ОСНОВЫ REDUX

LUXOFT TRAINING

LXFT LISTED NYSE

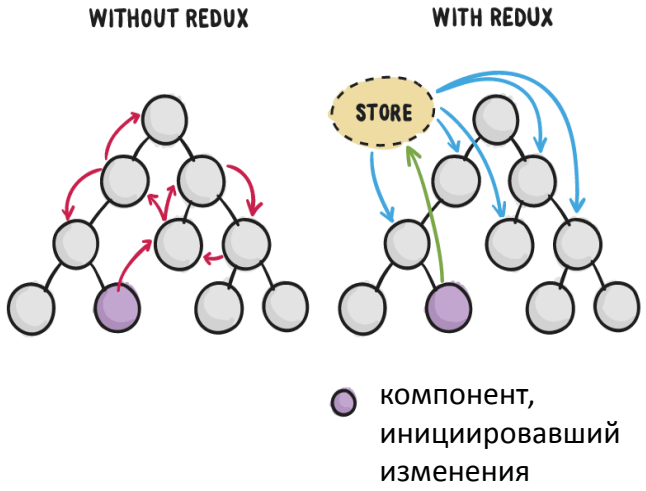
ИМПЛЕМЕНТАЦИИ АРХИТЕКТУРЫ FLUX

Существует множество готовых библиотек, уже реализующих архитектуру FLUX.

- Facebook Flux
- Fluxible by Yahoo
- Reflux
- Alt
- Flummox
- Marty.js
- McFly
- Lux
- Material Flux
- Redux
- Redux + Flambeau
- Nuclear.js
- Fluxette
- Fluxxor
- Freezer
- Fluxury

ЗАЧЕМ НАМ REDUX?

- *Если состояние хранится в компонентах, то это состояние **размазано по всему приложению.** Но одним компонентам постоянно требуется узнавать состояние других компонентов! Это порождает сложную передачу данных между компонентами, которой **очень трудно управлять**.*
- REDUX (и Flux) предлагают иметь **ОДНО** место, в котором хранится **все состояние** приложения, и откуда каждый компонент может взять для себя необходимую информацию.



REDUX

- Redux — это контейнер предсказуемого состояния данных для JavaScript-приложений.
- В Redux получают дальнейшее развитие идеи Flux, но исключается присущая Flux сложность.
- Redux нужен не для отображения элементов, не для маршрутизации или чего-то еще в этом роде. Его задача — сохранение состояния приложения.
- Основные **идеи** Redux просты и понятны:
  - Единый источник достоверных данных - Состояние приложения в целом хранится в дереве объектов в **едином хранилище**.
  - Состояние доступно только для чтения - Состояние можно изменить только одним способом — выдать действие, т. е. объект, описывающий, что произошло.
  - Изменения осуществляются с помощью чистых функций - Чтобы указать, как дерево состояний трансформируется действиями, необходимо написать чистые преобразователи данных.

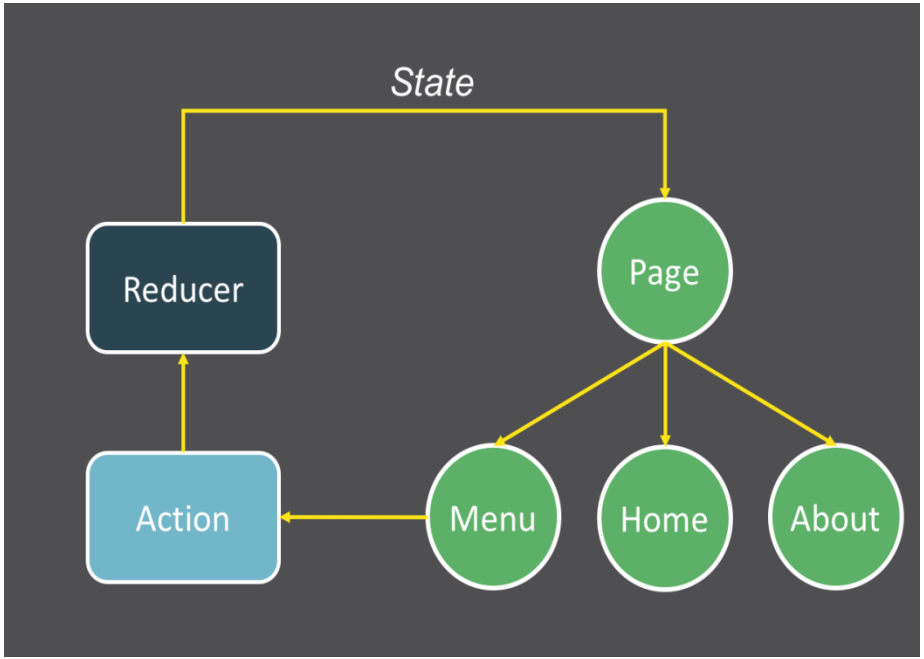
РЕДЬЮСЕР – ЧИСТАЯ ФУНКЦИЯ

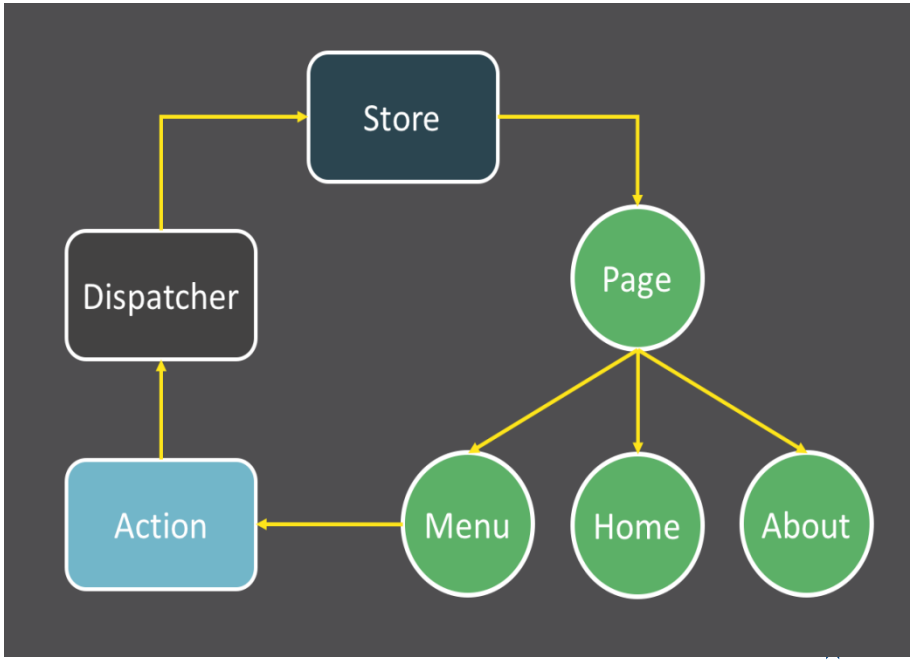
```
import { createStore } from 'redux';

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state – 1
    default:
      return state
  }
}
let store = createStore(counter);
store.subscribe(() =>
    console.log(store.getState())
);
store.dispatch({ type: 'INCREMENT' }) //1
store.dispatch({ type: 'INCREMENT' }) //2
store.dispatch({ type: 'DECREMENT' }) //1
```

REDUX

- Поток данных в Flux и поток данных в Redux
- Redux – отсутствие диспетчера, единое хранилище – State, функциональная композиция там, где Flux использует регистрацию обратных вызовов



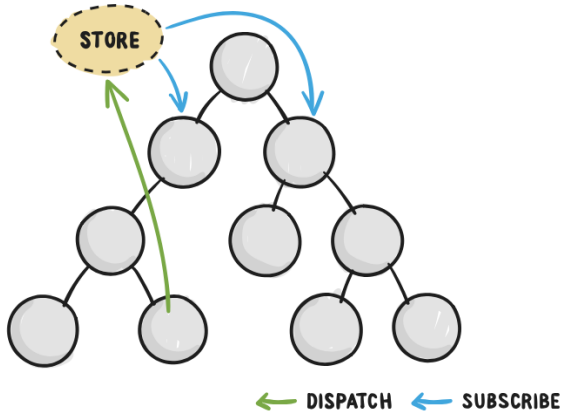


REDUX VS FLUX

- REDUX: Отсутствие диспетчера

```
var flightDispatcher = new FluxDispatcher();
flightDispatcher.dispatch({
  actionType: 'city-update',
  selectedCity: 'paris'
});

flightDispatcher.register(
  function(payload) {
    if (payload.actionType === 'city-update') {
      CityStore.city = payload.selectedCity;
    }
  });
```



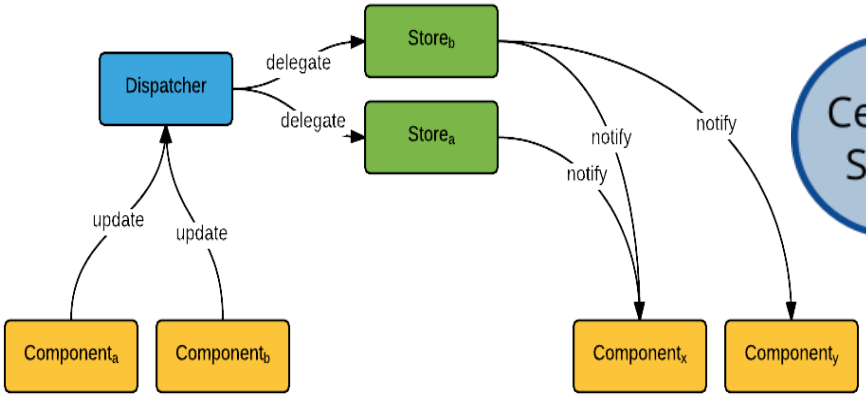
addTodo – Action Creator

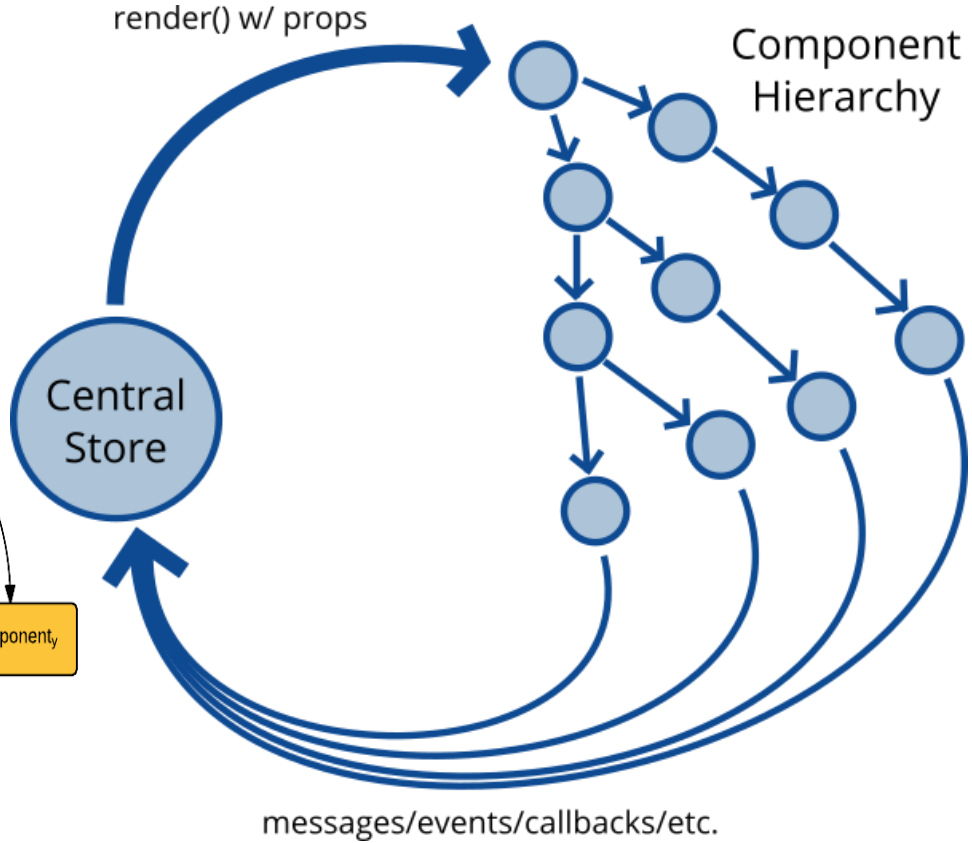
```
import {combineReducers} from 'redux'

export function addTodo(text) {
  return {
    type: ActionTypes.ADD_TODO,
    text: text
  };
}
store.dispatch(addTodo());
```

REDUX VS FLUX

- Единое хранилище/состояние





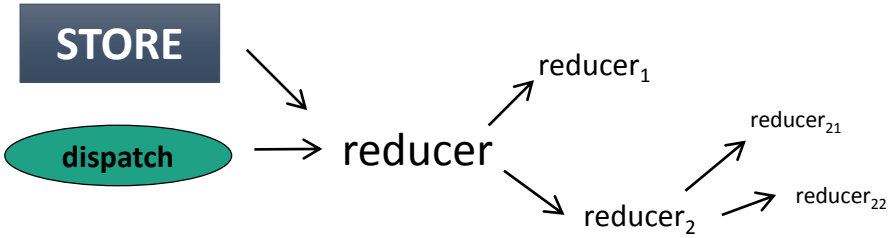
КОМПОЗИЦИЯ РЕДЬЮСЕРОВ

```
let reducer = combineReducers([
  reducer1,
  reducer2
]);
let store = createStore(reducer);
store.dispatch(ACTION);
```

Мы создаем Store, передавая туда редьюсер. Причем редьюсер можно передать **только один**. Значит ли это, что обработка любых ACTION в приложении должна выполняться в единственном редьюсере?

НЕТ!

Мы можем иметь **много редьюсеров**, объединенных с помощью ***функциональной композиции***.



Мы разбиваем главный редюсер на дочерние "подредьюсеры", которые выполняют каждый свою работу — обрабатывают какой-то один кусочек данных (срез состояния). А главный редьюсер только лишь решает, какому дочернему редьюсеру и какой срез состояния отдать.

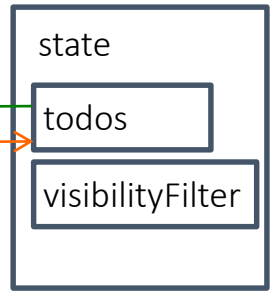
КОМПОЗИЦИЯ РЕДЬЮСЕРОВ

Функциональная композиция там, где Flux использует регистрацию обратных вызовов.

Пример:

```
const todos = (state = [], action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, action.todo]
      // создаем копию todos, изменяем ее
      // и возвращаем
    default:
      return state;
  }
};
const visibilityFilter =
  (state = 'SHOW_ALL', action) => {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter;
    default:
      return state;
  }
};
```

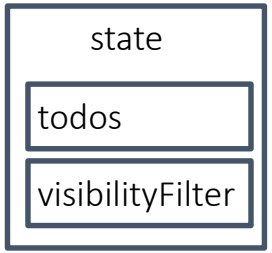
```
const todoApp =
  (state = {}, action) => {
    return {
      todos: todos(
        state.todos,
        action),
      visibilityFilter: visibilityFilter(
        state.visibilityFilter,
        action)
    };
};
```



ОБЪЕДИНЕНИЕ РЕДЬЮСЕРОВ

Выше мы уже говорили о композиции редьюсеров и о том, почему она необходима. Мы даже создали нечто похожее на утилиту Redux для объединения преобразователей:

```
const todoApp =
  (state = {}, action) => {
    return {
      todos: todos(
        state.todos,
        action),
      visibilityFilter: visibilityFilter(
        state.visibilityFilter,
        action)
    };
};
```



Но при наличии встроенного вспомогательного инструмента его можно использовать следующим образом:

```
const todoApp = combineReducers([
  todos,
  visibilityFilter
])
```

ХРАНИЛИЩЕ STORE

Хранилище в Redux представляет собой простой объект, в котором хранится состояние приложения. Хранилище выполняет следующие функции:

- Сохраняет состояние приложения:
- Предоставляет доступ к состоянию через **getState()**;
- Позволяет обновлять состояние через **dispatch(action)**;
- Регистрирует слушателей через **subscribe(listener)**;
- Выполняет отмену регистрации слушателей, используя функцию, возвращаемую подписчиком (слушателем).

Важно отметить, что в Redux-приложении будет только одно хранилище. Если необходимо разделить логику обработки данных, то вместо нескольких хранилищ используется композиция с преобразователями.

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

ДЕЙСТВИЯ

Действия — это пакеты информации, с помощью которых данных из приложения отправляются в хранилище. Они являются единственным источников информации для хранилища. Данные отправляются в хранилище с помощью store.dispatch().

Действия являются простыми объектами JavaScript. Они должны иметь свойство типа, которое указывает на тип выполняемого действия. Типы обычно определяются как строковые константы. Когда приложение становится достаточно большим, может возникнуть необходимость переместить их в отдельный модуль.

Например, вот действие, которое представляет добавление нового пункта в список дел:

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

РЕДЬЮСЕРЫ

Редьюсер объединяет хранилище и действия.

Действия описывают нечто случившееся, но не указывают, как в ответ на это меняется состояние приложения. Эту задачу выполняет **редьюсер**.

Почему эта функция называется **редьюсер**? Потому что обычно она выглядит следующим образом:

(previousState, action) => newState

Это означает, что ее можно передать в **Array.prototype.reduce(reducer, ?initialValue)**.

Очень важно, чтобы **редьюсер оставался чистой функцией**.

Чего никогда нельзя делать в редьюсере:

- Изменять его аргументы;
- Добавлять побочные эффекты, такие как вызовы API и переходы маршрутов;
- Вызывать не чистые функции, например Date.now() или Math.random().

Блок 3.
Задание 1.
Начинаем применять Redux