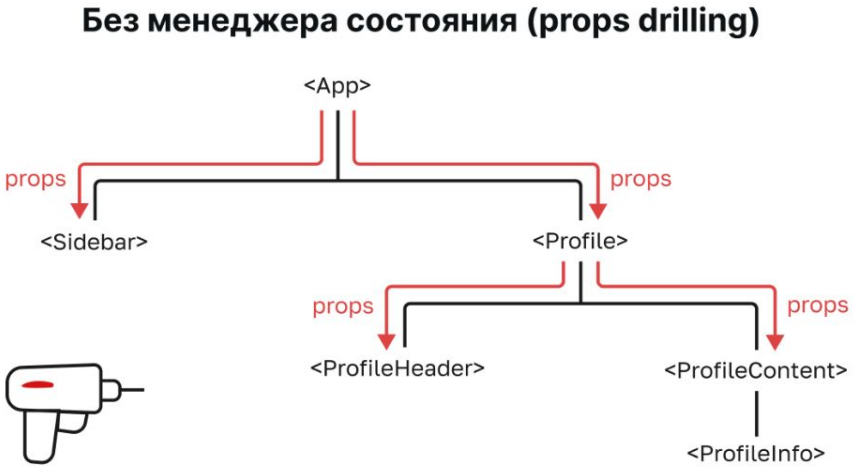
React
State Manager

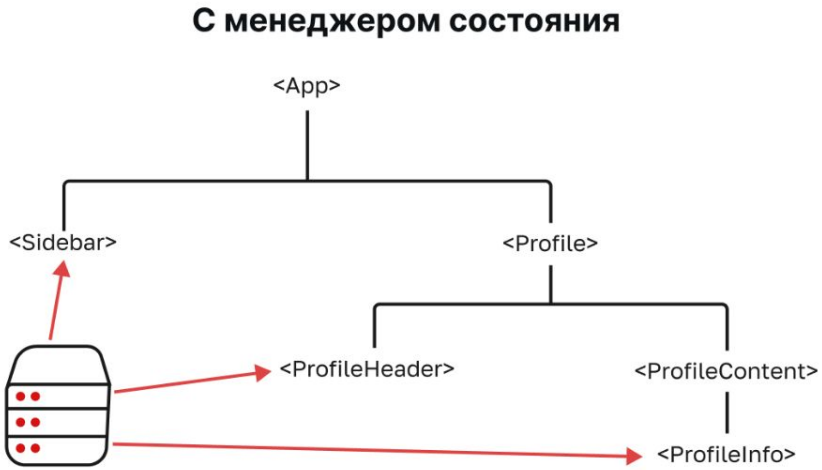
Концепция Store
useContext + useReducer
хук useSyncExternalStore
Redux | Zustand
MobX
Effector js | RxJS
XState

Концепция Store (менеджер состояний)

Это инструмент для управления глобальным состоянием приложения. С его помощью мы можем хранить state, отслеживать его изменения, а также обеспечить более гибкую архитектуру.

Без state managers работа с кодом на сложных проектах с большими объемами динамических данных будет сложнее и займет много времени.





Плюсы использования Store:

Централизация состояния: Все данные хранятся в одном месте, что делает их более управляемыми и предсказуемыми.

Явное управление данными: Состояние может изменяться только через определённые действия, что упрощает отладку.

Упрощённая синхронизация компонентов: Все компоненты могут получать состояние из единого источника, что облегчает взаимодействие между ними.

Масштабируемость: В больших приложениях, с множеством взаимодействующих частей, state manager помогает лучше контролировать логику изменения данных.

Минусы использования Store:

Избыточность: В небольших приложениях использование state manager может быть излишне сложным и привести к написанию большого количества лишнего кода.

Сложность настройки: Некоторые state managers требуют глубокого понимания архитектуры (например, Redux), что может усложнить старт разработки.

Медленная скорость разработки: Включение централизованного управления состоянием может замедлить процесс разработки, особенно если неправильно настроить или усложнить логику.

Где применять Store:

Большие приложения: Если у вас сложная логика и много компонентов, которые должны делиться состоянием, использование state manager помогает организовать код и предотвратить баги.

Многопользовательские приложения: Приложения, в которых важна синхронизация данных между различными частями интерфейса, например, чаты или социальные сети.

SPA (Single Page Applications): В одностраничных приложениях, где есть множество видов состояния (загруженные данные, пользовательские настройки и т.д.), state manager помогает управлять всеми этими данными централизованно.

В небольших приложениях state manager может быть избыточным, и можно обойтись более простыми методами управления состоянием, такими как локальное состояние компонентов.

Базовая логика для сравнения

Пример - реализация функционала TODO листа (CRUD) получить список задач, создать новую, изменить существующую в списке, удалить одну запись, очистить весь список. Сам список является ключевой точкой состояния - хранит в себе данные о списке, все функции создают новый список на основе текущего с изменениями (сохраняется имутабельность)

Состояние - состояние привязано месту (компоненту) инициализации логического конструктора в виде пользовательского хука, то есть повторный вызов хука порождает новое состояние и оба экземпляра не связаны.

Логика - логика работы основана на хуке состояния от React-а и инкапсулирована в пользовательский хук

Отображение - существует отдельный “глупый” компонент который принимает список и функции обратного вызова (callbacks) для “тригера” изменений.

```
se > use-app-todo-controller.js > ...
import { useCallback, useState } from "react";

export function useAppTodoController() {
  const [todoList, setTodoList] = useState([]);

  const addNewTodo = useCallback((newTitle) => {
    setTodoList((prev) => [
      ...prev,
      {
        id: (prev[prev.length - 1]?.id || 0) + 1,
        title: newTitle,
        status: false,
      },
    ]);
  }, []);

  const deleteTodo = useCallback((reqId) => {
    setTodoList((prev) => prev.filter(({ id }) => id !== reqId));
  }, []);

  const changeTodoStatus = useCallback((reqId) => {
    setTodoList((prev) =>
      prev.map((todo) => {
        todo.id === reqId
          ? {
              ...todo,
              status: !todo.status,
            }
          : todo;
      })
    );
  }, []);

  const clearTotoList = useCallback(() => {
    setTodoList(() => []);
  }, []);

  return [todoList, addNewTodo, changeTodoStatus, deleteTodo, clearTotoList];
}
```

```
import { useAppTodoController } from "@/use";
import { AppTodo } from "@/components";

export const AppTodoWithBasic = () => {
  const [
    todoList,
    addNewTodoHandler,
    changeTodoStatusHandler,
    deleteTodoHandler,
    clearTodoListHandler,
  ] = useAppTodoController();
  return (
    <AppTodo
      title="Basic TODO"
      todoList={todoList}
      addNewTodoHandler={addNewTodoHandler}
      changeTodoHandler={changeTodoStatusHandler}
      deleteTodoHandler={deleteTodoHandler}
      clearTodoListHandler={clearTodoListHandler}
    />
  );
};
```

useContext + useReducer (связка контекст + редуктор)

При связке данных хуков результат дает один экземпляр при инициализации редуктора и при помощи контекста передается функционал “сквозь” приложение для “тригера” единого экзепляра, что приводит к тому что в приложении есть единое состояние TODO листа

Состояние - состояние все так же привязывается к месту инициализации, но при вызове редуктора на “верхнем” уровне все компоненты “ниже” взаимодействуют с единым состоянием

Логика - логика инкапсулирована и реализуется внутри редуктора, далее редуктор делится своим функционалом с приложением посредством контекста.

Отображение - любой компонент вложенный в компонент с инициализацией имеет доступ к состоянию, что приводит к возможности отображать отдельный функционал независимо от связи “дочерних” компонентов.

Редуктор можно заменить на что угодно, за связь отвечает контекст

```jsx
import { AppTodo } from "@/components";
import { TODO_ACTIONS } from "@/const";
import { TodoContext } from "@/context";
import { useCallback, useContext } from "react";

export const AppTodoWithContext = () => {
  const [todoList, todoDispatch] = useContext(TodoContext);

  const addNewTodoHandler = useCallback(
    (newTitle) =>
      todoDispatch({ type: TODO_ACTIONS.ADD_TODO, payload: newTitle }),
    [todoDispatch]
  );

  const changeTodoStatusHandler = useCallback(
    (id) =>
      todoDispatch({ type: TODO_ACTIONS.TOGGLE_TODO_STATUS, payload: id }),
    [todoDispatch]
  );

  const deleteTodoHandler = useCallback(
    (id) => todoDispatch({ type: TODO_ACTIONS.DELETE_TODO, payload: id }),
    [todoDispatch]
  );

  const clearTodoListHandler = useCallback(
    () => todoDispatch({ type: TODO_ACTIONS.CLEAR }),
    [todoDispatch]
  );
  return (
    <AppTodo
      title="Context logic"
      todoList={todoList}
      addNewTodoHandler={addNewTodoHandler}
      changeTodoHandler={changeTodoStatusHandler}
      deleteTodoHandler={deleteTodoHandler}
      clearTodoListHandler={clearTodoListHandler}
    />
  );
};
```

```jsx
import { TodoContext } from "@/context";
import { todoReducer } from "@/reducers";
import { useReducer } from "react";

export const appTodoProvider = (WrappedComponent) => {
  return function WithAppTodoProvider() {
    const [todoList, dispatch] = useReducer(todoReducer, []);

    return (
      <TodoContext.Provider value={[todoList, dispatch]}>
        <WrappedComponent />
      </TodoContext.Provider>
    );
  };
};
```

```jsx
import { TODO_ACTIONS } from "@/const";

export function todoReducer(state, action) {
  switch (action.type) {
    case TODO_ACTIONS.ADD_TODO:
      return [
        ...state,
        {
          id: (state[state.length - 1]?.id || 0) + 1,
          title: action.payload,
          status: false,
        },
      ];
    case TODO_ACTIONS.DELETE_TODO:
      return state.filter(({ id }) => id !== action.payload);
    case TODO_ACTIONS.TOGGLE_TODO_STATUS:
      return state.map((todo) =>
        todo.id === action.payload ? { ...todo, status: !todo.status } : todo
      );
    case TODO_ACTIONS.CLEAR:
      return [];
    default:
      return state;
  }
}
```

хук useSyncExternalStore

Данный хук дает возможность реализации пользовательского Store-а для хранения состояния и работы с ним, для удобства использования данный хук оборачивается пользовательским хуком.

Состояние - состояние описывается в конфигурационном файле, но сам принцип работы с единой точкой истины предоставляется хуком useExternalStore.

Логика - логика реализована в отдельном файле который предоставляет требуемый по документации функционал для хука useSyncExternalStore, а также реализует методы для работы с самим состоянием

Отображение - любой компонент/хук в котором вызывается пользовательский хук обертка для useSyncExternalStore пердоставляет функционал и доступ к состоянию, далее его можно подвязывать под UI

createAppCustomStore

```
let state = { todoList: [] };
const listeners = new Set();

const getState = () => state;

const subscribe = (listener) => {
  listeners.add(listener);
  return () => {
    listeners.delete(listener);
  };
};

const setState = (newState) => {
  state = { ...state, ...newState };
  listeners.forEach((listener) => listener());
};
```

```
const methods = {
  addTodo: (title) => {
    setState({
      todoList: [
        ...state.todoList,
        {
          id: (state.todoList[state.todoList.length - 1]?.id || 0) + 1,
          title,
          status: false,
        },
      ],
    });
  },

  toggleTodo: (reqId) => {
    setState({
      todoList: state.todoList.map((todo) =>
        todo.id === reqId ? { ...todo, status: !todo.status } : todo
      ),
    });
  },

  deleteTodo: (reqId) => {
    setState({
      todoList: state.todoList.filter(({ id }) => id !== reqId),
    });
  },

  clearTodoList: () => {
    setState({
      todoList: [],
    });
  },
};
```

хук на основе Store

Store создает единое место хранилища состояния, но в дальнейшем нужно приминить хук useSyncExternalStore для преобразования состояния в стейт менеджер, теперь при вызове хука не создается новый экземпляр, каждый вызов кастомного хука имеет доступ к единому состоянию.

```
import { appCustomStore } from "@/store";
import { useSyncExternalStore } from "react";

export function useAppCustomStore(selector = (state) => state) {
  const subscribe = (callback) => appCustomStore.subscribe(callback);
  const getSnapshot = () => selector(appCustomStore.getState());
  const state = useSyncExternalStore(subscribe, getSnapshot);
  return {
    ...state,
    ...appCustomStore.methods,
  };
}
```

```
import { useAppCustomStore } from "@/use";
import { AppTodo } from "@/components";

export const AppTodoWithStore = () => {
  const { todoList, addTodo, toggleTodo, deleteTodo, clearTodoList }
    useAppCustomStore();
  return (
    <AppTodo
      title="Todo with Store"
      todoList={todoList}
      addNewTodoHandler={addTodo}
      changeTodoHandler={toggleTodo}
      deleteTodoHandler={deleteTodo}
      clearTodoListHandler={clearTodoList}
    />
  );
};
```

Redux (имутабельный)

Redux — это библиотека управления состоянием для JavaScript-приложений, часто используемая с **React**. Она позволяет централизованно управлять состоянием приложения, что особенно полезно в крупных проектах, где данные и события распространяются через множество компонентов.

Более удобная сборка - **ReduxToolKit**

Установка

npm install redux

npm install redux-react

Принцип работы REDUX

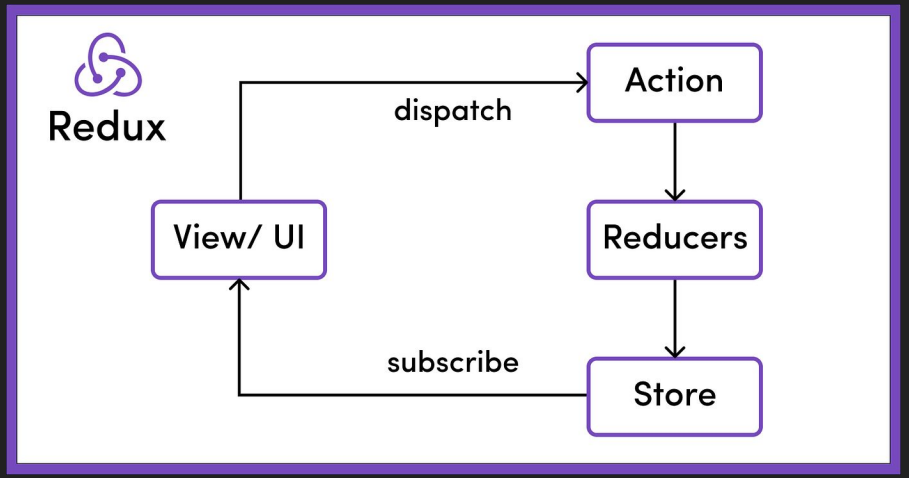
Store: Объект, который хранит всё состояние приложения.

Actions: Объекты, описывающие намерение изменить состояние. Обычно содержат тип действия и данные (payload).

Reducers: Функции, которые принимают состояние и действие, вычисляют и возвращают новое состояние.

Dispatch: Способ отправить action в хранилище.

Selectors: Функции для извлечения нужных данных из состояния.



- Компонент отправляет action с данными.
- Store передает этот action в редьюсер.
- Редьюсер возвращает новое состояние на основе action.
- Все подписанные на состояние компоненты получают обновленное состояние и перерисовываются.

Инициализация Store от Redux в проекте

```jsx
import { reduxStore } from "@/store";
import { Provider } from "react-redux";

export const appReduxProvider = (WrappedComponent) => {
  return function WithAppReduxProvider() {
    return (
      <Provider store={reduxStore}>
        <WrappedComponent />
      </Provider>
    );
  };
};
```

```jsx
import { createStore } from "redux";
import { todoReduxReducer } from "@/reducers";

export const reduxStore = createStore(todoReduxReducer);
```

```jsx
import { TODO_ACTIONS } from "@/const";

const initialState = [];

export function todoReduxReducer(state = initialState, action) {
  switch (action.type) {
    case TODO_ACTIONS.ADD_TODO:
      return [
        ...state,
        {
          id: (state[state.length - 1]?.id || 0) + 1,
          title: action.payload,
          status: false,
        },
      ];
    case TODO_ACTIONS.DELETE_TODO:
      return state.filter(({ id }) => id !== action.payload);
    case TODO_ACTIONS.TOGGLE_TODO_STATUS:
      return state.map((todo) =>
        todo.id === action.payload ? { ...todo, status: !todo.status } : todo
      );
    case TODO_ACTIONS.CLEAR:
      return [];
    default:
      return state;
  }
}
```

Использование Redux в компоненте

const todoList = useSelector((state) => state);

useSelector — хук, который позволяет получить данные из хранилища Redux. Он принимает функцию, которая извлекает нужные данные из состояния.

Здесь state — это всё состояние хранилища, и мы получаем весь список задач из него.

const dispatch = useDispatch();

useDispatch — возвращает функцию dispatch, которую можно использовать для отправки действий в хранилище Redux. Каждое действие будет обрабатываться редуктором.

```
import { AppTodo } from "@/components";
import { TODO_ACTIONS } from "@/const";
import { useDispatch, useSelector } from "react-redux";

export const AppTodoWithRedux = () => {
  const todoList = useSelector((state) => state);
  const dispatch = useDispatch();

  const addNewTodoHandler = (title) => {
    dispatch({ type: TODO_ACTIONS.ADD_TODO, payload: title });
  };

  const changeTodoStatusHandler = (id) => {
    dispatch({ type: TODO_ACTIONS.TOGGLE_TODO_STATUS, payload: id });
  };

  const deleteTodoHandler = (id) => {
    dispatch({ type: TODO_ACTIONS.DELETE_TODO, payload: id });
  };

  const clearTodoListHandler = () => {
    dispatch({ type: TODO_ACTIONS.CLEAR });
  };
  return (
    <AppTodo
      title="Basic TODO"
      todoList={todoList}
      addNewTodoHandler={addNewTodoHandler}
      changeTodoHandler={changeTodoStatusHandler}
      deleteTodoHandler={deleteTodoHandler}
      clearTodoListHandler={clearTodoListHandler}
    />
  );
};
```

Zustand (имутабельный)

Zustand — это библиотека для управления состоянием в React, которая обеспечивает простой и эффективный способ создания хранилищ состояния. Она является более легковесной и гибкой альтернативой Redux и Redux Toolkit.

Установка

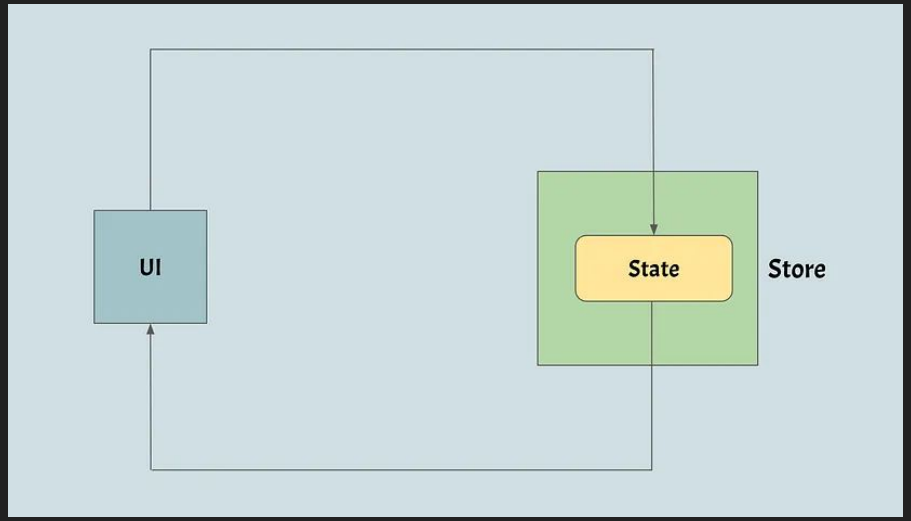
npm install zustand

Основные концепции Zustands

Хранилище состояния (Store): Zustand использует концепцию хранилища, которое управляет состоянием приложения. Хранилище создаётся с помощью функции create, которая возвращает хук, используемый для доступа к состоянию.

Селекторы: Zustand позволяет извлекать состояние и функции для изменения состояния через хук, который возвращает доступ к данным.

Иммутабельность: Zustand обеспечивает иммутабельность состояния, несмотря на то, что данные могут быть изменены через функции обновления.



Пример Zustand

Работает на основе хука ***useSyncExternalStore*** под капотом

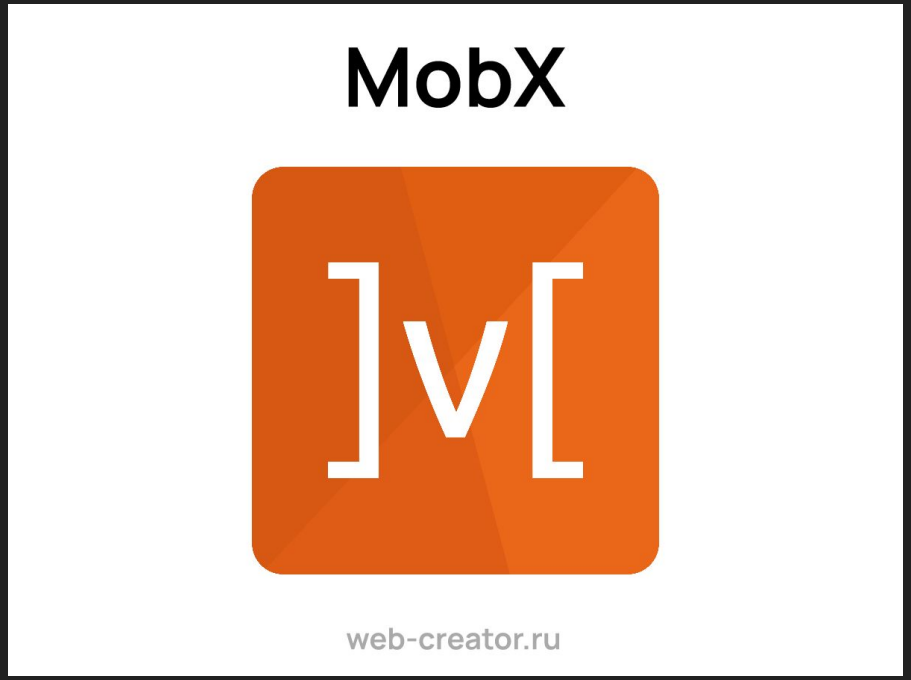
```tsx
import { useAppZustandStore } from "@/use";
import { AppTodo } from "@/components";

export const AppTodoWithZustand = () => {
  const [
    todoList,
    addNewTodoHandler,
    changeTodoStatusHandler,
    deleteTodoHandler,
    clearTodoListHandler,
  ] = useAppZustandStore();
  return (
    <AppTodo
      title="Zustand TODO"
      todoList={todoList}
      addNewTodoHandler={addNewTodoHandler}
      changeTodoHandler={changeTodoStatusHandler}
      deleteTodoHandler={deleteTodoHandler}
      clearTodoListHandler={clearTodoListHandler}
    />
  );
};
```

```tsx
import { appZustandStore } from "@/store";

export function useAppZustandStore() {
  const { todoList, addNewTodo, changeTodoStatus, deleteTodo, clearTotoList } =
    appZustandStore((state) => ({
      todoList: state.todoList,
      addNewTodo: state.addTodo,
      changeTodoStatus: state.toggleTodoStatus,
      deleteTodo: state.deleteTodo,
      clearTotoList: state.clearTodoList,
    }));

  return [todoList, addNewTodo, changeTodoStatus, deleteTodo, clearTotoList];
}
```

```tsx
import { create } from "zustand";

export const appZustandStore = create((setState) => ({
  todoList: [],

  addTodo: (title) =>
    setState((state) => ({
      todoList: [
        ...state.todoList,
        {
          id: (state.todoList[state.todoList.length - 1]?.id || 0) + 1,
          title,
          status: false,
        },
      ],
    })),

  toggleTodoStatus: (id) =>
    setState((state) => ({
      todoList: state.todoList.map((todo) =>
        todo.id === id ? { ...todo, status: !todo.status } : todo
      ),
    })),

  deleteTodo: (id) =>
    setState((state) => ({
      todoList: state.todoList.filter((todo) => todo.id !== id),
    })),

  clearTodoList: () =>
    setState(() => ({
      todoList: [],
    })),
}));
```

MobX (мутабельный)

MobX — это библиотека для управления состоянием в JavaScript и TypeScript приложениях, обеспечивающая простой и реактивный способ работы с состоянием. Она ориентирована на наблюдение за изменениями состояния и автоматическое обновление компонентов, когда состояние изменяется. MobX основан на принципах реактивного программирования.



Установка
npm install mobx
npm install mobx-react-lite

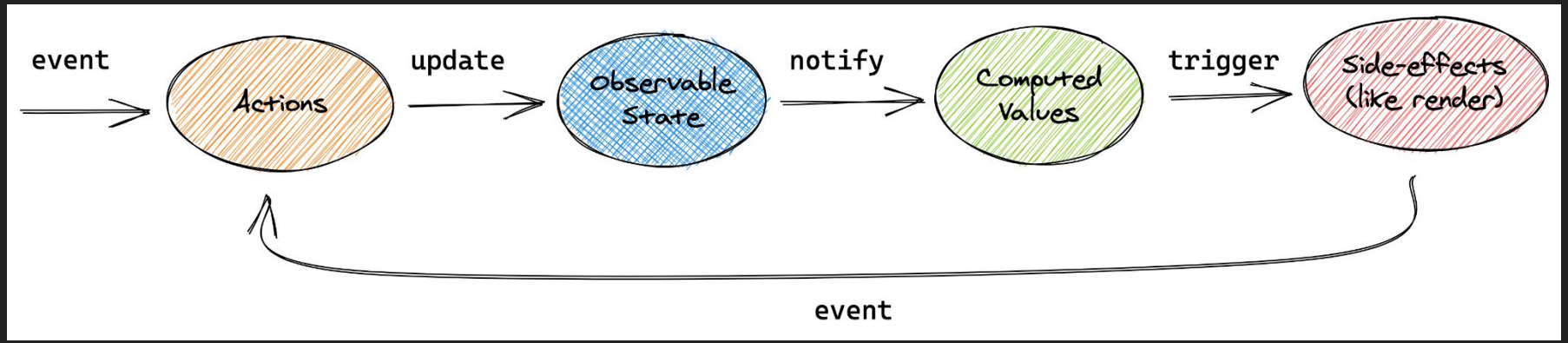
Основные принципы MobX

Наблюдаемые состояния (Observable State): Состояние, которое может изменяться и за которым можно наблюдать. MobX позволяет создавать наблюдаемые состояния с помощью makeAutoObservable или других функций.

Действия (Actions): Методы, которые изменяют состояние. В MobX действия используются для инкапсуляции логики изменения состояния и обеспечивают, чтобы изменения состояния происходили централизованно.

Выводы (Computed Values): Производные значения, которые зависят от наблюдаемых состояний. Они вычисляются автоматически, когда изменяются их зависимости.

Наблюдатели (Observers): Компоненты или функции, которые реагируют на изменения в наблюдаемых состояниях. В React компоненты оборачиваются в observer для автоматического обновления при изменении состояния.



Пример MobX

```
import { appMobxStore } from "@/store";
import { toJS } from "mobx";

export function useAppMobxStore() {
  return [
    toJS(appMobxStore.todoList),
    appMobxStore.addTodo,
    appMobxStore.toggleTodoStatus,
    appMobxStore.deleteTodo,
    appMobxStore.clearTodoList,
  ];
}
```

```
import { useAppMobxStore } from "@/use";
import { AppTodo } from "@/components";
import { observer } from "mobx-react-lite";

export const AppTodoWithMobx = observer(() => {
  const [
    todoList,
    addNewTodoHandler,
    changeTodoStatusHandler,
    deleteTodoHandler,
    clearTodoListHandler,
  ] = useAppMobxStore();
  return (
    <AppTodo
      title="Mobx TODO"
      todoList={todoList}
      addNewTodoHandler={addNewTodoHandler}
      changeTodoHandler={changeTodoStatusHandler}
      deleteTodoHandler={deleteTodoHandler}
      clearTodoListHandler={clearTodoListHandler}
    />
  );
});
```

```
import { makeAutoObservable } from "mobx";

class AppMobxStore {
  todoList = [];

  constructor() {
    makeAutoObservable(this);
  }

  addTodo = (title) => {
    this.todoList.push({
      id: (this.todoList[this.todoList.length - 1]?.id || 0) + 1,
      title,
      status: false,
    });
  };

  toggleTodoStatus = (id) => {
    const todo = this.todoList.find((todo) => todo.id === id);
    if (todo) {
      todo.status = !todo.status;
    }
  };

  deleteTodo = (id) => {
    this.todoList = this.todoList.filter((todo) => todo.id !== id);
  };

  clearTodoList = () => {
    this.todoList = [];
  };
}

export const appMobxStore = new AppMobxStore();
```

RxJS

RxJS (Reactive Extensions for JavaScript) — это библиотека для реактивного программирования с использованием потоков данных и операторов. Она позволяет обрабатывать асинхронные данные, события и потоки данных в JavaScript с помощью концепций из функционального программирования и реактивного программирования.

Установка
npm install rxjs

Пример счетчика на RxJS

```
// counter.js
import { BehaviorSubject } from 'rxjs';
import { scan } from 'rxjs/operators';

// Создаем начальное состояние
const initialCount = 0;

// Создаем Subject для счетчика
const count$ = new BehaviorSubject(initialCount);

// Создаем функции для изменения значения
const increment = () => count$.next(count$.getValue() + 1);
const decrement = () => count$.next(count$.getValue() - 1);

export { count$, increment, decrement };
```

```
// App.js
import React, { useEffect, useState } from 'react';
import { count$, increment, decrement } from './counter';
import { subscribe } from 'rxjs';

function App() {
  const [count, setCount] = useState(count$.getValue());

  useEffect(() => {
    const subscription = count$.subscribe(setCount);
    return () => subscription.unsubscribe();
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default App;
```

Effector js

Effector — это библиотека для управления состоянием в JavaScript и TypeScript приложениях. Она предоставляет инструменты для создания, управления и использования состояния и эффектов (побочных эффектов) в реактивном программировании. Effector ориентирован на простоту использования, производительность и удобство интеграции с различными фреймворками и библиотеками.

Установка
npm install effector

Пример счетчика на Effector

```
// counter.js
import { createEvent, createStore } from 'effector';

// Создаем события
const increment = createEvent();
const decrement = createEvent();

// Создаем хранилище с начальным значением 0
const $counter = createStore(0)
  .on(increment, (state) => state + 1)
  .on(decrement, (state) => state - 1);

export { $counter, increment, decrement };
```

```
// App.js
import React from 'react';
import { useStore } from 'effector-react';
import { $counter, increment, decrement } from './counter';

function App() {
  const count = useStore($counter);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => increment()}>Increment</button>
      <button onClick={() => decrement()}>Decrement</button>
    </div>
  );
}

export default App;
```

Машины состояний (MS) и конечный автомат (FSM)

Машины состояний и конечные автоматы состояний (FSM) — это концепции, используемые для управления поведением систем в зависимости от их состояния. Эти концепции полезны в программировании и системном дизайне, особенно для моделирования и управления сложными процессами.

Машина состояний — это модель, которая описывает поведение системы через переходы между различными состояниями. Каждое состояние представляет собой определенную конфигурацию или режим системы, а переходы между состояниями происходят в ответ на определенные события или условия.

Конечный автомат состояний — это разновидность машины состояний, которая имеет конечное количество состояний и переходов. Он представляет собой математическую модель

XState

XState — это библиотека для управления состоянием и логикой в JavaScript и TypeScript, основанная на концепции конечных автоматов (Finite State Machines, FSM) и машин состояний (Statecharts). Она предоставляет мощный способ моделирования и управления состоянием приложения с помощью визуальных и декларативных диаграмм состояний.

Установка
npm install xstate

Пример

Открытая Дверь:
Можно закрыть дверь, что переведет ее в состояние "закрыта".

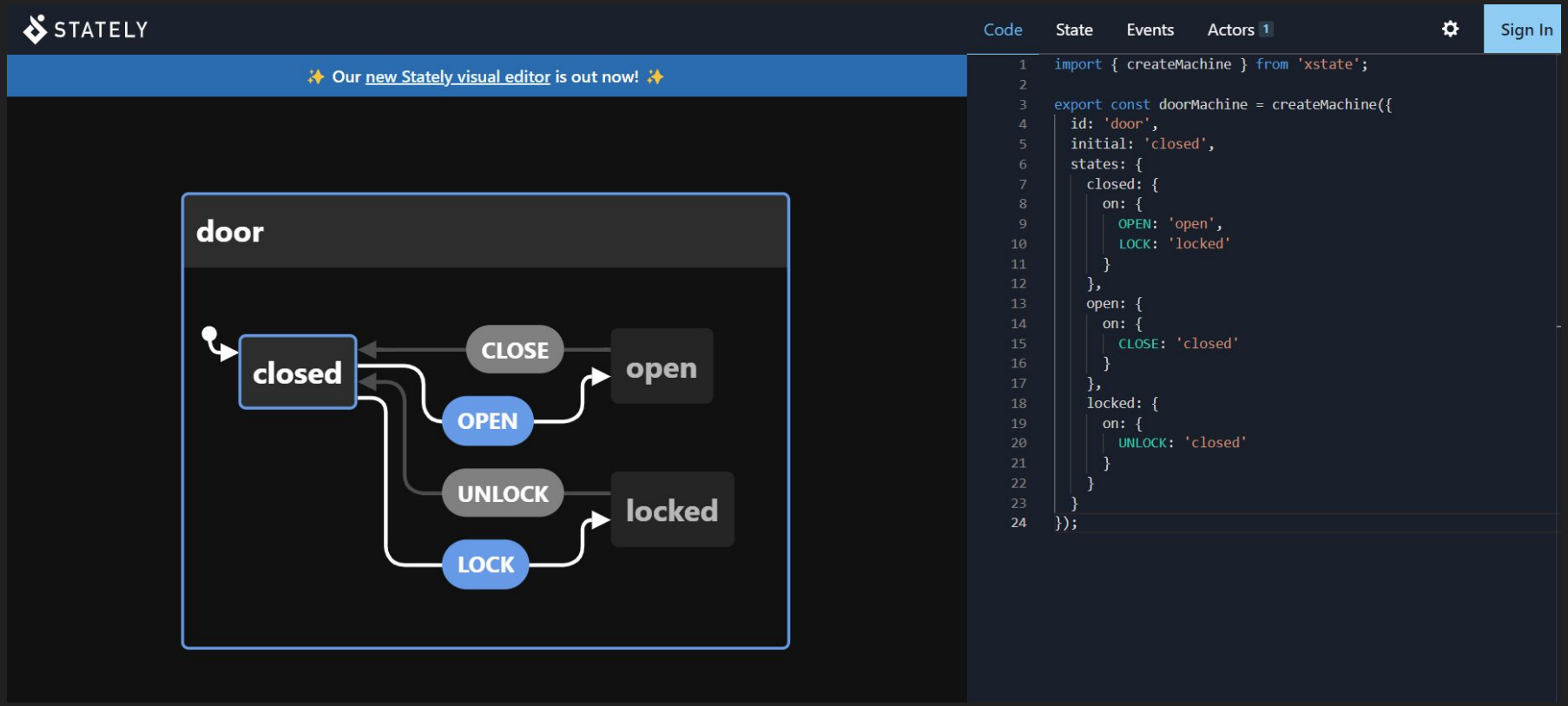
Закрытая Дверь:
Можно открыть дверь (перевод в состояние "открыта") или запереть ее (перевод в состояние "заперта").

Запертая Дверь:
Можно только отпереть дверь, что переведет ее в состояние "закрыта".

```
import { createMachine, interpret } from 'xstate';

// Создание машины состояний для двери
const doorMachine = createMachine({
  id: 'door',
  initial: 'closed',
  states: {
    open: {
      on: {
        CLOSE: 'closed'
      }
    },
    closed: {
      on: {
        OPEN: 'open',
        LOCK: 'locked'
      }
    },
    locked: {
      on: {
        UNLOCK: 'closed'
      }
    }
  }
});

const doorService = interpret(doorMachine).start();

export { doorService };
```

```
// DoorComponent.js
import React from 'react';
import { useActor } from '@xstate/react';
import { doorService } from './doorMachine';

function DoorComponent() {
  const [state, send] = useActor(doorService);

  return (
    <div>
      <h1>Door is currently: {state.value}</h1>
      {state.matches('open') && (
        <button onClick={() => send('CLOSE')}>Close Door</button>
      )}
      {state.matches('closed') && (
        <>
          <button onClick={() => send('OPEN')}>Open Door</button>
          <button onClick={() => send('LOCK')}>Lock Door</button>
        </>
      )}
      {state.matches('locked') && (
        <button onClick={() => send('UNLOCK')}>Unlock Door</button>
      )}
    </div>
  );
}

export default DoorComponent;
```

Визуализатор машин состояния



Ресурсы

Код с урока (git репозиторий) - ТЫК

Код с урока (vercel деплой) - ТЫК

пользовательские хуки - ТЫК

хук useContext - ТЫК

хук useReducer - ТЫК

хук useSyncExternalStore - ТЫК

Оф. документация Redux - ТЫК

Оф. документация Zustand - ТЫК

Оф. документация MobX - ТЫК

Оф. документация RxJS - ТЫК

Оф. документация Effector - ТЫК

Оф. документация XState - ТЫК

XState визуализатор - ТЫК