как обязательный шаг. Однако, потом нужно будет передать ваш `store` во все компоненты, которым потом он может пригодится. Это будет ужас 😑.

Поэтому будучи оболочкой корневого уровня для вашего приложения, `Provider` поможет избежать этого, делая `store` доступным для всех ваших компонентов автоматически 😊.

🔍 Заметка:

> Компонент `Provider` использует `React` `context` чтобы, волшебным образом, ввести `store` во все `container` компоненты вашего приложения без явной его передачи.

Функция `connect` используется для того чтобы соединить компонент `React` с `Store` как только приложение оборачивается компонентом `Provider`. Эта функция оборачивает ваш компонент, и позволяет вам указать `части состояния` и `действия` которые вы захотите прикрепить как `свойства`.

Из предыдущего урока вы наверное помните, что в данный момент у нас есть два свойства в нашем `Redux` `state` :

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

Так что вот как мы можем извлечь части `magicBook` из `state` используя функцию `react-redux` `connect()`:

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

// Core
import React, { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import { changePage } from '../../actions';

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

        this.changePage = ::this._changePage;
    }

    _changePage (event) {
        this.props.changePage(event.target.value);
    }

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

        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>
        );
    }
}

const mapStateToProps = (state) => ({
    currentPage: state.magicBook.currentPage,
    totalPages:  state.magicBook.totalPages,
    title:       state.magicBook.title
});

const mapDispatchToProps = (dispatch) => ({
    changePage: bindActionCreators(changePage, dispatch)
});

export default connect(mapStateToProps, mapDispatchToProps)(Book);
```

В этом примере, `connect()` получает `mapStateToProps` и `mapDispatchToProps` как аргументы, они оба являются функциями. И оба эти параметра не обязательны:

- `mapStateToProps()` определяет `части состояния`, которые вы захотите извлечь в ваш компонент;
- `mapDispatchToProps()` определяет `действия`, которое вы захотите извлечь.

Когда вы определяете `mapStateToProps()`, ваш компонент подпишется к обновлениям `store`. Каждый раз когда `store` получает новый объект `состояния` из `reducers`, вызывается `mapStateToProps`. Эта функция принимает параметр `state`, который означает объект `состояния` приложения и возвращает объект. Каждое свойство объекта, которое вы определите, будет `prop` для компонента, который оборачивается с помощью `connect()`.

Вторая функция, которую мы можем передать `connect()` – это `mapDispatchToProps()`. Эта функция позволяет вам указать на то, какие `действия` мы хотим передать как `props` компонента. Она получает `dispatch` как параметр и возвращает свойства обратного вызова, которые вы хотите передать.

🔍 Заметка:

> Нет никакой магии за именами функций `mapStateToProps` или `mapDispatchToProps`. Вы можете использовать любое удобное для вас название. К примеру, это могут быть аббревиатуры, такие как `MSTP` – что подходит для `mapStateToProps`. Или `MDTP` – что могло бы подойти в свою очередь для `mapDispatchToProps` Их названия полностью зависят от вас.

`bindActionCreators` – это часть `Redux`. Для того чтобы объяснить их использование, давайте сделаем шаг назад и рассмотрим три разных пути для передачи `actions` компоненту:

1. Первый вариант просто проигнорировать, так как `mapDispatchToProps` опциональный параметр в функции `connect()`:

```
_changePage (event) {
    this.props.dispatch(changePage(event.target.value));
}

// ...the rest of the Book implementation...

export default connect(mapStateToProps)(Book);
```

Когда вы опустите его, функция `dispatch` привяжется к вашему компоненту как `prop`. Это означает что вы можете вызвать `dispatch` вручную в теле компонента и передать `action creator`.

2. Второй вариант оборачивать вручную `action creators` в `dispatch` вызывая его внутри функции `mapDispatchToProps()`:

```
_changePage (event) {
    this.props.changePage(event.target.value);
}

// ...the rest of the Book implementation...

const mapDispatchToProps = (dispatch) => ({
    changePage: (page) => dispatch(changePage(page))
});

export default connect(mapStateToProps, mapDispatchToProps)(Book);
```

В сравнении с `первым вариантом`, `второй вариант` позволяет сделать вызов в теле компонента сокращенным ценой некоторого дополнительного кода, в `mapDispatchToProps`.

🖐🏻 Совет:

> Так как в предыдущих примерах постоянно требуется написание кучи повторяющихся шаблонных кодов, мы рекомендуем использовать последний третий вариант передач `dispatch` компоненту:

3. Вы можете использовать функцию `bindActionCreators`, которая является более удобной функцией, нежели оборачивать ваш `action creators` в `dispatch` самостоятельно:

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

// ...the rest of the Book implementation...

_changePage (event) {
    this.props.changePage(event.target.value);
}

// ...the rest of the Book implementation...

const mapStateToProps = (state) => ({
    currentPage: state.magicBook.currentPage,
    totalPages:  state.magicBook.totalPages,
    title:       state.magicBook.title
});

const mapDispatchToProps = (dispatch) => ({
    changePage: bindActionCreators(changePage, dispatch)
});

export default connect(mapStateToProps, mapDispatchToProps)(Book);
```

В основном `bindActionCreators` делает автоматически тоже что и мы во `втором варианте`, но этот вариант помогает избежать избыточных, повторяющихся кодов для вас 🎊.

🔍 Заметка:

> Функция `connect()` открывает возможность тонкой настройки параметров `соединения` между `React` и `Redux` очень гибким способом. Полный список возможностей вы сможете найти на странице официального репозитория в GitHub.

Компоненты высшего порядка

`Redux` находится под глубоким влиянием функциональной концепции программирования. В функциональном программировании существует широко распространённый приём использования `функций высшего порядка`, который будет темой для детальной дискуссии нашего следующего урока 📗.

`higher-order component`, или просто `HOC` – продвинутая техника для повторного использования логики компонента.

🔍 Заметка:

> `HOC` не является частью `React` или `Redux`. Они шаблонны которые алхимически возникли из принципов функционального программирования и `React`'s и `Redux`'s композиционного характера 🤖.

Таким образом, `HOC` – функция которая берет (оборачивает) компонент как аргумент и возвращает новый компонент.

Использование `HOC` позволяет вам:

- Вторичное использование кода, логики и начальной загрузки;
- Манипулировать свойствами;
- Абстрагировать и манипулировать состояниями;
- Расширять отображение.

`Render hijacking` так называется потому что `HOC` берет контроль над выводом отображения обернутого компонента и может делать все что вы захотите:

- Читать, добавлять, исправлять, удалять элементы древа `React` возвращенные из метода `render()`;
- Отображать древо элементов в зависимости от условия;
- Оборачивать элементы для стилизации, и многое другое.

🔍 Заметка:

> React `HOC` в некотором роде эквивалентно функциям `Redux` `enhancer`, потому что они обе похожи. Так что мы можем сказать, что React `HOC` – является `component enhancer`.

К примеру, с помощью recompose библиотеки утил для `React`, вы можете использовать `HOC`'s чтобы ввести методы `жизненного цикла` в `функциональные` компоненты! Чудесно 🎃!

Этот `HOC` защищает ваш компонент `Book` от не санкционированного чтения:

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

// ...the rest of the Book implementation...

const protect = (WrappedComponent) =>
    class Enhancer extends WrappedComponent {
        render () {
            return this.props.purhcased
                ? super.render()
                : 'Sorry, you need to purchase this book first!';
        }
```

```
    };

const mapStateToProps = (state) => ({
    currentPage: state.magicBook.currentPage,
    totalPages:  state.magicBook.totalPages,
    title:       state.magicBook.title
});

const mapDispatchToProps = (dispatch) => ({
    changePage: bindActionCreators(changePage, dispatch)
});

export default connect(mapStateToProps, mapDispatchToProps)(protect(Book));
```

Обратите внимание `линия кода 24`, где `protect` `HOC` вызывается с `Book` в качестве аргумента.

Что делает `protect` `HOC` — вызывает тернарную кон4трукцию для проверки свойства `purchased` оборачиваемого компонента. Если данная книга является `purchased` – читателю легко прочитать её, но в случае если это не так, читатель получает соответствующее уведомление.

Более того, Это только начало. Существует вероятность того, что вы уже знакомы с `HOC`'s лучше нежели вы того ожидаете.

Функция `connect()` из пакета `react-redux` является `HOC` собственной персоной!

`Ты` 😺 (должен быть удивлён, вероятнее всего):

– Что 🙀? Нееет, невозможно!

`Автор` 👨‍🎨:

– Да, именно ☺️. Эй, `Оскар` не будешь ли ты так любезен помочь нам показать как `connect()` `HOC` будет выглядеть под капотом, пожалуйста 😎?

`Оскар` 🧙‍♀️:

– Ну конечно, друзья. Я здесь для того чтобы вам помочь! Дай те как мне только разогреть мою волшебную палочку 🚀 и давайте зажигать 🎸! Юхуууу... 💫✨💥🌪️🌙

Функция `connect()` воплощается в жизнь:

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

```
import store from '../../store';

// ...the rest of the Book implementation...

const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent) =>
    class extends Component {
        render () {
            return (
                <WrappedComponent
                    { ...this.props }
                    { ...mapStateToProps(store.getState(), this.props) }
                    { ...mapDispatchToProps(store.dispatch, this.props) }
                />
            );
        }

        componentDidMount () {
            this.unsubscribe = store.subscribe(this.handleChange.bind(this));
        }

        componentWillUnmount () {
            this.unsubscribe();
        }

        handleChange () {
            this.forceUpdate();
        }
    };

const mapStateToProps = (state) => ({
    currentPage: state.magicBook.currentPage,
    totalPages:  state.magicBook.totalPages,
    title:       state.magicBook.title
});

const mapDispatchToProps = (dispatch) => ({
    changePage: bindActionCreators(changePage, dispatch)
});

export default connect(mapStateToProps, mapDispatchToProps)(Book);
```

Как мы уже знаем, `connect()` – это функция, которая вводит `Redux-related` свойства в ваш компонент. Вы можете вставить `данные состояния` и `функции обратных вызовов`, которые изменят данные с помощью `запущенных действий`.

Реализация вашего пользовательского `connect()` дает нам возможность ввести компонент как последний шаг, так что люди могут его использовать как `декоратор`.

На `линии кода 8`, возвращается анонимный компонент то, что `отображает` ваш `обернутый` компонент на `линии кода 11`. Важный момент, то что `store` приложения должно быть в пределах `connect()` (взгляните на `линию кода 3`) так что бы использовать API `store`.

В реальном `react-redux` пакете `store` является заботой компонента `Provider`. Компонент `Provider` вводит `store` с помощью `React` `context` API, так что нам не нужно импортировать `store` вручную.

Так что наша реализация `connect()`:

- `линия кода 20`: `subscribe` к `store` так что бы не пропустить обновления;
- `линия кода 24`: и `unsubscribe` позже;
- `линия кода 28`: и всякий раз когда состояние `store` изменяется, выполняется перерисовка.

Вот и все! Это конечно же более ментальная модель, потому что в действительности `connect()` `HOC` так же содержит в себе кучу проверок, чтобы покрыть острые углы в производительности. Использование `forceUpdate()` таким образом, является чрезмерным.

Целью является протестировать как работает `HOC`, и глубже понять предназначение `connect()` `HOC`.

Предназначение `connect()` заключается в том, что вам не нужно думать про `subscribing` на обновления `store` или собственное выполнение оптимизации - вместо этого вы можете уточнить какие `props` вы хотите получить для вашего компонента на основаннии `Redux state`.

`Author` 👨‍🎤:

– Спасибо, `Оскар`, теперь мы готовы к перевороту 🙌!

`Оскар` 🧙‍♀️:

– Всегда готов помочь 🕯. Теперь я вернусь к своим магическим исследованиям. Позовёте меня когда понадоблюсь ☎️!

🖐 Совет:

> Шаблон `HOC` обычно используется в современном приложении `React` с `Redux`. Попробуйте на практике написать ваш собственный `HOC` – это поможет вам понять их принципы. Для хорошего начала нужно написать базовый и простой `HOC` такой как `protect` `HOC` условного отображения наподобие того,что мы написали ранее. Так что

> когда к вам придет идея – вы можете продвинутся и написать более сложный вариант. Поверьте, написание `HOC`'s это весело 🤣!

Шаблон `HOC` очень выразителен и утвердился как довольно таки полезная функция за последние годы. Более того, мы ожидаем рост популярности шаблона `HOC` 💎!

Маршрутизация в Redux

Маршрутизация существенная часть почти всех приложений. На сегодняшний день, более распространенный, стабильный, функциональный вариант маршрутизации для `React` это react-router 🤖.

`React-router v4` был полностью переписан, учитывая тот же декларативный стиль что и в `React`. Новая версия `v4` принесла новую ментальную модель `динамических роутов`, но об этом позже.

Для повышения удобства использования, у `react-router` есть специальный драйвер для `Redux`, который дает возможность содержать всю информацию о маршрутизации в `store`. Более того, становиться возможной `навигация` между `маршрутами` используя `действия`! Драйвер называется react-router-redux 🗺️.

Так же `react-router-redux` использует специальный пакет history, который дает возможность управлять сессийной историей. Это абстрагирует различия в средах окружения и обеспечивает простое API для навигации.

Для использования `react-router-redux` нужно сделать следующими шаги:

1. `root reducer` должен быть обновлён с помощью поля `router` которое содержит относящиеся к `react-router-redux` данные:

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

import { combineReducers } from 'redux';
import user from './user';
import magicBook from './magicBook';
import { routerReducer as router } from 'react-router-redux';

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

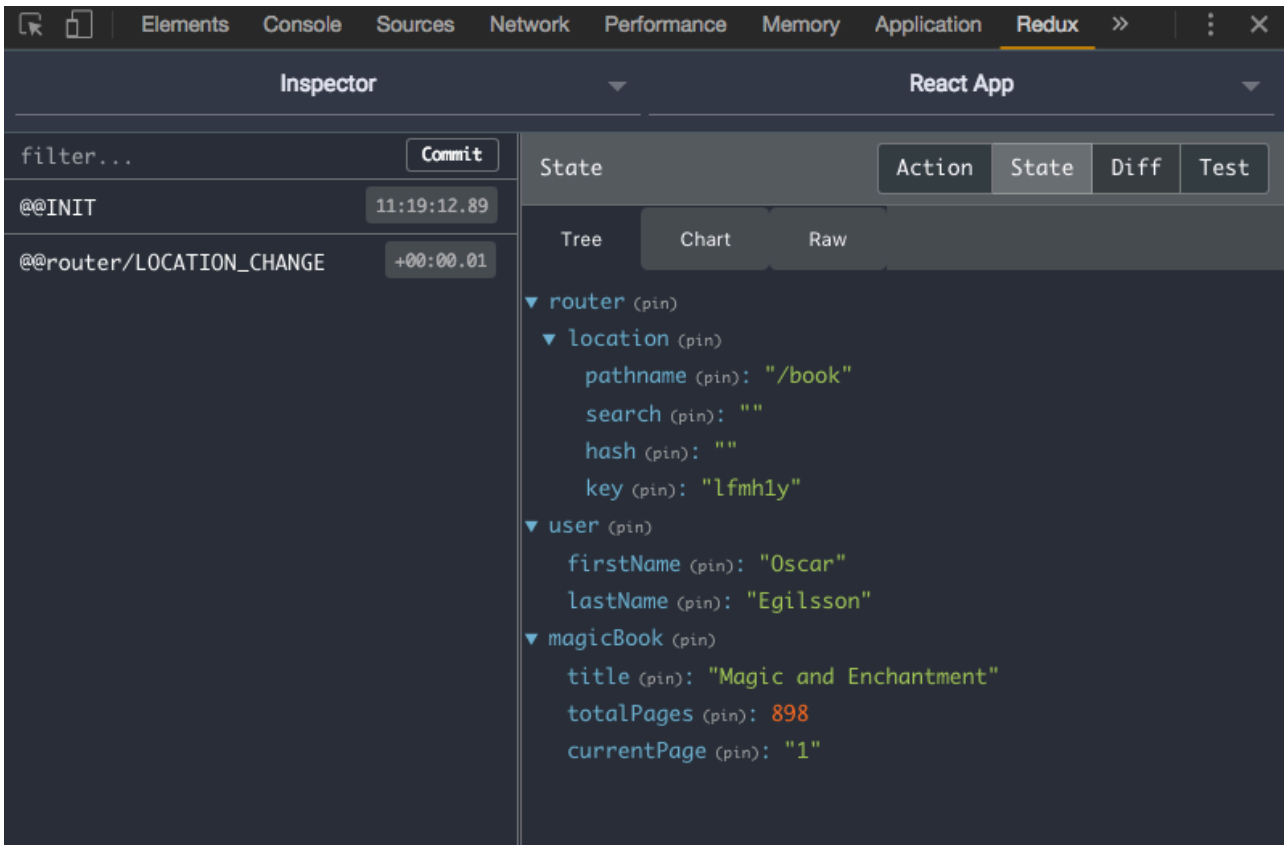
2. Новый `routerMiddleware` должен быть добавлен к `store` во время его создания, с переданной `history` в качестве аргумента (заметьте, что модуль `createBrowserHistory` находится в конструкции импорта):

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

import { createStore, applyMiddleware, compose } from 'redux';
import reducer from '../reducers';
import { routerMiddleware } from 'react-router-redux';
import createHistory from 'history/createBrowserHistory';

const history = createHistory();
const middleware = routerMiddleware(history);

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ||
compose;

export { history };
export default createStore(
    reducer,
    composeEnhancers(applyMiddleware(middleware))
);
```

3. Все приложение должно быть обернуто в дополнительный провайдер `ConnectedRouter`. Это работает так же как и `Provider` `Redux`, используя API `React` `context` чтобы ввести соответственные свойства в целое дерево приложения:

```
// app/index.js

// Core
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store, { history } from './store';
import { ConnectedRouter } from 'react-router-redux';

// App
import App from './containers/App';

ReactDOM.render(
    <Provider store = { store }>
        <ConnectedRouter history = { history }>
            <App />
        </ConnectedRouter>
    </Provider>,
    document.getElementById('root')
);
```

Великолепно! Теперь у нас есть `router` данные внутри `store`:



4. Теперь мы готовы идти дальше. Давайте создадим новый компонент `Profile`, чтобы мы могли использовать навигацию между этим компонентом и компонентом `Book`, который у нас уже есть:

```
// app/components/Profile/index.js

// Core
import React, { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import { push } from 'react-router-redux';

class Profile extends Component {
    constructor () {
        super();

        this.navigateToBook = ::this._navigateToBook;
    }

    _navigateToBook () {
```

```
        this.props.push('/book');
    }

    render () {
        const { firstName, lastName } = this.props;

        return (
            <section>
                <nav>
                    <button onClick = { this.navigateToBook }>
                        Start reading...
                    </button>
                </nav>
                <h1>
                    Welcome, {firstName} {lastName}!
                </h1>
            </section>
        );
    }
}

const mapStateToProps = ({ user }) => ({
    firstName: user.firstName,
    lastName:  user.lastName
});

const mapDispatchToProps = (dispatch) => ({
    push: bindActionCreators(push, dispatch)
});

export default connect(mapStateToProps, mapDispatchToProps)(Profile);
```

🔍 Заметка:

> Обратите внимание на конструкцию `import { push } from 'react-router-redux';`. `push` – это `создатель действия`, функция, возвращающая объект `действия`. `push` `создатель действия` используется, для изменения текущего маршрута, для того чтобы `передать` новый маршрут в браузерный маршрутный стек. Беря во внимание что `push` – простой `создатель действия`, мы можем использовать его внутри так называемого `bindActionCreators()`, у нас он присутствует в компоненте `Profile` как `prop`.

5. Финальный этап - определить систему маршрутизации. Контейнер `App` хорошо подходит для этого:

```
// app/containers/App/index.js

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

// Components
import Book from '../../components/Book';
import Profile from '../../components/Profile';

export default class App extends Component {
    render () {
        return (
            <section>
                <Route path = '/book' component = {Book} />
                <Route path = '/profile' component = {Profile} />
            </section>
        );
    }
}
```

6. Давайте посмотрим как компонент `Book` будет выглядеть:

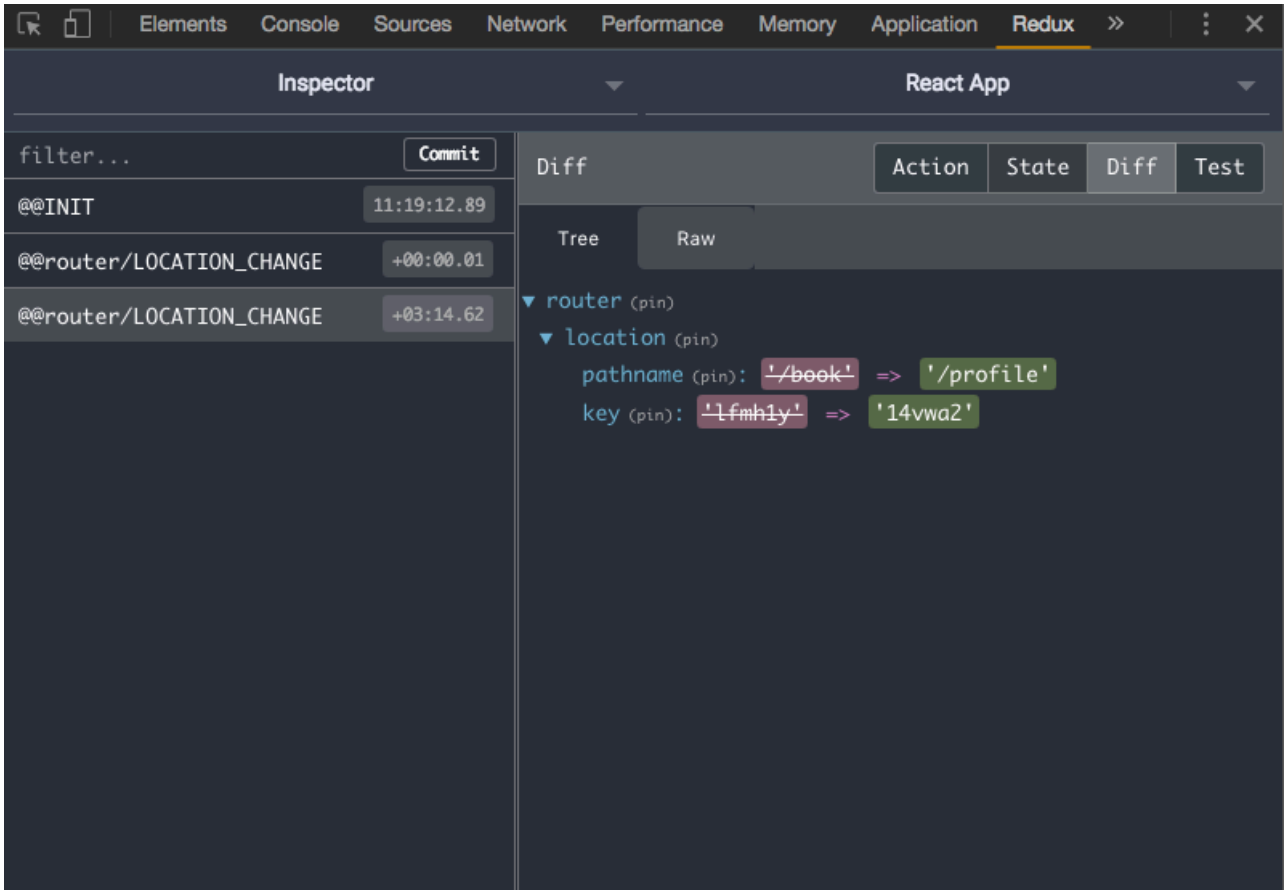
```
// Core
import React, { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import { changePage } from '../../actions';
import { push } from 'react-router-redux';

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

        this.changePage = ::this._changePage;
        this.navigateToProfile = ::this._navigateToProfile;
    }

    _navigateToProfile () {
        this.props.actions.push('/profile');
    }

    _changePage (event) {
        this.props.changePage(event.target.value);
    }
```

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

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

        return ( <section>
                <nav>
                    <button onClick = { this.navigateToProfile }>
                        To profile...
                    </button>
                </nav>
                <h1>{title}</h1>
                Go to
                <select value = { currentPage } onChange = { this.changePage
}>
                    {pagesToSelect}
                </select>
                page
                {/* current page content */}
                <p>Total pages: {totalPages}</p>
            </section>
        );
    }
}

const mapStateToProps = (state) => ({
    currentPage: state.magicBook.currentPage,
    totalPages:  state.magicBook.totalPages,
    title:       state.magicBook.title
});

const mapDispatchToProps = (dispatch) => ({
    actions: bindActionCreators({ changePage, push }, dispatch)
});

export default connect(mapStateToProps, mapDispatchToProps)(Book);
```

Невероятно! Теперь мы можем использовать навигацию между компонентами `Book` и `Profile` с помощью `react-redux-router` 🎉!

Каждый компонент отображается в браузере когда URL получит правильное значение. Когда пользователь использует навигацию к `www.book-reader.io/book` – компонент `Book` отображается. А компонент `Profile` возвращается в случае если URL становится `www.book-reader.io/profile`.

Изменения URL вызываются с помощью `dispatching` `push(ADRESS)`. Где `ADDRESS` представляет желаемый URL.



Теперь каждые изменение локации регистрируются, могут быть проанализированы, и легко проверены. Сохранение части состояния приложения такой как `routing` в `Redux` имеет большое значение. Это позволяет вам быстрее найти ошибки и создать более сложное приложение, с уникальной логикой и поведением 🦊.

Итоги

В этом уроке мы провели обзор важных различий между `container` и `presentational` компонентом: мы `присоединяем` только `container` компоненты к `Redux`. Вне `presentational` компонентов мы незнаем ничего про `Redux` – они всего лишь получают то что нужно как `свойства` 🧙.

Мы можем использовать библиотеку `react-redux` чтобы `присоединить` наши компоненты к `Redux` с помощью:

- Обертки наших компонентов в компонент `Provider`;
- и `присоединения` наших компонентов к хранилищу `Redux` с помощью функции `connect()`;
- функция `mapStateToProps()` помогает нам определить какую `часть состояния`, мы хотели бы передать нашим `container` в `props`;
- и функция `mapDispatchToProps` позволяет нам определить, какие `действия`, мы хотим передать в `props`.

И как мы видели,было три разных пути `mapping dispatch to props` но очевидно, что больше всего подходит один из них, наиболее идеальный – использование вспомогательной функции `bindActionCreators` из `Redux`.

Вы можете использовать функцию `connect()` для того чтобы `присоединить` ваши компоненты для использования `Store` просто и элегантно. Более того, `connect()` не простая функция – это `higher-order component`.

Доминантный `higher-order component` или шаблон `HOC` могут применяться для расширения функциональности других компонентов. Вы можете подумать про `HOC` как про `enhancer component`, или даже `decorator component`. Широко известным именем все еще является `higher-order component`, но идея та же.

Драйвер `react-router-redux` позволяет вам иметь все данные `роутера` в `store` внутри свойства `router`, что становиться удобным, по мере увелчиения размера приложения.

Спасибо что остаётесть с нами! Увидимся на следующем уроке 🤗!

> Если у вас есть предложения по улучшению этого урока, пожалуйста, поделитесть вашими мыслями с нами `hello@lectrum.io`. Ваш отзыв очень важен для нас!