# MIDDLEWARE В REDUX

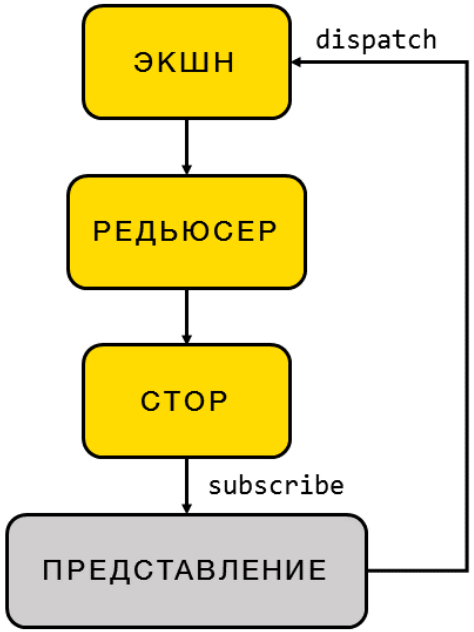
# АСИНХРОННЫЕ ACTIONS С REDUX-THUNK

LUXOFT TRAINING

LXFT LISTED NYSE

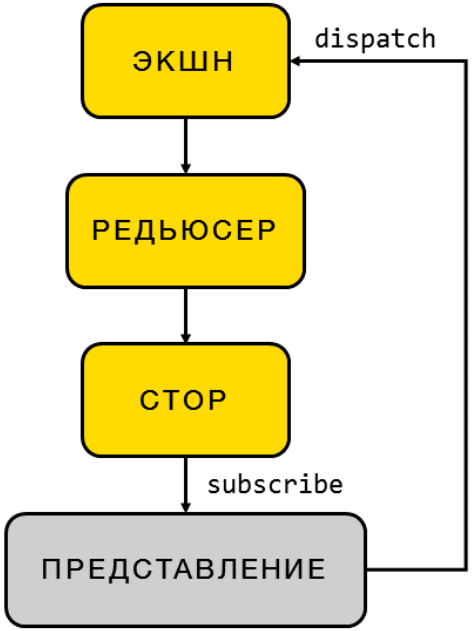
# REDUX

У нас есть прекрасная, четкая, легко отлаживаемая, поддерживаемая, понятная схема построения приложения на REDUX, основанная на идее чистых функций…

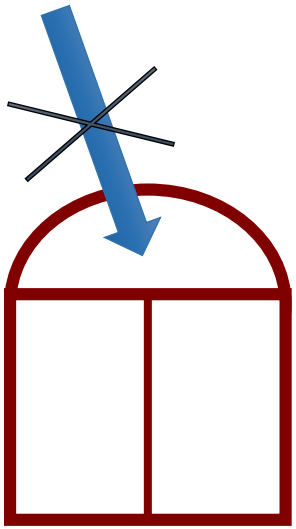


# REDUX

У нас есть прекрасная, четкая, легко отлаживаемая, поддерживаемая, понятная схема построения приложения на REDUX, основанная на идее чистых функций…




очень **чистых** функций…



**внешний мир**

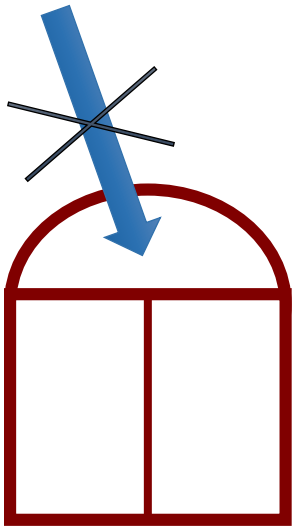
# САЙД-ЭФФЕКТЫ

**Вопрос: что делать с сайд-эффектами?**

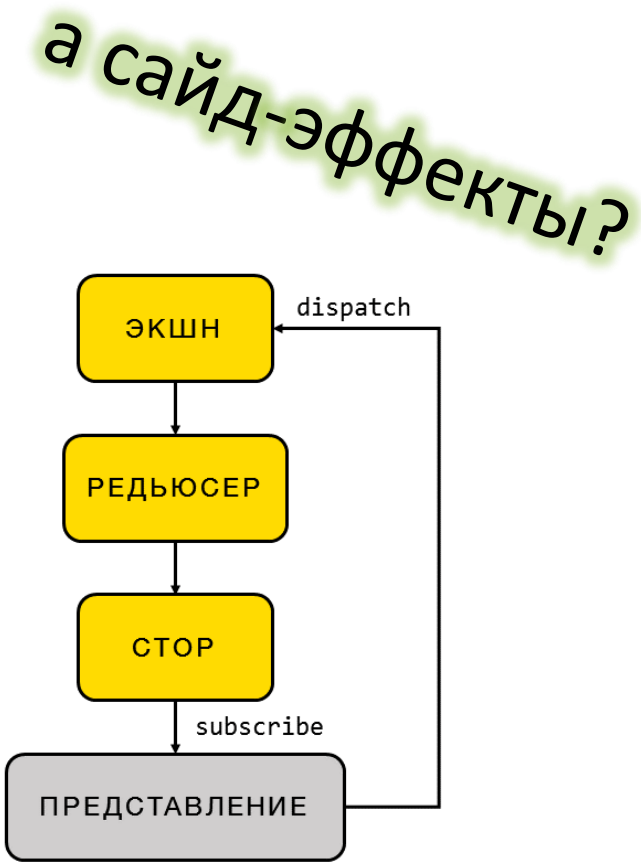
**Сайд-эффекты** — это когда вы изменяете глобальную переменную, делаете запрос на сервер, пишете что-то в лог.

Проблема в том, что в этой схеме сайд-эффектам **нет места**.

Единственное место, где происходят изменения — это **редьюсер**, а он обязан быть **чистой функцией**.

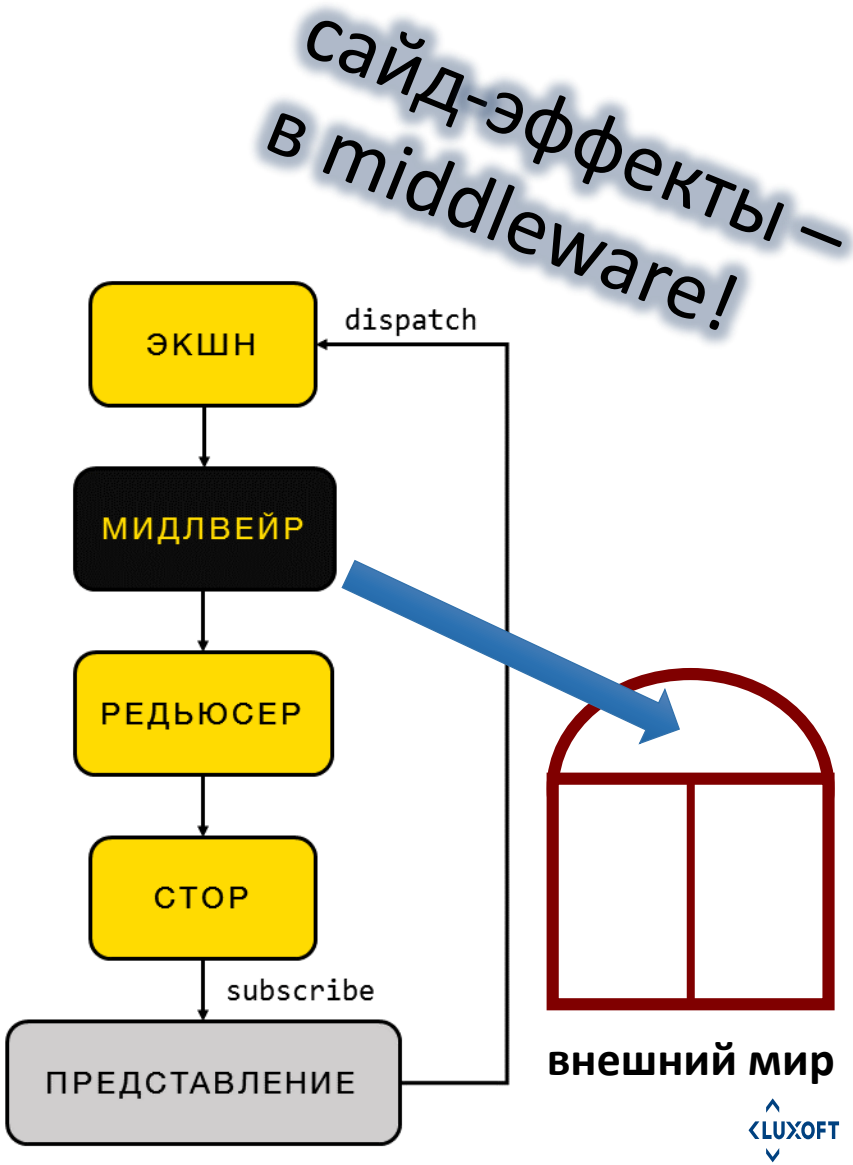


**внешний мир**



# MIDDLEWARE

На помощь приходит **middleware**.
**Middleware** находится между кодом, который диспатчит экшены (то есть генерирует события) и редьюсером (то есть функцией, которая их преобразует в новое состояние), пропускает через себя все возникающие экшены, меняет их (если считает нужным) и отправляет дальше — или не отправляет.
Особенность **middleware** в том, что он быть чистой функцией не обязан — и поэтому он может делать сайд-эффекты вроде запросов на сервер или чего угодно.



# MIDDLEWARE

На самом деле мидлвейров может быть много, и тогда они организуются в цепочки. Фактически, **middleware** — это просто функция, которая принимает текущий стор, следующий в цепочке мидлвейр и текущий экшен и что-то делает с ним.

# ТИПИЧНЫЕ ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ

- Логирование
- Оповещение об ошибках
- Работа с асинхронным API
- Роутинг
- …

# ПРЕДИСЛОВИЕ. КАРРИРОВАНИЕ

```
var greetCurried = function(greeting) {
   return function(name) {
      console.log(greeting + ", " + name);
   };
};
```

***Это небольшое улучшение к способу написания функции позволит нам создать новую функцию для любого типа приветствия и передать этой новой функции имя человека, которого мы хотим приветствовать:***

```
var greetHello = greetCurried("Hello"); //<-- выбираем приветствие и создаем функцию с приветствием "внутри"
greetHello("Heidi"); //"Hello, Heidi"
greetHello("Eddie"); //"Hello, Eddie"
```

***Мы вызываем оригинальную каррированную функцию, просто передавая каждый из параметров в отдельных круглых скобках один за другим:***

```
greetCurried("Hi there")("Howard"); //"Hi there, Howard"
```

***При этом саму функцию можно переписать в таком виде:***

```
const greetCurried = greeting => name => console.log(greeting + ", " + name);
```

***Пример каррирования – функция коннект из Redux:***

```
connect(mapStateToProps)(SomeComponent)
```

# ЗАДАЧА: ЛОГГИРОВАНИЕ: ПИШЕМ САМИ

```
let action = { type: APP_LOADED, appState: true };


console.log('prev state is ', store.getState());
console.log('dispatching ...', action);
store.dispatch(action);
console.log('next state is ', store.getState());
```

***Неудобно!***

# ЗАДАЧА: ЛОГГИРОВАНИЕ. ОБОРАЧИВАЕМ DISPATCH

```
function dispatchAndLog(store, action) {
        console.log('dispatching', action);
        store.dispatch(action);
        console.log('next state', store.getState())
}
```

***Опять неудобно - импортировать эту функцию каждый раз, когда нужно вызвать dispatch***

# ЗАДАЧА: ЛОГГИРОВАНИЕ. ПОДМЕНЯЕМ DISPATCH

```
let next = store.dispatch

store.dispatch = function dispatchAndLog(action) {
        console.log('dispatching', action)
        let result = next(action)
        console.log('next state', store.getState())
        return result
}
```

***А если нам надо несколько действий выполнять, а не только логировать?***

# ЗАДАЧА: ЛОГГИРОВАНИЕ. ПРЯЧЕМ ПОДМЕНУ DISPATCH

```
function logger(store) {
        let next = store.dispatch;
        // ранее было так:
        // store.dispatch = function dispatchAndLog(action) {
        return function dispatchAndLog(action) {
                console.log('dispatching', action)
                let result = next(action)
                console.log('next state', store.getState())
                return result
        }
}
```

# ЗАДАЧА: ЛОГГИРОВАНИЕ. ПРЯЧЕМ ПОДМЕНУ DISPATCH

```
function applyMiddlewareByMonkeypatching(store, middlewares) {
        middlewares = middlewares.slice()
        middlewares.reverse()
        // Изменяем функцию dispatch каждым мидлваром.
        middlewares.forEach(middleware =>
                store.dispatch = middleware(store)
        )
}
```

```
applyMiddlewareByMonkeypatching(store, [ logger, /*какой-либо еще обработчик*/])
```

# ЗАДАЧА: ЛОГГИРОВАНИЕ. ИЗБАВЛЯЕМСЯ ОТ ПОДМЕНЫ DISPATCH

```
function logger(store) {
        return function wrapDispatchToAddLogging(next) {
                return function dispatchAndLog(action) {
                        console.log('dispatching', action)
                        let result = next(action)
                        console.log('next state', store.getState())
                        return result
                }
        }
}
```

# ЗАДАЧА: ЛОГГИРОВАНИЕ. КАРРИРУЕМ

```
const logger = store => next => action => {
        console.log('dispatching', action)
        let result = next(action)
        console.log('next state', store.getState())
        return result
}
```

***Profit!***

# ЗАДАЧА: ЛОГГИРОВАНИЕ. ПРИМЕНЯЕМ LOGGER

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

let rootReducer = combineReducers(reducers);
let store = createStore(rootReducer,
        // applyMiddleware() говорит createStore() как обрабатывать мидлвары
        applyMiddleware(logger)
)
```

***Done!***

# ПРОБЛЕМА – СИНХРОННОСТЬ

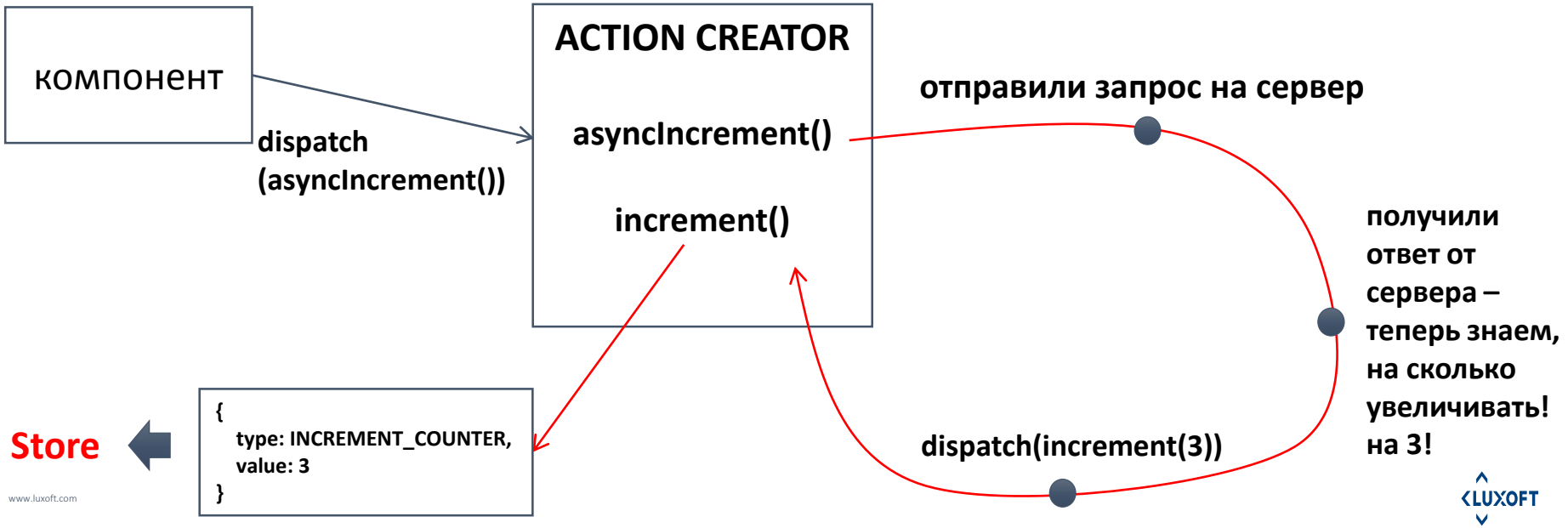
- Хранилище Redux поддерживает только синхронный поток данных
- Как быть. Если есть необходимость совершать запросы к данным, или производить длительные операции?

***Асинхронный middleware***

# Redux-thunk

- Позволяет из Action Creator вместо конкретного Action вернуть функцию
- Когда Action Creator вернет функцию, она будет обработана redux-thunk
- Теперь мы можем диспатчить новые actions прямо в action creator!

**Задача:** допустим, нам нужно увеличить (increment) какой-то параметр.
Но на сколько именно, знает только сервер!
А значит, нам надо **дождаться ответа от сервера**, а потом уже увеличивать…



# REDUX-THUNK. ПРИМЕР ACTION CREATOR

***в компоненте – первый dispatch - асинхронный:***

```
dispatch(asyncIncrement());
```

***в Action Creator – второй dispatch – уже синхронный:***

```
const asyncIncrement = () =>
        dispatch =>
                axios.get('/inc',  inc => dispatch(increment(inc)));
```

**теперь уже синхронно отправляем action в Store**

*пришел ответ от сервера: увеличиваем на 3!*

**Store**

```
function increment(inc) {
        return { type: INCREMENT_COUNTER, value: inc };
 }
```

# REDUX-THUNK. ЧТО ЕЩЕ МОЖНО ДЕЛАТЬ?

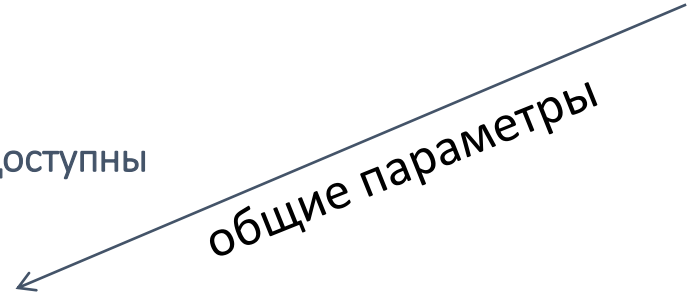
Передавать композицию функций

```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

function incrementAsync() {

    return dispatch => {

        setTimeout(() => {

            //Передаем несколько ACTION

            dispatch(increment());

            dispatch(refreshSmth());

        }, 1000);

    };

}
```

# REDUX-THUNK. ЧТО ЕЩЕ МОЖНО ДЕЛАТЬ?

## Пробрасывать дополнительные параметры

```
const store = createStore( reducer, applyMiddleware(thunk.withExtraArgument(param)) )

...

// в Action Creator параметры теперь доступны

function fetchSmth(id) {

    return (dispatch, getState, param) => { if(param) dispatch (doSmth()); }

}
```




доступ к state в коллбэке

**Блок 3.**
**Задание 3.**
**Используем Redux Thunk**