**Три принципа**

Подведём итог. Что главное в Redux:

* Single source of truth — используя Redux, мы работаем только с одним хранилищем на приложение. Всё состояние в одном месте
* State is read-only — единственный способ изменить состояние – послать действие внутрь хранилища
* Changes are made with pure functions — внутри хранилища можно использовать *только* чистые функции, благодаря этому мы можем "путешествовать во времени"
* Ниже полный пример использования Redux:
* **import** { createStore } **from** 'redux';
* *// Редьюсер – функция, которая описывает то, как изменяются данные внутри хранилища*
* *// Она принимает на вход текущее состояние приложения и должна вернуть новое*
* *// Именно так работает функция reducer, отсюда и название, но оно может быть любым*
* *// Второй параметр описывает действие, с его помощью мы узнаем*
* *// как конкретно надо обновить данные для конкретного вызова*
* *// action — это объект, в котором обязательно есть поле type, содержащее имя действия*
* **const** reducer **=** (state **=** 0, action) **=>** {
* **switch** (action.type) {
* **case** 'INCREMENT':
* **return** state **+** 1;
* **case** 'DECREMENT':
* **return** state **-** 1;
* **default**: *// действие по умолчанию – возврат текущего состояния*
* **return** state;
* }
* };
* *// Создание хранилища на основе редьюсера*
* *// Именно в этом хранилище находится состояние, которое возвращает редьюсер*
* **const** store **=** createStore(reducer);
* *// Состояние можно извлечь с помощью функции getState()*
* store.getState(); *// 0 – так как это начальное значение состояния*
* *// Функция subscribe позволяет подписываться на изменение состояния внутри хранилища*
* *// Она очень похожа на addEventListener, но без указания события*
* *// Как только меняется любая часть состояния, хранилище вызывает переданную функцию*
* *// Здесь мы просто извлекаем состояние и печатаем его на экран*
* store.subscribe(() **=>** console.log(store.getState()));
* *// dispatch – функция, которая вызывает редьюсер*
* *// Редьюсер увеличивает состояние на единицу*
* store.dispatch({ type: 'INCREMENT' }); *// 1*
* *// Редьюсер увеличивает состояние на единицу*
* store.dispatch({ type: 'INCREMENT' }); *// 2*
* *// Редьюсер уменьшает состояние на единицу*
* store.dispatch({ type: 'DECREMENT' }); *// 1*
* store.getState(); *// 1*
* *// Для избежания дублирования и повышения уровня абстракции, вынесем действия в функции*
* **const** increment **=** () **=>** ({ type: 'INCREMENT' });
* **const** decrement **=** () **=>** ({ type: 'DECREMENT' });
* store.dispatch(increment()); *// 2*
* store.dispatch(decrement()); *// 1*

Работает это так: для каждого свойства верхнего уровня пишется свой собственный редьюсер, а затем они с помощью функции combineReducers() объединяются в корневой (root) редьюсер, который уже используется для создания хранилища. Экшены попадают во все редьюсеры, собранные внутри combineReducers().

**import** { combineReducers, createStore } **from** 'redux';

**const** todosReducer **=** (state **=** [], action) **=>** {

*// сюда попадут данные из todos*

};

**const** commentsReducer **=** (state **=** [], action) **=>** {

*// сюда попадут данные из comments*

};

**const** rootReducer **=** combineReducers({

todos: todosReducer,

comments: commentsReducer,

});

**const** store **=** createStore(rootReducer);

*// Если назвать редьюсеры как свойства в состоянии, то код можно сократить:*

*// const todos = (state = [], action) => { ... };*

*// const comments = (state = [], action) => { ... };*

*// const rootReducer = combineReducers({ todos, comments });*

Так как каждый редьюсер имеет доступ только к своей части состояния, действия, порождающие изменения сразу в нескольких местах, будут повторяться в разных редьюсерах:

**const** todos **=** (state **=** {}, action) **=>** {

**switch** (action.type) {

**case** 'TODO\_REMOVE':

*// ...*

}

};

**const** comments **=** (state **=** {}, action) **=>** {

**switch** (action.type) {

*// При удалении ToDo нужно удалить все его комментарии*

**case** 'TODO\_REMOVE':

*// ...*

}

};

То есть правильный подход состоит в том, чтобы повторять часть case в нужных редьюсерах, а не в том, чтобы пытаться получить недостающие части состояния.

import \_ from 'lodash';

import { combineReducers } from 'redux';

const comments = (state = {}, action) => {

  // BEGIN (write your solution here)

  switch (action.type) {

    case 'TASK\_COMMENT\_ADD': {

      const { comment } = action.payload;

      return { ...state, [comment.id]: comment };

    }

    case 'TASK\_COMMENT\_REMOVE': {

      return \_.omit(state, action.payload.id);

    }

    case 'TASK\_REMOVE': {

      const { id } = action.payload;

      return \_.omitBy(state, (c) => c.taskId === id);

    }

    default:

      return state;

  }

  // END

};

const tasks = (state = {}, action) => {

  // BEGIN (write your solution here)

  switch (action.type) {

    case 'TASK\_ADD': {

      const { task } = action.payload;

      return { ...state, [task.id]: task };

    }

    case 'TASK\_REMOVE': {

      const { id } = action.payload;

      return \_.omit(state, id);

    }

    default:

      return state;

  }

  // END

};

export default combineReducers({

  comments,

  tasks,

});

# Мидлвары — JS: Redux (React)

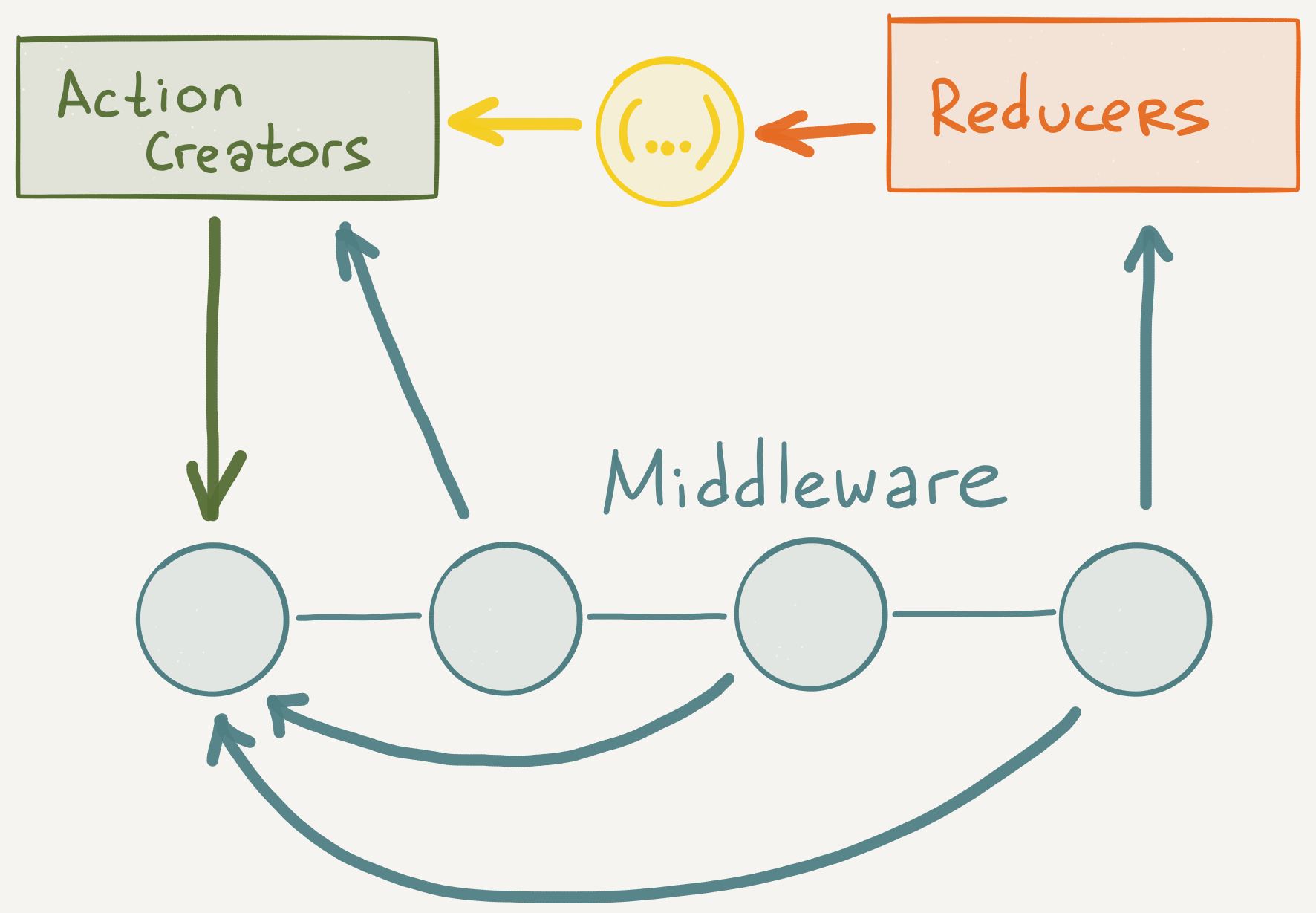
**Мидлвары** (middlewares) — это функции, которые последовательно вызываются в процессе обновления данных в хранилище.

**Как работают мидлвары**

Общий принцип такой:

* Сначала мидлвары встраиваются в хранилище при его создании
* Затем во время диспатчинга (отправки действий) данные проходят через мидлвары и только затем попадают в редьюсер

Схематично этот принцип можно показать так:



Такая организация позволяет программистам расширять библиотеки новой функциональностью, не переписывая исходный код Redux под конкретную задачу.

Мидлвары используются в таких задачах, как:

* Логирование
* Оповещение об ошибках
* Работа с асинхронным API
* Маршрутизация

У всех мидлвар одинаковая структура из трех компонентов. Рассмотрим ее на примере выше:

1. **Внешняя функция**

Именно она является мидлварой и передается в метод applyMiddleware(). Функция получает на вход объект store, который содержит методы dispatch() и getState() для работы с флоу Redux.

1. **Первая вложенная функция**

Её аргументом будет особая функция next(). Вызов этой функции в теле мидлвары с действием в качестве аргумента может прокидывать действие дальше по цепочке мидлвар.

Но если next() вызван в последней мидлваре в цепочке (цепочка может состоять и из одной мидлвары), она диспатчит действие, отправляя его в редьюсер и вызывая обновление стейта.

1. **Вторая вложенная функция**

Это функция, замыкающая в себе действие action при его диспатчинге. Всякое действие в приложении, отправляемое в редьюсер, будет перехватываться мидлварой.

Если мидлвар несколько, придется воспользоваться еще одной функцией:

**import** { createStore, applyMiddleware, compose } **from** 'redux';

**import** thunk **from** 'redux-thunk';

**import** logger **from** 'redux-logger';

**const** store **=** createStore(

reducer,

*/\* preloadedState, \*/*

compose(

applyMiddleware(thunk),

applyMiddleware(logger)

),

)

# Подробнее о работе слайсов — React: Redux Toolkit

Начнем погружаться в Toolkit со слайсов. Сразу с главного. Что бы мы ни делали внутри слайсов, в конце концов они генерируют обычные редьюсеры и действия, которые затем передаются в Redux. То есть слайсы не добавляют никаких новых возможностей в сам Redux. Они автоматизируют рутину, сокращают количество кода и предоставляют более удобные "ручки" для управления действиями и состоянием.

Для создания слайса нам нужно как минимум три компонента: имя, начальное состояние, набор редьюсеров.