

# Лекция 10

# Redux

Проектирование систем и продуктовая веб-разработка

Канев Антон Игоревич

# Доработки фронта

## Авторизация в 7ой лабораторной

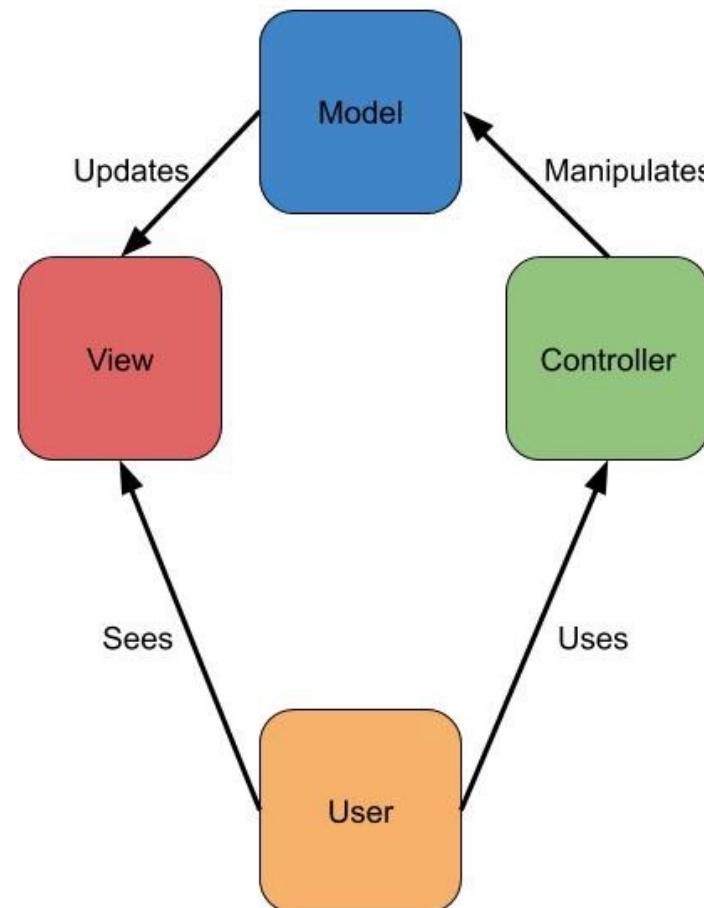
- Добавить окна регистрации и аутентификации
- Добавить логику проверки авторизации пользователя – после успешной авторизации меняется **состояние приложения**
- Авторизованный пользователь может разлогиниться
- Авторизованному пользователю доступен больший объем функционала в зависимости от его роли

## В бой лабораторной

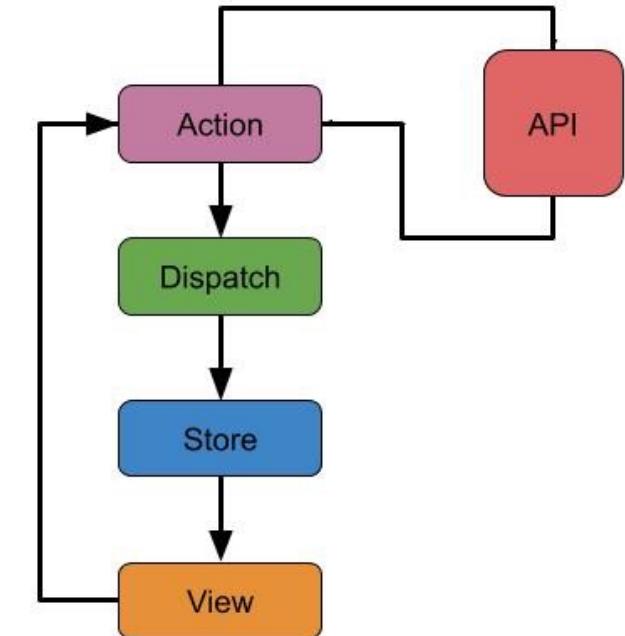
- Поля поиска, сложные формы с дополнительными полями и тд

# MVC vs Flux

- Нам нужен аналог **MVC** чтобы получать изменение одной **модели Store** в разных **представлениях** (компонентах)
- Менеджеры состояний: Redux для React, Vuex для Vue.js NgRx для Angular



MVC base Model

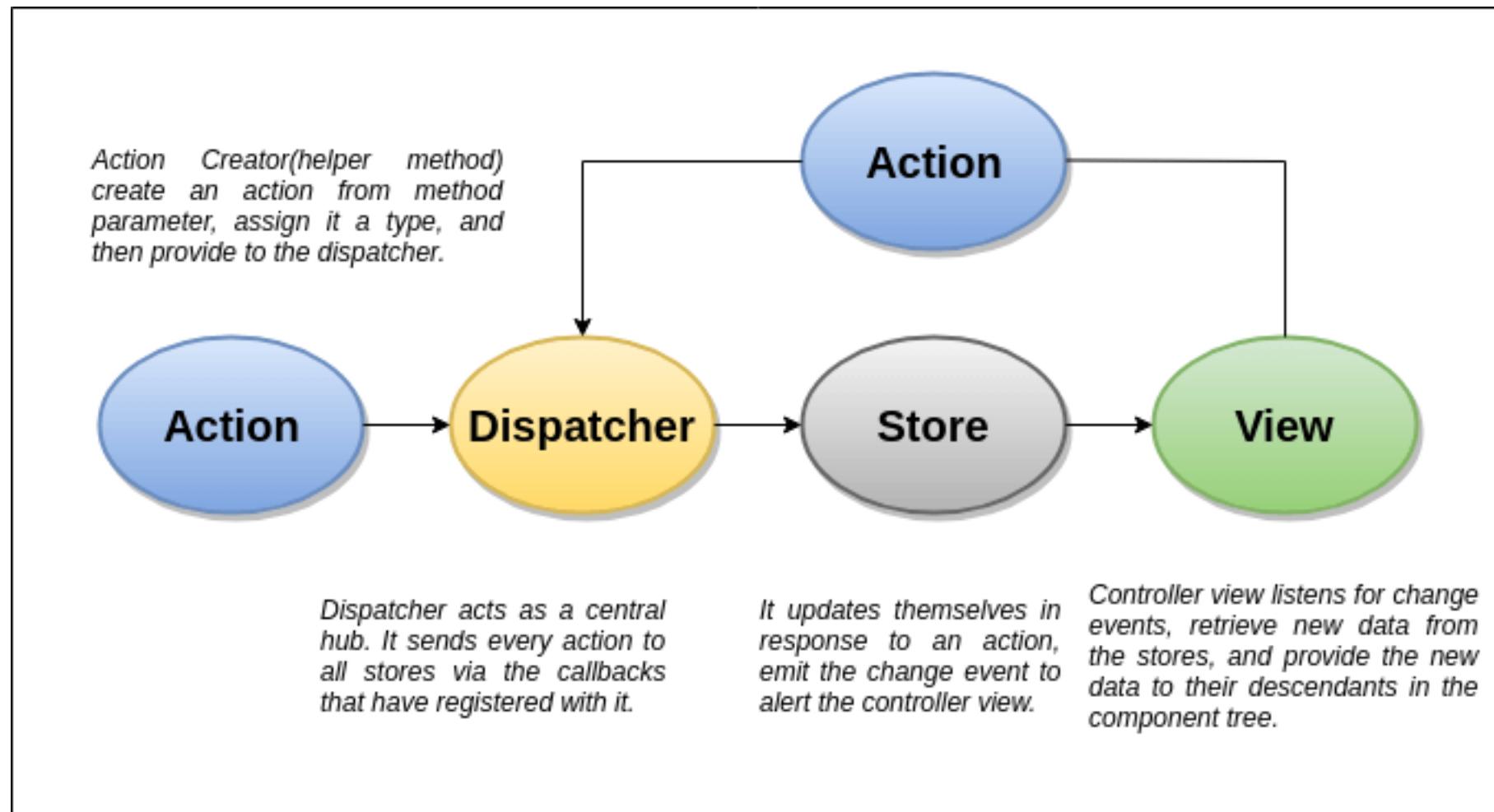


React's MVC model

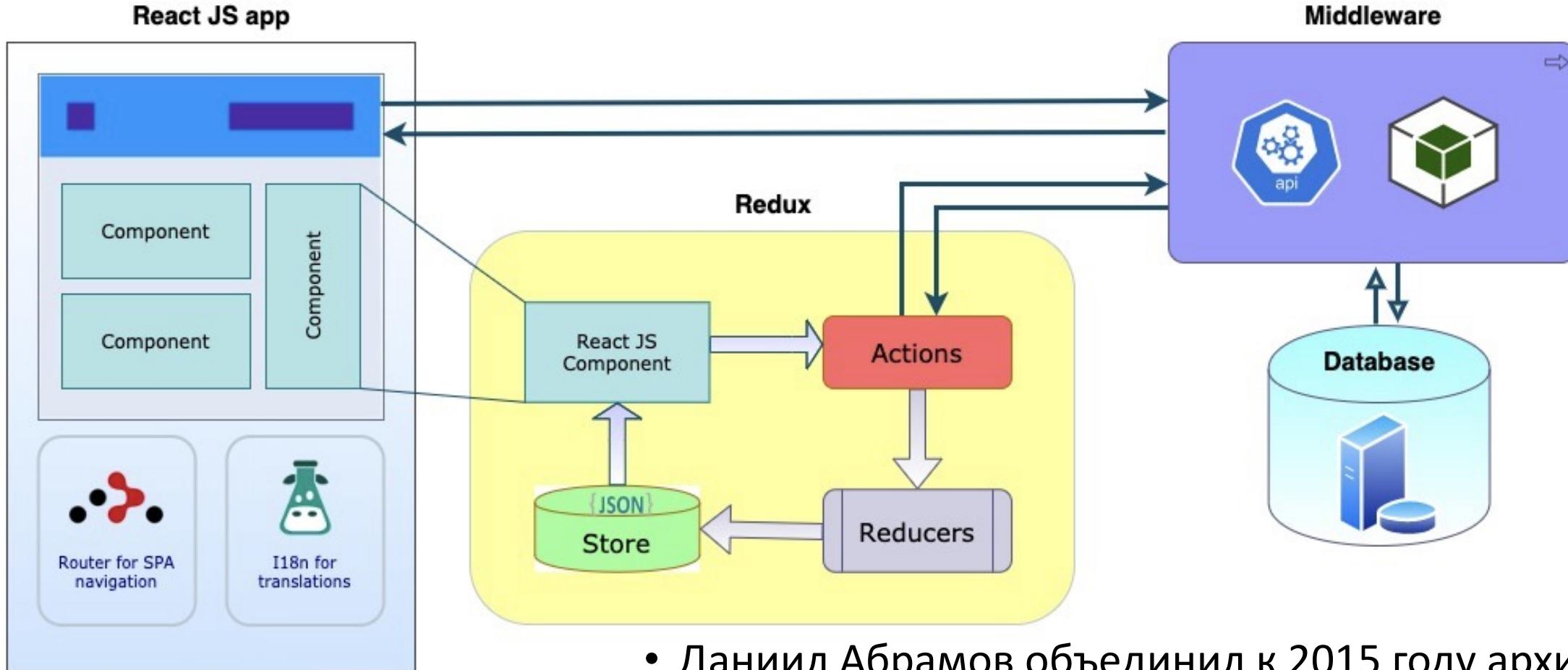
- Тут представлен вариант **общения с API** как в 7ой лабораторной

# Архитектура Flux

- Изначально разработчик React предлагал всем реализовывать у себя какую-то версию **Flux**
- Было множество реализаций
- Компонент вызывает функцию **Action**, которая меняет данные в **Store**



# React



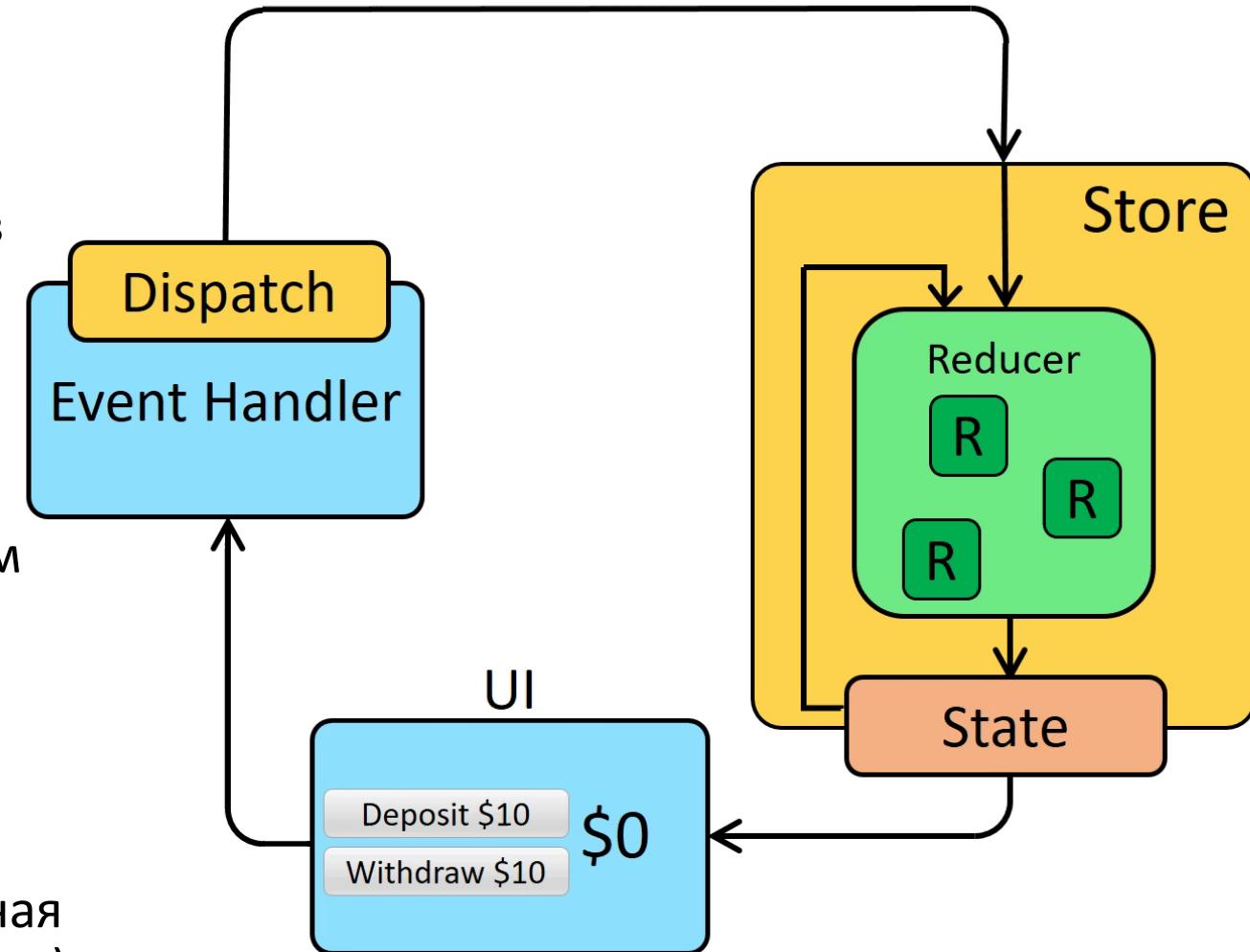
<https://react.dev/community/team>

- Общение с API через Action в 7ой лабораторной
- Общение с API через сам компонент в 5, 6ой лабораторных

- Даниил Абрамов объединил к 2015 году архитектуру flux и reducer, таким образом получился Redux
- Теперь **action** это данные, а функция в **reducer**

# React + Redux

- Компонент имеет две кнопки, каждая из которых связана со своим событием onClick
- Событие обрабатывается в функции handler, функция для кнопки Deposit создает **Action типа deposit** со значением **payload 10**
- С помощью функции **dispatch** из хука **useDispatch** это действие передается в Store
- В нем от типа действия вызывается нужная функция **Reducer** (редуктор, так не говорят)
- Эта функция **Reducer** для действия deposit получает **предыдущее значение состояния** (например amount), сейчас оно 0 и **добавляет payload 10**
- В компоненте с помощью **хука useSelector** получаем новое значение состояния amount 10 и **обновляем DOM**



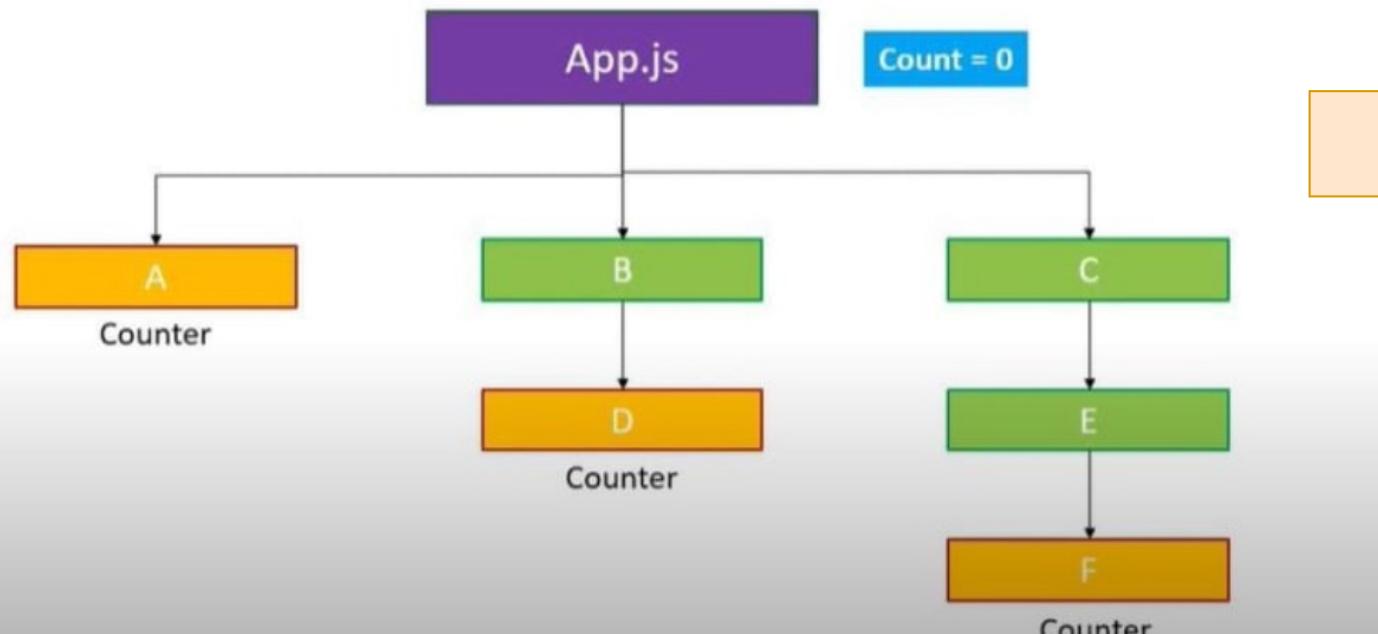
# Виды Redux

- Redux классовый
- Redux функциональный через Hooks – в лабораторной 6
- Redux Toolkit – мы используем его в лабораторной 7
- Помимо Redux есть другие контекстные менеджеры, например MobX
- Также всегда остается вариант через стандартные хуки React: useContext + useReducer
- Этот вариант появился после популярности Redux

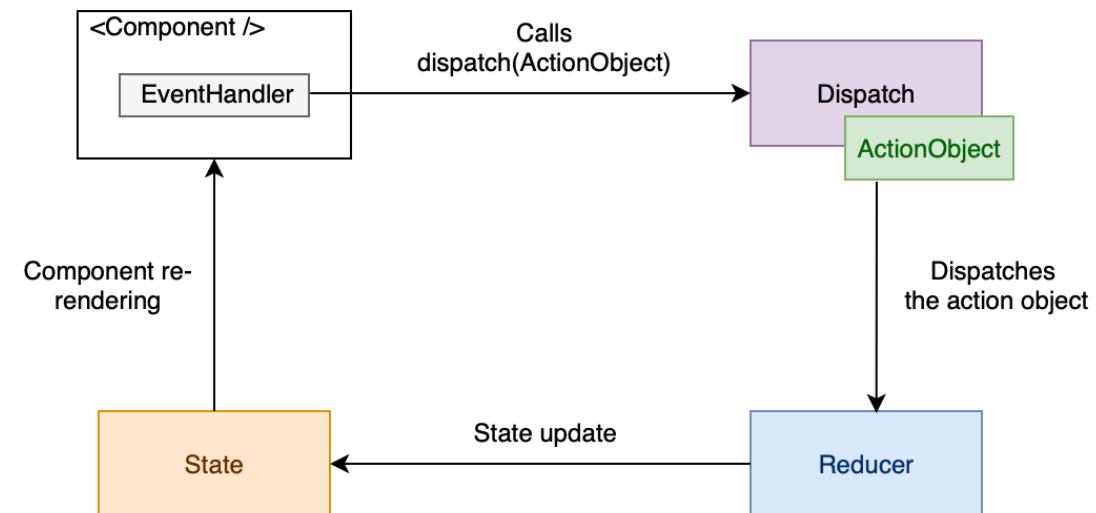
# useReducer + useContext

- **useReducer** позволяет разделить условия обновления состояния (добавление в или сброс всей корзины), передавая для изменения не значение а действие **action**

## useReducer with useContext



## useReducer()



- **useContext** позволяет нам создать единое состояние на все приложение

# Базовый пример Redux Toolkit

- Опишем пример простого Slice
- Здесь уже все наши основные составляющие: reducer, action, initialState

```
const dataSlice = createSlice({
  name: "data",
  // в initialState мы указываем начальное состояние нашего глобального хранилища
  initialState: {
    Data: []
  },
  // Редьюсеры в слайсах мутируют состояние и ничего не возвращают наружу
  reducers: {
    setData(state, {payload}) {
      state.Data = payload
    }
  }
})

export const useSelector = () =>
  useSelector((state) => state.ourData.Data)

export const {
  setData: setDataAction
} = dataSlice.actions
```

# Хранилище

- Создадим store
- Подключим его в наше приложение

```
import { combineReducers, configureStore } from "@reduxjs/toolkit"
import dataReducer from "./slices/dataSlice"

export default configureStore({
  reducer: combineReducers({
    ourData: dataReducer
  })
})
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App'
import store from "./store";
import { Provider } from "react-redux";

const root = ReactDOM.createRoot(
  document.getElementById('root')
);

root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

# Сделаем корзину без API

- Опишем slice
- Опишем наши данные и начальные состояния для них
- Состав корзины и ее стоимость

```
const dataSlice = createSlice({
  name: "data",
  initialState: {
    Data: [
      {
        "id": 1,
        "title": "Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops",
        "price": 109,
        "category": "men's clothing",
        "rating": {
          "rate": 3.9,
          "count": 120
        }
      },
      {
        "id": 2,
        "title": "Mens Casual Premium Slim Fit T-Shirts",
        "price": 22,
        "category": "men's clothing",
        "rating": {
          "rate": 4.1,
          "count": 259
        }
      }
    ],
    SumShoppingCart: 0,
  },
})
```

# Reducer, action, selector

- Укажем какие action каким reducer соответствуют
- Опишем наши функции reducer для изменения данных в store – указываем варианты для изменения наших данных
- Используем useSelector, чтобы отобразить изменения из нашего store в UI

```
export const {  
    setData: setDataAction,  
    setSum: setSumAction,  
    delSum: delSumAction  
} = dataSlice.actions
```

```
reducers: {  
    setData(state, {payload}) { // изменяем состояние на полученные данные  
        state.Data = payload  
    },  
    setSum(state, {payload}) { // суммируем цены выбранных товаров  
        state.SumShoppingCart += payload  
    },  
    delSum(state) { // обнуляем сумму выбранных товаров  
        state.SumShoppingCart = 0  
    }  
}
```

```
export const useData = () =>  
    useSelector((state) => state.ourData.Data)
```

```
export const useSum = () =>  
    useSelector((state) => state.ourData.SumShoppingCart)
```

# Наш компонент

- useDispatch чтобы иницировать из event (события) изменение в store через action
- useSelector чтобы поймать изменение в store для render UI



```
export default function ShoppingCart(){  
  const dispatch = useDispatch()  
  const sum = useSum()  
  const data = useData()  
  return(  
    <div>  
      <div className="large"> Сумма заказа: { sum }</div>  
      {  
        data.map((good) =>  
          <div key={good.id}>  
            <p> { good.title } </p>  
            <p> Цена – { good.price } </p>  
            <button onClick={() => {  
              dispatch(setSumAction( good.price ))  
            }}>  
              Добавить  
            </button>  
          </div>  
        )  
      }  
      <button onClick={() => {  
        dispatch(delSumAction())  
      }}>  
        Обнулить  
      </button>  
    </div>  
  )  
}
```

# Добавим взаимодействие с API

- Добавим вызов axios в useEffect при монтировании страницы с нашим компонентом
- При этом мы только отправляем новые данные в store
- UI описан в отдельном файле и изменения в store в него придут через useSelector

```
import {useEffect} from "react";
import axios from "axios"
import {setDataAction} from "./slices/dataSlice";
import {useDispatch} from "react-redux";

export function GetData() {
    const dispatch = useDispatch()
    async function fetchData() {
        const response = await axios.get('https://fakestoreapi.com/products?limit=5') // получение данных с API
        dispatch(setDataAction(response.data))
    }
    useEffect(() => {
        fetchData()
    }, [])
}
```