JS Level 2

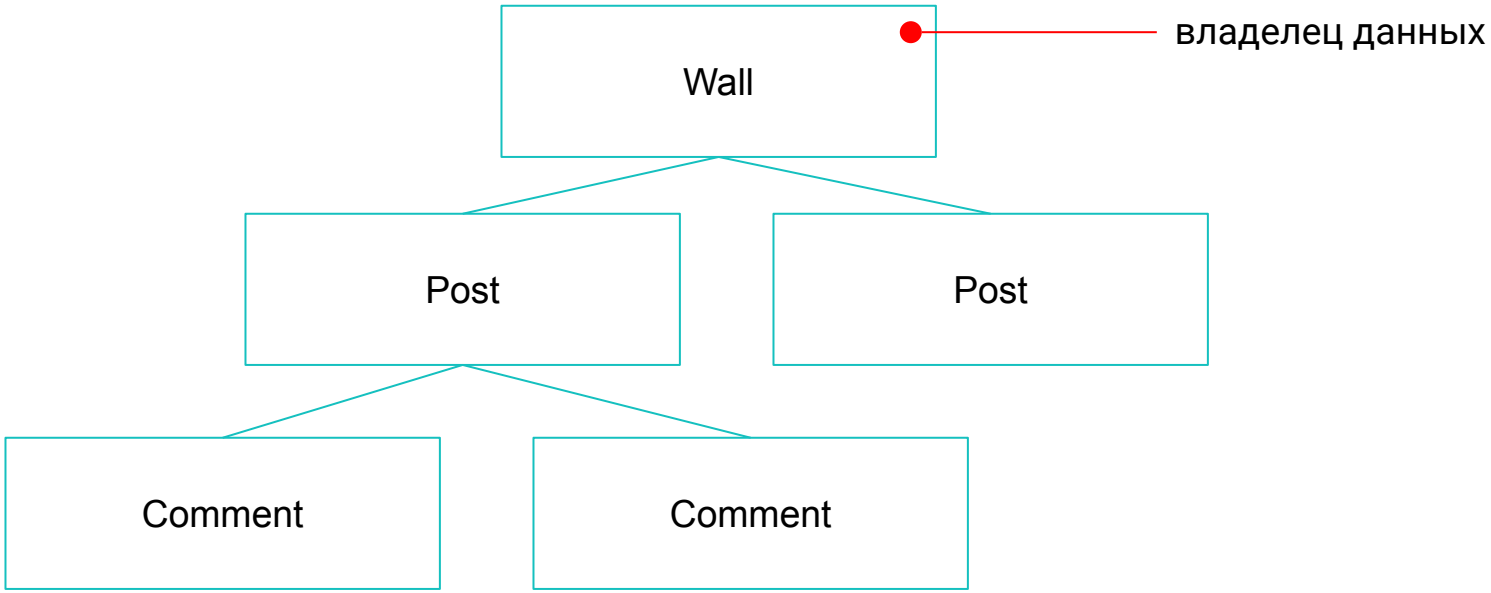
CONTEXT API

React

На прошлой лекции мы завершили разработку CRUD-приложения. Самое время поговорить о том, как можно более удобно и профессионально организовать работу (и само приложение).

Состояние

Хранить состояние в компонентах так, как делали мы – это вполне допустимо и работает. Но чем больше становится приложение, тем больше у нас становится компонентов, растёт само дерево компонентов и получается нечто вроде:



И если мы хотим произвести какое-то действие с Comment, то мы должны прокидывать callback'и от Wall через Post (согласитесь, очень неудобно).

Состояние

Т.е. это выглядело бы как:

Wall.js: <Post onCommentLike={...} />

Post.js: <Comment onLike={...} />

Comment.js: вызов onLike

А если вложенность возрастёт? Тогда цепочка станет ещё больше.

Context API

Поэтому в React придумали Context API, которое позволяет делать то же самое, но "скрывает" от нас явную передачу подобных коллбеков.

Основная идея кроется в том, что мы выносим состояние (целиком всего приложения или его часть в один из вышестоящих компонентов, а все дочерние имеют доступ к нему – этому состоянию).

Давайте посмотрим на примере.

Context API

JS PostsContext.js ×

src > contexts > JS PostsContext.js > ...

```
1  import { createContext } from 'react';
2
3  const PostsContext = createContext();
4
5  export default PostsContext;
```

Функция createContext создаёт Context, который мы и будем использовать дальше.

Context API

Контекст представляет собой обычный JS объект, с некоторыми заранее определёнными полями:

```
▼ Object
    $$typeof: Symbol(react.context)
  ▶ Consumer: {$$typeof: Symbol(react.context), _context: {…}, _calculateChangedBits: null, …}
  ▶ Provider: {$$typeof: Symbol(react.provider), _context: {…}}
    _calculateChangedBits: null
    _currentRenderer: null
    _currentRenderer2: null
    _currentValue: undefined
    _currentValue2: undefined
    _threadCount: 0
  ▶ __proto__: Object
```

Context API

Сам по себе контекст не существует в вакууме: должен быть компонент, который предоставляет этот контекст дочерним компонентам, а дочерние компоненты смогут "потреблять" этот контекст, используя хуки.

Provider

После создания контекста React предоставляет нам специальный компонент, который используется для того, чтобы передавать контекст вниз по дереву компонентов. Называется он ИмяКонтекста.Provider, т.е. если у нас контекст назывался PostsContext.Provider.

Но сам провайдер данные не хранит, поэтому мы создадим отдельный компонент, который и будет в своём state хранить нужные нам данные.

Provider

PostsProvider.js

src > contexts > PostsProvider.js > ...

```
1   import React, { useState } from 'react'
2   import PostsContext from './PostsContext'
3
4   export default function PostsProvider(props) {
5     const [posts, setPosts] = useState([]);
6     const [edited, setEdited] = useState();
7
8     const value = {posts, setPosts, edited, setEdited};
9
10    return (
11      <PostsContext.Provider value={value}>
12        {props.children}
13      </PostsContext.Provider>
14    )
15  }
```

Первые строки отвечают за использование state, а дальше становится интереснее.

Provider

Во-первых, Provider требует, чтобы мы передавали значение value, которое и смогут использовать дочерние компоненты.

```
return (
  <PostsContext.Provider value={value}>
    {props.children}
  </PostsContext.Provider>
)
```

Во-вторых, здесь используется специальная запись props.children. Что она значит? Она значит, что если мы будем писать вот так в JSX:

<PostsProvider>

<Wall />

</PostsProvider>

То именно Wall попадёт в props.children и таким образом мы сможем его отрисовать.

App.js

App.js

src > App.js > ...

```
1   import React from 'react';
2   import Wall from './components/Wall/Wall';
3   import PostsProvider from './contexts/PostsProvider';
4
5   function App() {
6     return (
7       <div className="App">
8         <PostsProvider>
9           <Wall />
10        </PostsProvider>
11      </div>
12    );
13  }
14
15  export default App;
```

В App.js мы "заворачиваем" Wall в наш провайдер.

Дерево компонентов

Благодаря props.children у нас получается вот такое дерево компонентов (PostsContext.Provider заменился на Context.Provider):

Wall

```
function Wall(props) {
  const {posts, setPosts, edited, setEdited} = useContext(PostsContext);
```

Для использования контекста нам понадобится хук useContext с указанием того контекста, который мы хотим использовать (value которого будет предоставлять один из компонентов выше по дереву).

Остальное оставляем так же, проверяем, что работает.

Wall

Пока особых бонусов не видно. Но что, если разгрузить компонент Wall и вынести все вот эти функции в провайдер нашего контекста?

Давайте попробуем, заодно их немного переименуем.

Provider

```
export default function PostsProvider(props) {
  const [posts, setPosts] = useState([]);
  const [edited, setEdited] = useState();

  const like = (id) => {…
  };

  const remove = (id) => {…
  };

  const toggleVisibility = (id) => {…
  };

  const edit = (id) => {…
  };

  const save = (post) => {…
  };

  const cancel = () => {…
  };

  const value = {
    posts,
    like,
    remove,
    toggleVisibility,
    edit,
    save,
    cancel,
    edited,
    setEdited,
  };

  return (
    <PostsContext.Provider value={value}>
      {props.children}
    </PostsContext.Provider>
  )
}
```

Wall

```
function Wall(props) {
  const {posts, like, remove, toggleVisibility, edit, save, cancel, edited, setEdited} = useContext(PostsContext);

  const handlePostLike = (id) => {
    like(id);
  };

  const handlePostRemove = (id) => {
    remove(id);
  };

  const handleTogglePostVisibility = (id) => {
    toggleVisibility(id);
  };

  const handlePostEdit = (id) => {
    edit(id);
  };

  const handlePostSave = (post) => {
    save(post);
  };

  const handlePostCancel = () => {
    cancel();
  };
```

Wall

Но тут возникает вопрос: а зачем компоненту Wall эти функции? Они же вызываются в Post'е? Мы пока оставили несколько, чтобы не сломалось редактирование:

```
function Wall(props) {
  const {posts, save, cancel, edited} = useContext(PostsContext);

  const handlePostSave = (post) => {
    save(post);
  };

  const handlePostCancel = () => {
    cancel();
  };

  return (
    <>
      <PostForm edited={edited} onSave={handlePostSave} onCancel={handlePostCancel}/>
      <div>
        {posts.map(o => <Post key={o.id} post={o} />)}
      </div>
    </>
  );
}

export default Wall;
```

Post

```
function Post({post}) {
  const {author} = post;
  const {photo} = post;

  const {like, remove, toggleVisibility, edit} = useContext(PostsContext);

  const handleLike = () => {
    like(post.id);
  };

  const handleRemove = () => {
    remove(post.id);
  };

  const handleHide = () => {
    toggleVisibility(post.id);
  };

  const handleEdit = () => {
    edit(post.id);
  };

  const handleShow = () => {
    toggleVisibility(post.id);
  };
```

Post

Обратите внимание: Post по-прежнему получает props post, поскольку в контексте хранятся все посты, а не конкретный.

Таким образом, мы почти разгрузили компонент Wall: теперь его задача заниматься только отображением постов.

Осталось завершить эту работу и переделать компонент PostForm.

Provider

Для этого поменяем немного провайдер (установим в качестве редактируемого пустой пост):

```
const empty = {
  id: 0,
  author: {
    id: 1,
    avatar: 'https://alif-skills.pro/logo_js.svg',
    name: 'Alif Skills',
  },
  content: '',
  photo: null,
  hit: false,
  likes: 0,
  likedByMe: false,
  hidden: false,
  tags: null,
  created: 0,
};
export default function PostsProvider(props) {
  const [posts, setPosts] = useState([]);
  const [edited, setEdited] = useState(empty);
```

Provider

А также сами функции сохранения и отмены:

```
const save = (post) => {
  if (edited?.id === 0) {
    setPosts((prevState) => [{...post}, ...prevState]);
    setEdited(empty);
    return;
  }

  setPosts((prevState) => prevState.map((o) => {
    if (o.id !== post.id) {
      return o;
    }

    return {...post};
  }))
  setEdited(empty);
  return;
};

const cancel = () => {
  setEdited(empty);
};
```

PostForm

Тогда в самой форме:

```
export default function PostForm() {
  const {save, cancel, edited, setEdited} = useContext(PostsContext);
  const firstFocusEl = useRef(null);
```

И везде post придётся заменить на edited, либо написать в форме:

```
export default function PostForm() {
  const {save, cancel, edited: post, setEdited: setPost} = useContext(PostsContext);
  const firstFocusEl = useRef(null);
```

Что означает деструктуризацию с переименованием (мы создаём переменную post, в которую кладём значение поля edited из контекста).

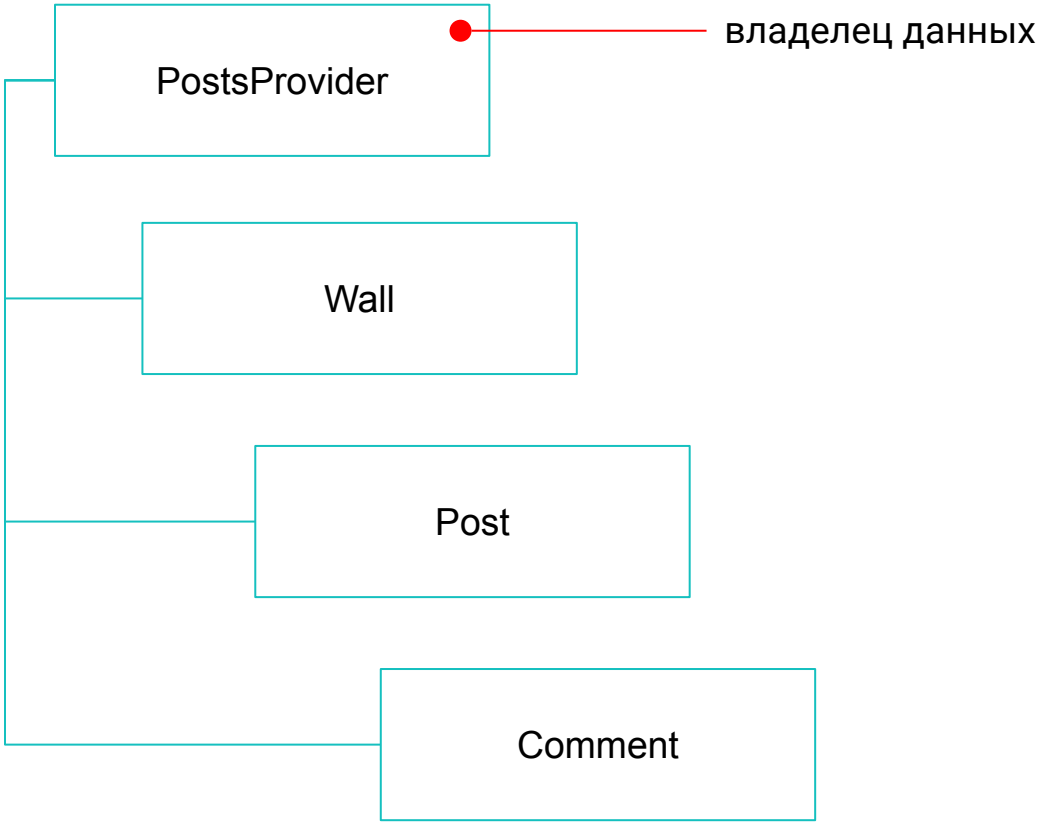
Wall

A Wall станет вообще простым:

```
function Wall() {
  const {posts} = useContext(PostsContext);

  return (
    <>
      <PostForm />
      <div>
        {posts.map(o => <Post key={o.id} post={o} />)}
      </div>
    </>
  );
}

export default Wall;
```

Context

Чего мы добились? Теперь вообще без разницы, на каком уровне располагаются компоненты относительно друг друга. Wall связан с PostForm только тем, что содержит его внутри себя. А с Post только тем, что передаёт туда post.



PostForm

Давайте пойдём немного дальше и наведём красоту в компоненте PostForm. Зачем он делает setPost? Ведь ему достаточно передавать в провайдер только информацию о том, какое конкретно поле изменилось:

```
const handleChange = (evt) => {
  const {name, value} = evt.target;
  if (name === 'tags') {
    const parsed = value.split(' ');
    setPost((prevState) => ({...prevState, [name]: parsed}));
    return;
  }
```

Давайте исправим это.

Provider

```
const submit = () => {…
};

const change = ({name, value}) => {…
};

const value = {
  posts,
  like,
  remove,
  toggleVisibility,
  edit,
  save,
  edited,
  cancel,
  submit,
  change,
};
```

Обратите внимание, мы убрали из value setPosts, setEdited, поскольку они не нужны дочерним компонентам.

PostForm

Теперь и PostForm стал "красивый":

```
export default function PostForm() {
  const {cancel, edited, submit, change} = useContext(PostsContext);
  const firstFocusEl = useRef(null);

  const handleSubmit = (evt) => {
    evt.preventDefault();
    submit();
    firstFocusEl.current.focus();
  };

  const handleReset = (evt) => {
    evt.preventDefault();
    cancel();
  };

  const handleChange = (evt) => {
    const {name, value} = evt.target;
    change({name, value});
  };
```

Компоненты

Обратите внимание, что теперь наши компоненты почти не содержат логики и с ними очень легко работать. Но вот PostProvider "разбух". А вот эта часть вообще ужасна:

```
const value = {
  posts,
  like,
  remove,
  toggleVisibility,
  edit,
  save,
  edited,
  cancel,
  submit,
  change,
};
```

Если внимательно посмотреть, то получается, что дочерние компоненты отправляют Provider'у информацию о том, что необходимо что-то сделать, вызывая соответствующие функции, которые приводят к созданию нового state.

Reducer

В React есть хук useReducer, который позволяет нам в "более удобном" (поначалу вам будет казаться, что совсем в неудобном) виде организовать изменение state.

Давайте попробуем его использовать.

Reducer

```
const initialState = {
  posts: [],
  edited: empty,
};

const reducer = (state, action) => {
  switch (action) {
    default:
      return state;
  }
};

export default function PostsProvider(props) {
  const [state, dispatch] = useReducer(reducer, initialState);
  console.log(state);
  const value = { state, dispatch };

  return (
    <PostsContext.Provider value={value}>
      {props.children}
    </PostsContext.Provider>
  )
}
```

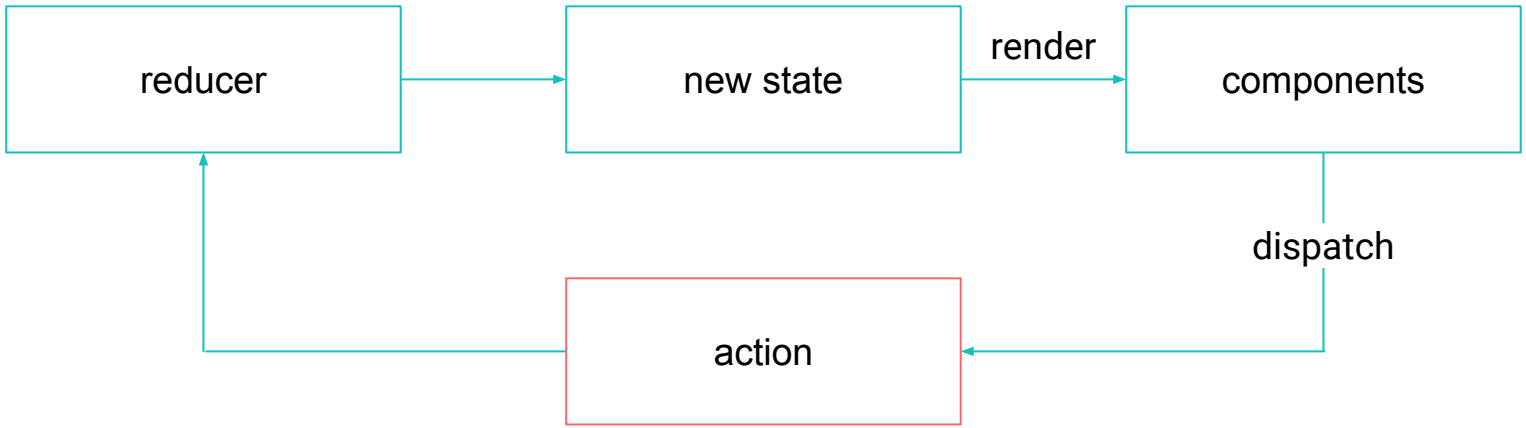
Итак, поехали: initialState – начальное состояние.

reducer – функция, которая получает предыдущее состояние и какой-то непонятный пока для нас action и создаёт новое состояние.

dispatch – специальная функция, предназначенная для вызова дочерними компонентами.

reducer и dispatch

Основная идея: дочерние компоненты, используя функцию dispatch, передают action (действие, которое необходимо выполнить). Это приводит к вызову reducer'а, который на основании action'а генерирует новое состояние. Возникновение нового состояния приводит к перерисовке компонентов.



Основная идея в возникновении однонаправленного потока изменения.

action'ы

Давайте попробуем отправить первый action. И договоримся мы следующим образом: будем отправлять обычные JS объекты вида:

```
{
   type: название action'a
   payload: данные
}
```

Примечание*: на самом деле, это общепринятое соглашение, но пока сделаем вид, что мы так сами придумали.

Wall

Компонент Wall особо не пострадал после нашей переделки:

```
function Wall() {
  const {state: {posts}} = useContext(PostsContext);

  return (
    <>
      <PostForm/>
      <div>
        {posts.map(o => <Post key={o.id} post={o}/>)}
      </div>
    </>
  );
}

export default Wall;
```

PostForm

PostForm теперь активно занимается dispatch'ингом action'ов вместо прямых вызовов:

```
export default function PostForm() {
  const {state: {edited}, dispatch} = useContext(PostsContext);
  const firstFocusEl = useRef(null);

  const handleSubmit = (evt) => {
    evt.preventDefault();
    dispatch({type: 'POST_EDIT_SUBMIT'});
    // submit();
    firstFocusEl.current.focus();
  };

  const handleReset = (evt) => {
    evt.preventDefault();
    dispatch({type: 'POST_EDIT_CANCEL'});
    // cancel();
  };

  const handleChange = (evt) => {
    const {name, value} = evt.target;
    dispatch({type: 'POST_EDIT_CHANGE', payload: {name, value}});
    // change({name, value});
  };
```

Post

```
function Post({post}) {
  const {author} = post;
  const {photo} = post;

  const {dispatch} = useContext(PostsContext);

  const handleLike = () => {
    dispatch({type: 'POST_LIKE', payload: {id: post.id}});
    // like(post.id);
  };

  const handleRemove = () => {
    dispatch({type: 'POST_REMOVE', payload: {id: post.id}});
    // remove(post.id);
  };

  const handleHide = () => {
    dispatch({type: 'POST_HIDE', payload: {id: post.id}});
    // toggleVisibility(post.id);
  };

  const handleEdit = () => {
    dispatch({type: 'POST_EDIT', payload: {id: post.id}});
    // edit(post.id);
  };

  const handleShow = () => {
    dispatch({type: 'POST_SHOW', payload: {id: post.id}});
    // toggleVisibility(post.id);
  };
```

Post изменится аналогично:

reducer

Ну и теперь самое интересное - это reducer:

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'POST_EDIT_SUBMIT':
      return state;
    case 'POST_EDIT_CANCEL':
      return state;
    case 'POST_EDIT_CHANGE':
      return state;
    case 'POST_LIKE':
      return state;
    case 'POST_REMOVE':
      return state;
    case 'POST_HIDE':
      return state;
    case 'POST_EDIT':
      return state;
    case 'POST_SHOW':
      return state;
    default:
      return state;
  }
};
```

Для начала напишем заглушку, которая всегда возвращает state ("старое" состояние).

Как работает switch: по типу action'а выбирает одну из веток case и поскольку там у нас стоит return, то reducer возвращает состояние.

Возврат "старого" состояния "говорит" о том, что ничего не изменилось и перерисовывать ничего не нужно.

change и submit

Мы реализуем самые "тяжёлые": change и submit:

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'POST_EDIT_SUBMIT':
      return reduceSubmit(state, action);
    case 'POST_EDIT_CANCEL':
      return state;
    case 'POST_EDIT_CHANGE':
      return reduceChange(state, action);
    case 'POST_LIKE':
      return state;
    case 'POST_REMOVE':
      return state;
    case 'POST_HIDE':
      return state;
    case 'POST_EDIT':
      return state;
    case 'POST_SHOW':
      return state;
    default:
      return state;
  }
};
```

change

```
const reduceChange = (state, action) => {
  const {edited} = state;
  const {payload: {name, value}} = action;
  if (name === 'tags') {
    const parsed = value.split(' ');
    return {
      ...state,
      edited: {...edited, [name]: parsed},
    }
  }

  if (name === 'photo' || name === 'alt') {
    const prop = name === 'photo' ? 'url' : name;
    return {
      ...state,
      edited: {...edited, photo: {...edited.photo, [prop]: value}},
    }
  }

  return {
    ...state,
    edited: {...edited, [name]: value},
  }
};
```

всегда копируем весь state и заменяем только нужные поля

submit

```
const reduceSubmit = (state, action) => {
  const {edited, posts} = state;
  const parsed = edited.tags?.map(o => o.replace('#', '')).filter(o => o.trim() !== '') || [];
  const tags = parsed.length !== 0 ? parsed : null;
  const post = {
    ...edited,
    id: edited.id || Date.now(),
    created: edited.created || Date.now(),
    tags,
    photo: edited.photo?.url ? {alt: '', ...edited.photo} : null
  };

  if (edited?.id === 0) {
    return {
      ...state,
      posts: [{...post}, ...posts],
      edited: empty,
    }
  }

  return {
    ...state,
    posts: posts.map((o) => {
      if (o.id !== post.id) {
        return o;
      }

      return {...post};
    }),
    edited: empty,
  }
};
```

Рефакторинг

Наше решение работает, но у него есть ряд минусов:

1. Типы action'ов мы пишем строками, очень легко ошибиться
2. payload'ы мы пишем каждый раз руками - тоже можем ошибиться
3. В Provider'е стало слишком много всего

Давайте попробуем это улучшить: создадим каталог actions, в котором опишем наши action'ы и удобные функции для их создания, и каталог reducers, в который поместим наш reducer:

Actions

JS index.js

src > store > actions > JS index.js > ...

```
export const POST_EDIT_SUBMIT = 'POST_EDIT_SUBMIT';
export const POST_EDIT_CANCEL = 'POST_EDIT_CANCEL';
export const POST_EDIT_CHANGE = 'POST_EDIT_CHANGE';
export const POST_LIKE = 'POST_LIKE';
export const POST_REMOVE = 'POST_REMOVE';
export const POST_HIDE = 'POST_HIDE';
export const POST_EDIT = 'POST_EDIT';
export const POST_SHOW = 'POST_SHOW';

export const editSubmit = () => {
  return {
    type: POST_EDIT_SUBMIT,
    payload: {},
  };
};

export const editCancel = () => {
  return {
    type: POST_EDIT_CANCEL,
    payload: {},
  };
};

export const editChange = (name, value) => {
  return {
    type: POST_EDIT_CHANGE,
    payload: {name, value},
  };
};

export const like = (id) => {
  return {
    type: POST_LIKE,
    payload: {id},
  };
};
```

Остальные (remove, hide, edit, show) пишутся по аналогии с like.

Пока все action'ы в одном файле, как только их станет много (в следующих лекциях) мы разделим их на файлы.

Reducers

Пока у нас один reducer на приложение, мы можем спокойно хранить его в файле index.js и называть reducer.

Как только их (reducer'ов) станет несколько, придётся переименовывать.

index.js

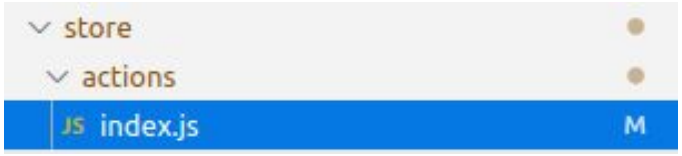
src > store > reducers > index.js > ...

```
import {
  POST_EDIT,
  POST_EDIT_CANCEL,
  POST_EDIT_CHANGE,
  POST_EDIT_SUBMIT,
  POST_HIDE,
  POST_LIKE,
  POST_REMOVE, POST_SHOW
} from '../actions';

const empty = {…
};

export const initialState = {…
};

export const reducer = (state, action) => {
  switch (action.type) {
    case POST_EDIT_SUBMIT:
      return reduceSubmit(state, action);
    case POST_EDIT_CANCEL:
      return state;
    case POST_EDIT_CHANGE:
      return reduceChange(state, action);
    case POST_LIKE:…
    case POST_REMOVE:…
    case POST_HIDE:…
    case POST_EDIT:…
    case POST_SHOW:…
    default:…
  }
};

const reduceSubmit = (state, action) => {…
};

const reduceChange = (state, action) => {…
};
```

index.js

Обратите внимание: index.js – это специальное имя файла, которое необязательно указывать при импорте. Т.е. если у вас в index.js экспортируются какие-то имена, то достаточно в импорте указать каталог, в котором лежит index.js:

```
1  import {
2    POST_EDIT,
3    POST_EDIT_CANCEL,
4    POST_EDIT_CHANGE,
5    POST_EDIT_SUBMIT,
6    POST_HIDE,
7    POST_LIKE,
8    POST_REMOVE, POST_SHOW
9  } from '../actions';
```



Provider

Теперь в Provider всё стало ещё проще:

PostsProvider.js

src > contexts > PostsProvider.js > ...

```
1  import React, { useReducer } from 'react'
2  import PostsContext from './PostsContext'
3  import { reducer, initialState } from '../store/reducers';
4
5  export default function PostsProvider(props) {
6    const [state, dispatch] = useReducer(reducer, initialState);
7    const value = {state, dispatch};
8
9    return (
10     <PostsContext.Provider value={value}>
11       {props.children}
12     </PostsContext.Provider>
13   )
14 }
```

PostForm

В PostForm и Post мы пользуемся уже функциями для создания action'ов (так называемыми action creator'ами):

```
export default function PostForm() {
  const {state: {edited}, dispatch} = useContext(PostsContext);
  const firstFocusEl = useRef(null);

  const handleSubmit = (ev) => {
    ev.preventDefault();
    dispatch(editSubmit());
    // dispatch({type: 'POST_EDIT_SUBMIT'});
    firstFocusEl.current.focus();
  };

  const handleReset = (ev) => {
    ev.preventDefault();
    dispatch(editCancel());
    // dispatch({type: 'POST_EDIT_CANCEL'});
  };

  const handleChange = (ev) => {
    const {name, value} = ev.target;
    dispatch(editChange(name, value));
    // dispatch({type: 'POST_EDIT_CHANGE', payload: {name, value}});
  };
```

Post.js

src > components > Post > Post.js > Post

```
1   import React, { useContext } from 'react';
2   import './Post.css';
3   import Tags from '../Tags/Tags';
4   import PostsContext from '../../contexts/PostsContext';
5   import { edit, hide, like, remove, show } from '../../store/actions';
6
7   function Post({post}) {
8     const {author} = post;
9     const {photo} = post;
10
11    const {dispatch} = useContext(PostsContext);
12
13    const handleLike = () => {
14      dispatch(like(post.id));
15    };
16
17    const handleRemove = () => {
18      dispatch(remove(post.id));
19    };
20
21    const handleHide = () => {
22      dispatch(hide(post.id));
23    };
24
25    const handleEdit = () => {
26      dispatch(edit(post.id));
27    };
28
29    const handleShow = () => {
30      dispatch(show(post.id));
31    };
```

useMemo

Ну и последняя вишенка на торте – хук useMemo, который позволяет хранить "мемоизированную" (т.е. "запомненную") версию объекта, пока не изменится одна из зависимостей (аналогично useEffect, только там запускалась функция, а тут перевычисляется значение):

```
export default function PostsProvider(props) {
  const [state, dispatch] = useReducer(reducer, initialState);
  const value = useMemo(() => ({state, dispatch}), [state]);

  return (
    <PostsContext.Provider value={value}>
      {props.children}
    </PostsContext.Provider>
  )
}
```

Зачем он нужен? Затем, что при каждом изменении value будут перерисовываться все дочерние компоненты, что может быть очень накладным.

Итоги

Вам может показаться, что многие действия "избыточны". В случае небольших приложений это действительно так: мы потратили достаточно много времени на настройку всего, хотя до этого "и так работало".

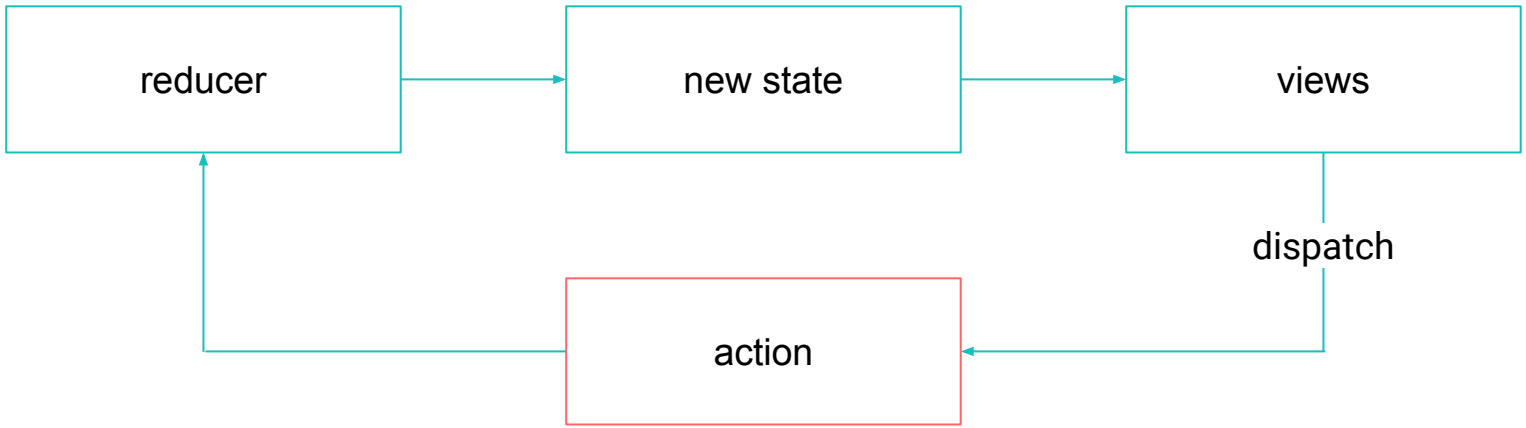
Но чем больше проект, тем важнее становится подобное разделение.

REDUX

Redux

Context API был долгое время "внутренним" API, не рекомендованным к использованию. Поэтому управление состоянием в React приложениях осуществлялось с помощью сторонней библиотеки, которая называется Redux.

Redux создавалась для управления состоянием в JS приложениях и именно она предложила (на самом деле предложила библиотека Flux, а Redux была её последователем, упростив некоторые концепции) подход:



Redux

Ключевое: Redux можно использовать и без React (как и React без Redux). Но достаточно часто они используются именно в этой связке.

Q: чем Redux лучше Context API?

A: на самом деле, Redux использует Context API, но предоставляет нам более удобные механизмы и обладает более развитой экосистемой (т.е. набором готовых инструментов, написанных под него).

Redux

Ключевые термины (которых у нас не было в Context API):

- View – отображаемый на экране компонент (в React – дерево компонентов), умеет отображать данные, но не умеет изменять их (вместо этого посылает через dispatcher action)
- Store – объект, хранящий состояние приложения

На всё приложение должен быть всего один Store (Context API позволяет же вам делать сколько угодно контекстов) и есть ограничения на Action'ы:

- это должны быть обычные объекты – т.е. не наследоваться ни от кого, и не создаваться с помощью new (мы специально их сделали такими, чтобы нам было легче перейти на Redux, Context API вас не ограничивает в том, какие Action'ы вы отправляете)
- у них должно быть поле type

Redux

Если Context API идёт "из коробки", то Redux нужно устанавливать:

npm install redux react-redux

Далее первым делом необходимо создать store:

index.js

src > store > index.js > ...

```
1  import { createStore } from 'redux';
2  import { reducer } from './reducers';
3
4  const store = createStore(reducer);
5  export default store;
```

store создаётся с помощью функции createStore и reducer'а.

Redux

Далее мы удаляем Context и Provider, поскольку они нам уже не нужны. А reducer видоизменяем на:

```
export const reducer = (state = initialState, action) => {
  switch (action.type) {
    case POST_EDIT_SUBMIT:
      return reduceSubmit(state, action);
    case POST_EDIT_CANCEL:
      return state;
    case POST_EDIT_CHANGE:
      return reduceChange(state, action);
    case POST_LIKE:…
    case POST_REMOVE:…
    case POST_HIDE:…
    case POST_EDIT:…
    case POST_SHOW:…
    default:…
  }
};
```

Таким образом будет инициализироваться начальный state нашего приложения.

Provider

Поскольку store у нас будет один на всё приложение, то мы заворачиваем компонент App в специальный компонент Provider, который и предоставляет библиотека React-Redux:

index.js

src > index.js

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6  import { Provider } from 'react-redux';
7  import store from './store';
8
9  ReactDOM.render(
10   <React.StrictMode>
11     <Provider store={store}>
12       <App />
13     </Provider>
14   </React.StrictMode>,
15   document.getElementById('root')
16 );
```

useSelector

React-Redux предоставляет нам хук useSelector, который позволяет выбрать нужный кусочек state:

Wall.js

src > components > Wall > Wall.js > ...

```
1  import React from 'react';
2  import Post from '../Post/Post';
3  import PostForm from '../PostForm/PostForm';
4  import { shallowEqual, useSelector } from 'react-redux';
5
6  function Wall() {
7    const posts = useSelector((state) => state.posts, shallowEqual);
8
9    return (
10     <>
11       <PostForm/>
12       <div>
13         {posts.map(o => <Post key={o.id} post={o}/>)}
14       </div>
15     </>
16   );
17 }
18
19 export default Wall;
```

shallowEqual позволяет более эффективно сравнивать объекты, не перерисовывая каждый раз компонент.

useDispatch

Кроме того, поскольку store у нас всего один, есть отдельный хук useDispatch, который предоставляет dispatch:

PostForm.js

src > components > PostForm > PostForm.js > PostForm

```
1  import React, { useRef} from 'react';
2  import { editCancel, editChange, editSubmit } from '../../store/actions';
3  import { shallowEqual, useDispatch, useSelector } from 'react-redux';
4
5  export default function PostForm() {
6    const dispatch = useDispatch();
7    const edited = useSelector((state) => state.edited, shallowEqual);
8    const firstFocusEl = useRef(null);
```

Post

Post.js

src > components > Post > Post.js > Post > handleShow

```
1  import React from 'react';
2  import './Post.css';
3  import Tags from '../Tags/Tags';
4  import { edit, hide, like, remove, show } from '../../store/actions';
5  import { useDispatch } from 'react-redux';
6
7  function Post({post}) {
8    const {author, photo} = post;
9
10   const dispatch = useDispatch();
11
```

App

App.js

src > App.js > ...

```
1  import React from 'react';
2  import Wall from './components/Wall/Wall';
3
4  function App() {
5    return (
6      <div className="App">
7        <Wall />
8      </div>
9    );
10 }
11
12 export default App;
```

reducers

Несмотря на то, что наше приложение работает, оно написано не совсем правильно с точки зрения Redux.

В Redux принято разделять state, предоставляя каждому reducer'у отдельный кусочек. Но об этом мы поговорим уже на следующей лекции, как и об инструментах для удобной отладки изменений в store.

ИТОГИ

Итоги

Сегодня мы рассмотрели вопросы управления состоянием в React приложениях с помощью двух инструментов:

1. Context API
2. Redux

Конечно же, мир управления состояниями в экосистеме React не ограничивается этими решениями - есть и MobX, и XState, и другие решения. Но Redux пока остаётся самым популярным.

ДОМАШНЕЕ ЗАДАНИЕ

Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе).

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

Если не успеете сдать в срок домашнее задания, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в Телеграм канале.

ДЗ: Context

Мы с вами реализовали обработку двух action'ов:

```
export const reducer = (state = initialState, action) => {
  switch (action.type) {
    case POST_EDIT_SUBMIT:
      return reduceSubmit(state, action);
    case POST_EDIT_CANCEL:
      return state;
    case POST_EDIT_CHANGE:
      return reduceChange(state, action);
    case POST_LIKE:…
    case POST_REMOVE:…
    case POST_HIDE:…
    case POST_EDIT:…
    case POST_SHOW:…
    default:…
  }
};
```

Вам нужно реализовать обработку остальных. Расположение файлов должно быть аналогично лекции. Бот будет сам подставлять провайдер и контекст (используя ваш reducer и initialState).

ДЗ: Redux

Мы с вами реализовали обработку двух action'ов:

```
export const reducer = (state = initialState, action) => {
  switch (action.type) {
    case POST_EDIT_SUBMIT:
      return reduceSubmit(state, action);
    case POST_EDIT_CANCEL:
      return state;
    case POST_EDIT_CHANGE:
      return reduceChange(state, action);
    case POST_LIKE:…
    case POST_REMOVE:…
    case POST_HIDE:…
    case POST_EDIT:…
    case POST_SHOW:…
    default:…
  }
};
```

Вам нужно реализовать обработку остальных. Расположение файлов должно быть аналогично лекции. Бот будет сам подставлять store, используя ваш reducer.

Спасибо за внимание

alif skills

2023г.