React - REDUX

MVC vs FLUX
Redux
React-Redux
JavaScript + Redux
JavaScript + React + Redux
JavaScript + React + Redux + React-Redux

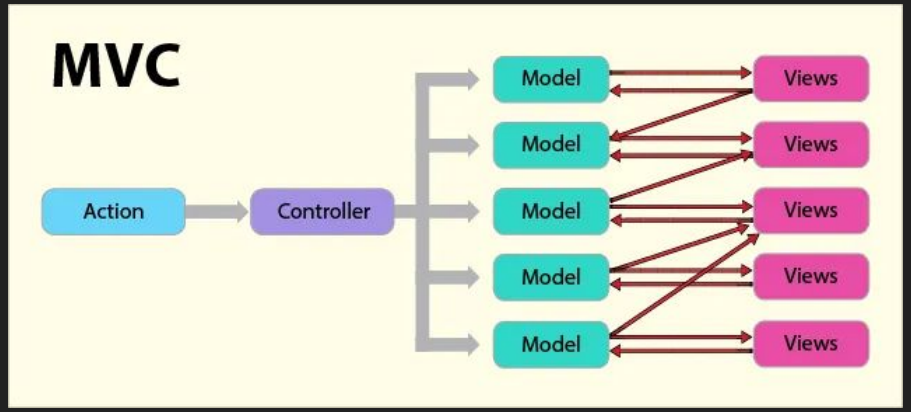
MVC

Есть различные виды MVC паттернов, но главный концепт каждого из них сводится к одному:

Model — поддерживает поведение и данные домена приложения.

View — отображает Model в пользовательском интерфейсе.

Controller — использует пользовательский ввод, управляет Model и View.



FLUX

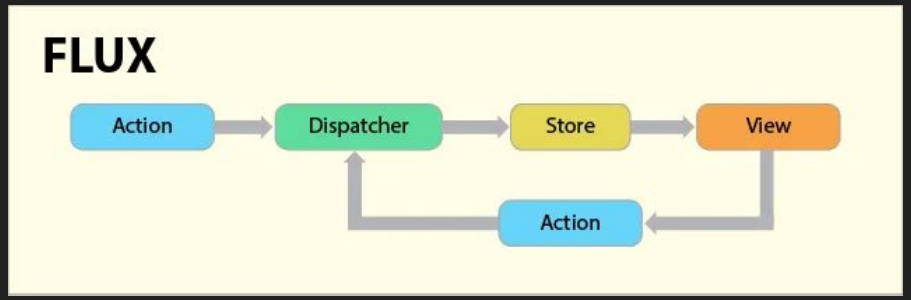
Это архитектура, ответственная за создание слоя данных в JavaScript приложениях и разработку серверной стороны в веб-приложениях. Flux дополняет составные компоненты вида View в React, используя однонаправленный поток данных.

Диспетчер (Dispatcher)

Хранилище (Stores)

Представления (Views) (React компонент)

Действие (Action)



Особенности Flux

The Flow — Flux очень требователен к потоку данных в приложении. Dispatcher данных устанавливает строгие правила и исключения для управления потоком. В MVC нет такой вещи, и потоки реализуются по-разному.

Однонаправленный поток в Flux — в то время как MVC двунаправленный в своем потоке, во Flux все изменения проходят через одно направление, через Dispatcher данных. Store не может быть изменено само по себе, и тот же самый принцип работает для других Actions. Изменения, которые необходимо внести, должны пройти через Dispatcher, через Actions.

Store — в то время как MVC не может моделировать отдельные объекты, Flux может делать это для того, чтоб хранить любые связанные с приложением данные.

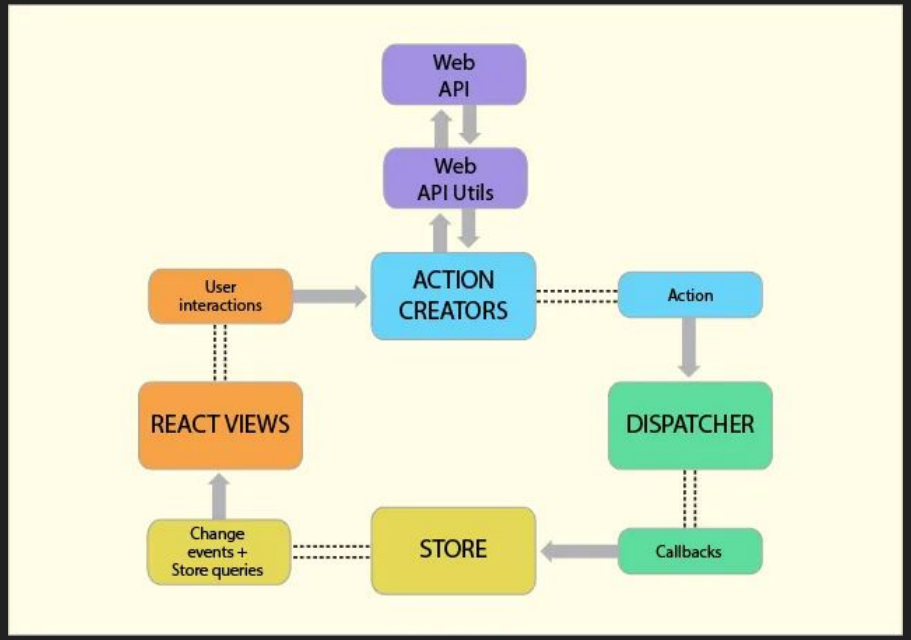
В стандартной архитектуре Flux следующие компоненты:

Actions — помощники, которые передают данные в Dispatcher

Dispatcher — получает эти действия и передает полезную нагрузку зарегистрированным callback-ом.

Stores — действуют как контейнеры для состояния приложения и логики. Реальная работа приложения происходит в Stores. Stores, зарегистрированные для прослушивания действий Dispatcher, будут соответственно и обновлять View.

Controller Views — React компоненты захватывают состояние из Stores, а затем передают дочерним компонентам.



REDUX

Redux — это библиотека для управления состоянием приложений, которая обеспечивает предсказуемость поведения вашего приложения. Она позволяет централизованно хранить состояние и управлять им с помощью действий (actions) и функций-редьюсеров (reducers). Redux широко используется в приложениях, где требуется отслеживать глобальное состояние, и особенно популярен в связке с React, но он может быть использован с любым другим UI-решением.

Установка: ***npm install redux***

Как работает Redux

Инициализация: Вы создаете хранилище с начальным состоянием и редьюсерами.

Действия (Actions): В вашем приложении что-то происходит (например, пользователь нажимает кнопку), и вы отправляете действие с помощью dispatch().

Редьюсеры (Reducers): Редьюсеры принимают текущее состояние и действие, обрабатывают его и возвращают новое состояние.

Обновление хранилища (Store): После обработки действия редьюсером, новое состояние сохраняется в хранилище.

Подписчики (Subscribers): Все подписчики получают уведомления об изменении состояния и могут обновить интерфейс.

Хранилище (Store)

Хранилище — это объект, который содержит всё состояние приложения. Это единственный источник правды для всего приложения.

Хранилище создаётся с помощью функции createStore(), которая принимает редьюсер и начальное состояние приложения.

В хранилище можно:

- Получать текущее состояние через метод getState().
- Отправлять действия через dispatch().
- Подписываться на изменения состояния с помощью subscribe()

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

const store = createStore(reducer); // Создание хранилища
```

Действия (Actions)

Действия — это простые объекты, которые описывают события, происходящие в приложении, и передают информацию о том, как должно измениться состояние.

Действие всегда содержит обязательное поле type, которое описывает тип этого действия. Дополнительно оно может содержать другие данные, которые передаются через поле payload.

```
const incrementAction = { type: 'INCREMENT' };
const decrementAction = { type: 'DECREMENT', payload: 5 };
```

Редьюсеры (Reducers)

Редьюсер — это чистая функция, которая принимает текущее состояние и действие, и возвращает новое состояние. Важно понимать, что редьюсеры не изменяют состояние напрямую, а создают новое состояние на основе старого.

В Redux обычно есть один основной редьюсер, который может быть составлен из нескольких под-редьюсеров (с помощью combineReducers).

```
const initialState = { count: 0 };

function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - action.payload };
    default:
      return state;
  }
}
```

Метод dispatch()

Метод dispatch() используется для отправки действия в хранилище. Redux обрабатывает это действие, передаёт его в редьюсер, и затем обновляет состояние хранилища.

```
store.dispatch(incrementAction); // Отправка действия для увеличения счётчика
```

Метод getState()

Этот метод позволяет получить текущее состояние из хранилища. Он возвращает объект с текущим состоянием приложения.

```
const currentState = store.getState(); // Получение текущего состояния
```

Метод subscribe()

subscribe() позволяет подписаться на изменения состояния. Каждый раз, когда состояние меняется, функция, переданная в subscribe(), будет вызвана.

Подписка может быть удалена с помощью функции, возвращаемой из subscribe().

```
const unsubscribe = store.subscribe(() => {
  console.log('State changed:', store.getState());
});

// Чтобы отписаться:
unsubscribe();
```

React-Redux

React-Redux — это официальная библиотека, которая предоставляет привязку между React и Redux, делая интеграцию Redux в React-приложение удобной и эффективной. Основная цель React-Redux — облегчить работу с глобальным состоянием, предоставляемым Redux, и обеспечить правильное обновление компонентов React при изменении состояния.

Установка: ***npm install react-redux***

Как работает Redux

react-redux — это библиотека, которая упрощает интеграцию Redux с React. Она предоставляет несколько удобных инструментов для взаимодействия компонентов React с хранилищем Redux, избавляя от необходимости вручную подписываться на изменения состояния и диспатчить экшены.

Provider — делает Redux хранилище доступным для всех компонентов.

useSelector — позволяет получать данные из хранилища и подписываться на их изменения.

useDispatch — даёт доступ к функции dispatch для отправки экшенов.

connect — более старый способ интеграции Redux с React, который позволяет явно передавать данные и функции через пропсы.

Provider

Компонент **Provider** используется для передачи хранилища (**store**) Redux всему дереву React-компонентов. Он оборачивает всё приложение и позволяет любому вложенному компоненту получать доступ к состоянию Redux.

Благодаря **Provider**, нам не нужно вручную передавать хранилище через props каждому компоненту.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { store } from './reduxStore'; // Предположим, у нас есть готовое хранилище
import App from './App';

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

useSelector

Хук **useSelector** позволяет компонентам React получать данные из Redux-хранилища. Он принимает функцию-селектор, которая извлекает часть состояния из хранилища.

Этот хук автоматически подписывает компонент на обновления хранилища и перерисовывает его только тогда, когда выбранная часть состояния изменяется.

```
import { useSelector } from 'react-redux';

const CounterDisplay = () => {
  const count = useSelector((state) => state.counter);
  return <div>Counter: {count}</div>;
};
```

useDispatch

Хук **useDispatch** возвращает функцию dispatch, которую можно использовать для отправки действий в Redux.

Этот хук удобен для вызова действий внутри компонентов.

```
import { useDispatch } from 'react-redux';
import { incrementAction } from './reduxActions';

const IncrementButton = () => {
  const dispatch = useDispatch();
  return (
    <button onClick={() => dispatch(incrementAction())}>
      Increment
    </button>
  );
};
```

connect:

connect — это более **старый**, но всё ещё популярный способ связать Redux с компонентами React. Он позволяет передавать состояние и действия как пропсы компоненту.

connect принимает две функции:

- mapStateToProps — выбирает части состояния из хранилища, которые нужны для компонента.
- mapDispatchToProps — передаёт действия (action creators) компоненту как пропсы.

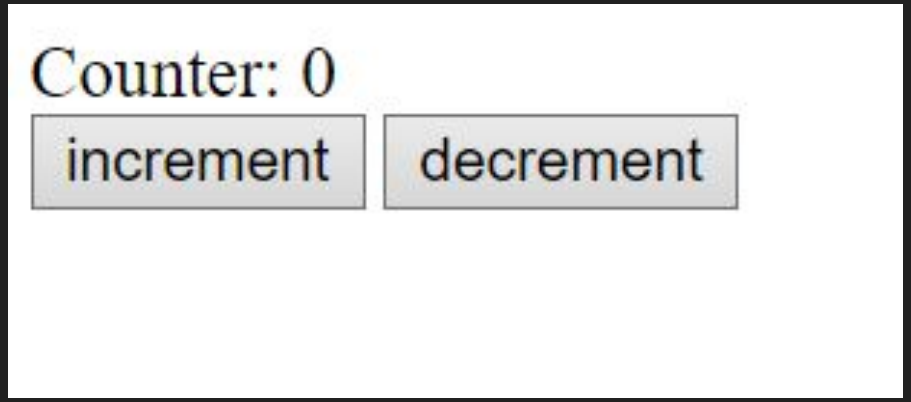
```
import { connect } from 'react-redux';
import { incrementAction } from './reduxActions';

const Counter = ({ count, increment }) => (
  <div>
    <p>Counter: {count}</p>
    <button onClick={increment}>Increment</button>
  </div>
);

const mapStateToProps = (state) => ({
  count: state.counter,
});

const mapDispatchToProps = {
  increment: incrementAction,
};

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

Пример Redux + JavaScript

Реализуем простой пример счетчика: отображение состояния счетчика и 2 кнопки - увеличение на единицу и уменьшение на единицу.

Для реализации на чистом JavaScript и Redux для контроля состояния разделим логику на две части:

- UI (JavaScript составляющая) - отвечает за связь с HTML и отрисовку
- Logic (Redux составляющая) - отвечает за предоставления реактивности и контроля логики счетчика



Для корректной работы модульности (export/import) а также для более облегченного способа добавления REDUX библиотеки использован сборщик Vite

JavaScript + HTML (UI)

```
export const APP_ID = {
  APP_ROOT: "app",
  APP_COUNTER: "app_counter",
  APP_INCERMENT: "app_increment",
  APP_DECREMENT: "app_decrement",
};
```

```
<!DOCTYPE html>
<html lang="en">
  <head> ···
  </head>
  <body>
    <div id="app">
      <div id="app_counter"></div>
      <div>
        <button id="app_increment">increment</button>
        <button id="app_decrement">decrement</button>
      </div>
    </div>
    <script type="module" src="/main.js"></script>
  </body>
</html>
```

```
import { APP_ID } from "./const";

export const AppRoot = document.getElementById(APP_ID.APP_ROOT);
export const AppCounter = document.getElementById(APP_ID.APP_COUNTER);
```

Функционал для интеграции в UI

Функция **renderCounterValue** - простая функция получает значение и записывает в элемент **AppCounter** (render UI)

Функция **setButtonClicks** - принимает 2 колбека на увеличение и уменьшение счетчика, добавляет в прослушку события (используется Делегирование событий)

```
import { APP_ID } from "./const";
import { AppCounter, AppRoot } from "./elements";

export const renderCounterValue = (value) => {
  AppCounter.innerText = `Counter: ${value}`;
};

export const setButtonClicks = ({ incrementClick, decrementClick }) => {
  AppRoot.addEventListener("click", (event) => {
    switch (event.target?.id) {
      case APP_ID.APP_INCERMENT:
        incrementClick();
        break;
      case APP_ID.APP_DECREMENT:
        decrementClick();
        break;
      default:
        break;
    }
  });
};
```

Инициализация Store

Создание **store** с помощью createStore функции из библиотеки redux, аргументом передаем редкутор в котором описана логика работы с счетчиком.

```
import { createStore } from "redux";
import { counterReducer } from "./reductor";

export const store = createStore(counterReducer);
```

Редкутор и Экшены (reducer & actions)

counterReducer - Чистая функция описывающая работу с счетчиком (увеличение на 1 и уменьшение на 1) начальное состояние счетчика 0. Все возможные действия вынесены в шаблон **ACTION_TYPES** (бойлер плейт) для удобства.

Экшены **incrementAction** & **decrementAction** - объекты передаваемые редуктуру для определения выполняемого функционала

```
const initialState = { count: 0 };

const ACTION_TYPES = {
  INCREMENT: "INCREMENT",
  DECREMENT: "DECREMENT",
};

export function counterReducer(state = initialState, action) {
  switch (action.type) {
    case ACTION_TYPES.INCREMENT:
      return { ...state, count: state.count + 1 };
    case ACTION_TYPES.DECREMENT:
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}

export const incrementAction = { type: ACTION_TYPES.INCREMENT };
export const decrementAction = { type: ACTION_TYPES.DECREMENT };
```

Функционал на экспорт

Используя функционал библиотеки Redux создаем функцию декоратор **withStoreCounterData**, которая передает колбеку актуальное состояние счетчика (состояние счетчика в момент вызова)

Функция **storeLisnter** - принимает колбек который подвязывается на подписку обновления **store**, каждый раз когда **store** обновляется/изменяется вызывается данный колбек

Две функции работы с увеличением и уменьшением счетчика - **incrementStoreHandler** & **decrementStoreHandler** - тригерят **store** на изменение и передают редуктору инструкцию по действию.

```
import { store } from "./store";
import { decrementAction, incrementAction } from "./reductor";

export const withStoredCounterData = (fn) => () => {
  const counterData = store.getState().count;
  fn(counterData);
};

export const storeLisnter = (fn) => {
  store.subscribe(fn);
};

export const incrementStoreHandler = () => store.dispatch(incrementAction);
export const decrementStoreHandler = () => store.dispatch(decrementAction);
```

Redux + React + React-Redux

Пример проекта с использованием Redux в React

Два подхода:

React + Redux

В данном случае нужно реализовывать связь реактивности react и контроля состояния redux посредством встроенных хуков в библиотеку react

React + Redux + React-Redux

При использовании доп.библиотеки React-Redux связь уже реализована за нас, более того react-redux более оптимизирован.

AppCounter (базовая верстка UI)

- Компонент для отображения счетчика в React
- Получает пропс counter и отображает его
- Есть две кнопки, на каждую подвязан соответствующий колбек
- Для оптимизации компонент обернут memo HOC

```jsx
import PropTypes from "prop-types";
import { memo } from "react";

export const AppCounterCore = ({ counter, increment, decrement }) => (
  <div>
    <div>Counter: {counter}</div>
    <div>
      <button onClick={increment}>increment</button>
      <button onClick={decrement}>decrement</button>
    </div>
  </div>
);

AppCounterCore.propTypes = {
  counter: PropTypes.number,
  increment: PropTypes.func,
  decrement: PropTypes.func,
};

export const AppCounter = memo(AppCounterCore);
```

Пример React + Redux

Состояние компонента: В useState инициализируется локальное состояние count, которое берёт начальное значение из состояния Redux-хранилища (store.getState().count).

Подписка на изменения состояния Redux: В useEffect компонент подписывается на изменения хранилища через метод store.subscribe. Каждый раз, когда состояние хранилища изменяется, вызывается коллбек, который обновляет локальное состояние count с помощью setCount.
При размонтировании компонента подписка удаляется через функцию unsubscribe().

Диспетчеризация действий: В функции incrementHandler и decrementHandler происходит отправка (диспетчеризация) действий incrementAction и decrementAction в хранилище Redux для изменения состояния (увеличение и уменьшение счётчика).

Рендер дочернего компонента: Компонент AppCounter получает текущее значение счётчика (count), а также функции для инкремента и декремента (incrementHandler, decrementHandler) через пропсы. Эти функции вызывают соответствующие действия Redux.

```
import { useEffect, useState } from "react";
import { AppCounter } from "./AppCounter";
import { store } from "@/store";
import { decrementAction, incrementAction } from "@/reducer";

export const AppCounterRedux = () => {
  const [count, setCount] = useState(store.getState().count);

  useEffect(() => {
    const unsubscribe = store.subscribe(() => {
      setCount(store.getState().count);
    });

    return () => {
      unsubscribe();
    };
  }, []);

  const incrementHandler = () => store.dispatch(incrementAction);
  const decrementHandler = () => store.dispatch(decrementAction);

  return (
    <AppCounter
      counter={count}
      decrement={incrementHandler}
      increment={decrementHandler}
    />
  );
};
```

React + Redux + React-Redux

Получение состояния из Redux: Хук useSelector используется для извлечения состояния count из Redux-хранилища. Он принимает функцию-селектор (state) => state.count, которая возвращает текущее значение счётчика из хранилища.

Получение функции для отправки действий: Хук useDispatch возвращает функцию dispatch, которая позволяет отправлять действия (actions) в Redux-хранилище.

Обработчики событий: incrementHandler и decrementHandler вызывают counterDispatcher (функцию dispatch), передавая ей действия incrementAction и decrementAction соответственно. Эти действия изменяют состояние хранилища.

Рендер дочернего компонента: Значение счётчика (count) через проп counter. Функции для инкремента и декремента (incrementHandler, decrementHandler) через соответствующие пропсы.

```jsx
import { useDispatch, useSelector } from "react-redux";
import { AppCounter } from "./AppCounter";
import { incrementAction, decrementAction } from "@/reducer";

export const AppCounterReactRedux = () => {
  const count = useSelector((state) => state.count);

  const counterDispatcher = useDispatch();

  const incrementHandler = () => counterDispatcher(incrementAction);
  const decrementHandler = () => counterDispatcher(decrementAction);

  return (
    <AppCounter
      counter={count}
      decrement={incrementHandler}
      increment={decrementHandler}
    />
  );
};
```

Редуктор, хранилище и провайдер (reducer + store + provider)

```jsx
import { Provider } from "react-redux";

export const appStoreProvider = (WrapperComponent, store) => {
  const WithStoreProvider = ({ ...props }) => (
    <Provider store={store}>
      <WrapperComponent {...props} />;
    </Provider>
  );

  return WithStoreProvider;
};
```

```jsx
import { createStore } from "redux";
import { counterReducer } from "@/reducer";

export const store = createStore(counterReducer);
```

```jsx
const initialState = { count: 0 };

const ACTION_TYPES = {
  INCREMENT: "INCREMENT",
  DECREMENT: "DECREMENT",
};

export function counterReducer(state = initialState, action) {
  switch (action.type) {
    case ACTION_TYPES.INCREMENT:
      return { ...state, count: state.count + 1 };
    case ACTION_TYPES.DECREMENT:
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}

export const incrementAction = { type: ACTION_TYPES.INCREMENT };
export const decrementAction = { type: ACTION_TYPES.DECREMENT };
```

Корневой файл JSX - APP

WrapperComponent используется для стилизации и оборачивания дочерних компонентов.

appStoreProvider используется как HOC, который добавляет Redux провайдер к компоненту.

App рендерит компоненты, работающие с Redux двумя способами, для демонстрации различий в использовании Redux напрямую и через react-redux.

```jsx
import { appStoreProvider } from "@/hoc";
import { store } from "@/store";
import PropTypes from "prop-types";
import { AppCounterRedux, AppCounterReactRedux } from "@/components";

const WrapperComponent = ({ children }) => (
  <div className="wrapper">{children}</div>
);

WrapperComponent.propTypes = {
  children: PropTypes.node,
};

const WrapperComponentWithStore = appStoreProvider(WrapperComponent, store);

const App = () => (
  <>
    <h1>JavaScrip + react + redux</h1>
    <WrapperComponent>
      <AppCounterRedux />
      <hr />
      <AppCounterRedux />
    </WrapperComponent>
    <hr />
    <h1>JavaScrip + react + redux + react-redux</h1>
    <WrapperComponentWithStore>
      <AppCounterReactRedux />
      <hr />
      <AppCounterReactRedux />
    </WrapperComponentWithStore>
  </>
);

export default App;
```

Ресурсы

Пример JavaScript + Redux - ТЫК

Пример react + redux | react + redux + redux-react - ТЫК

Документация REDUX (en) - ТЫК

Документация REDUX (ru) - ТЫК

Учебник REDUX - ТЫК

Визуализация работы REDUX - ТЫК

FLUX статья - ТЫК