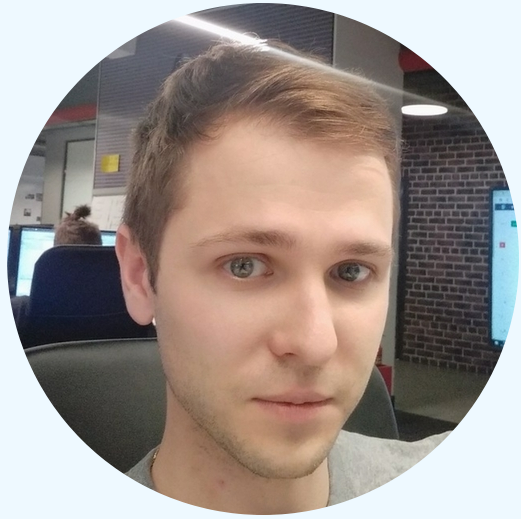
НЕТОЛОГИЯ

HOOKS, CONTEXT API

АЛЕКСАНДР РУСАКОВ



АЛЕКСАНДР РУСАКОВ

co-founder PCNP Studio

a_s_rusakov@mail.ru

@a_s_rusakov

ПЛАН ЗАНЯТИЯ

1. Хуки
2. useState
3. useEffect
4. useRef
5. Кастомные хуки
6. useContext & Context API

ХУКИ

ХУКИ

React Hooks (далее - хуки), появились в React в 2018 году, с целью решить несколько проблем:

- невозможность переиспользовать логику работы с состоянием в компонентах
- разрастание компонентов
- необходимость использовать классы

Поскольку с классами вы уже умеете работать, будем считать, что у нас существуют только две первых проблемы.

ДОП.МОТИВАЦИЯ

Изначально, компонент рассматривался как некоторая функция, которая на вход получает `props` и на выходе генерирует визуальное представление.

Для этого были очень удобны функциональные компоненты. Но при этом мог возникнуть момент, когда в компонент было необходимо добавить внутреннее состояние - и тогда компонент приходилось конвертировать в классы.

В итоге возникал вопрос - зачем вообще использовать функциональные компоненты, если их потом придётся конвертировать в классы?

ХУКИ

Говоря простым языком, хуки позволяют:

1. добавить в функциональные компоненты то, чего в них не хватало - управление состоянием и побочные эффекты
2. переиспользовать логику работы с состоянием, путём простого выноса её в функции

Давайте смотреть на примере: представим, что мы делаем интернет-портал, и в нём один из компонентов отвечает за отображение актуальных новостей.

АКТУАЛЬНЫЕ НОВОСТИ

Какие у вас есть идеи по поводу организации получения новостей с сервера? Не забудьте, мы хотим, чтобы новости были актуальными и пользователю не приходилось обновлять страницу для получения обновлений.

POLLING

Самое простое - обычный polling, когда мы с определённым интервалом будем опрашивать сервер (у нас не чат, поэтому long-polling, sse и websocket'ы в данной задаче будут излишними).

Т.е. нам при монтировании компонента (`componentDidMount`) нужно выполнить начальную загрузку и запланировать интервал (`setInterval`).

А перед размонтированием компонента (`componentWillUnmount`) нужно выполнить очистку интервала (`clearInterval`).

И вроде понятно, как это сделать в class-based компоненте, но как в функциональном?

USESTATE

useState

Начнём с того, что вспомним про тот хук, который мы уже с вами изучили - `useState`.

Возвращает переменную для получения значения состояния и функцию для установки значения состояния.

```
const [news, setNews] = useState([]);
// где-то после получения данных:
// вариант 1: установка нового state, не зависящего от предыдущего:
setState(data);
// вариант 2: установка нового state, зависящего от предыдущего:
setState(prevState => {... do something and return new state ...});
```

`useState`

В принципе, особо говорить больше про него нечего, нам нужен будет первый вариант (установка нового состояния, не зависящего от предыдущего).

Ну, и, возможно, стоит подумать над тем, как мы будем отображать пользователю сообщения об ошибке и процесс загрузки (а она может быть длительной).

USEEFFECT

useEffect

Хук, для выполнения побочных эффектов.

Полная сигнатура выглядит следующим образом:

```
useEffect(() => { // обычная функция (не асинхронная)
  // сам effect

  // можем вернуть функцию очистки
  return unsubscribe-func;
}, [deps]); // зависимости эффекта
```

`useEffect` VS LIFECYCLE

Неправильно говорить, что `useEffect` заменяет методы жизненного цикла, которые были в class-based компонентах.

На самом деле, он занимается выполнением побочных эффектов после рендера, для того, чтобы привести "окружение" React в нужное состояние (например, выполнив HTTP-запрос после рендера).

При этом стоит отметить, что выполнение побочных эффектов внутри основного тела функционального компонента (не callback'ах) - недопустимо.

REST API

Создадим небольшое REST API для нашего приложения с помощью Koa:

```
npm init
npm install forever koa koa-router koa2-cors faker
```

Пакет faker нам нужен будет для генерации фейковых данных.

REST API

```
const http = require('http');
const Koa = require('koa');
const Router = require('koa-router');
const cors = require('koa2-cors');
const faker = require('faker');

const app = new Koa();
app.use(cors());

let nextId = 1;
const latestCount = 3;
const news = [
  {id: nextId++, content: faker.lorem.sentence()},
  {id: nextId++, content: faker.lorem.sentence()},
  {id: nextId++, content: faker.lorem.sentence()},
];

setInterval(() => {
  news.push({id: nextId++, content: faker.lorem.sentence()});
}, 1 * 1000); // только для примера
```

REST API (ПРОДОЛЖЕНИЕ)

```
const router = new Router();
router.get('/news/latest', async (ctx, next) => {
  ctx.response.body = news.slice(news.length - latestCount, news.length).reverse();
});
app.use(router.routes()).use(router.allowedMethods());

http.createServer(app.callback()).listen(process.env.PORT || 7070);
```

REST API

.foreverignore:

```
node_modules
```

scripts в `package.json`:

```
"scripts": {
  "prestart": "npm install",
  "start": "forever server.js",
  "watch": "forever -w server.js"
},
```

LATESTNEWS

```
export default function LatestNews() {
  const [news, setNews] = useState([]);

  useEffect(() => {
    ...
  });

  console.log('return');
  return (
    <ul>
      {news.map(o => <li key={o.id}>#{o.id} {o.content}</li>)}
    </ul>
  )
}
```

ENV

Используем возможности `create-react-app`: создадим файл `.env`, в котором пропишем URL:

```
REACT_APP_NEWS_URL=http://localhost:7070/news/latest
```

Напоминаем, что префикс `REACT_APP_` обязателен.

Не забудьте перезапустить npm.

LATESTNEWS `useEffect`

```
useEffect(() => {
  const intervalId = setInterval(async () => {
    try {
      const response = await fetch(process.env.REACT_APP_NEWS_URL);
      if (!response.ok) {
        throw new Error(response.statusText);
      }
      const news = await response.json();
      setNews(news);
    } catch (e) {
      console.error(e);
    }
  }, 5 * 1000);
  return () => clearInterval(intervalId);
});
```

LATESTNEWS `useEffect`

И вроде всё хорошо, но ровно до тех пор, пока мы не посмотрим внимательнее:

```
useEffect(() => {
  console.log('set intervalId');
  const intervalId = setInterval(async () => {
    ...
  }, 5 * 1000);
  console.log(`intervalId: ${intervalId}`);
  return () => {
    console.log(`clear intervalId ${intervalId}`);
    clearInterval(intervalId);
  }
});
```

`useEffect`

Вспоминаем: эффект отрабатывает после каждого рендера. А рендер происходит каждый раз, как меняется `state` или `props`.

Поскольку мы в `useEffect` после завершения fetch вызываем `setState`, то состояние будет изменено и будет вызван новый рендер. А это не то, что бы нам хотелось.

И вторая важная деталь: очистка для предыдущего эффекта срабатывает после следующего рендера, перед запуском следующего эффекта.

DEPS

Это внешние параметры (например, `state`, `props`, включая функции), от изменения которых должен зависеть запуск нашего эффекта.

По умолчанию, этот параметр равен `undefined` - т.е. наш эффект будет перезапускаться при каждом рендере.

Это не совсем то, что нам нужно, т.к. мы хотим, чтобы хук не перезапускался при повторных рендерах.

DEPS

Давайте посмотрим, от чего на самом деле, зависит наш эффект: из внешних (содержащихся в самом компоненте) мы используем только `setNews`, и вроде как должны указать его в зависимостях.

Но документация на `useState` говорит, что: React гарантирует, что идентичность функции setState стабильна и не изменяется при повторных рендерах. Поэтому её можно безопасно не включать в списки зависимостей хуков useEffect и useCallback.

Поэтому получается, что наш эффект ни от чего не зависит и в параметре `deps` можно указать пустой массив.

ВАЖНО

Несмотря на то, что во многих источниках вы встретите информацию о том, что `desp = []` эквивалент `componentDidMount` - это неверно.

Указание `deps = []` позволяет добиться отсутствия перезапуска эффекта, т.к. не меняются зависимости.

ВАЖНО

Наше приложение страдает четырями проблемами:

1. Первый запрос делается только через 5 секунд
2. Нет индикатора загрузки
3. Нет обработки ошибок
4. `setInterval` не ждёт завершения предыдущего запроса

setInterval

Если первый запрос по каким-то причинам будет идти 5 секунд, а второй - меньше одной, то мы сначала увидим результаты из второго запроса, а потом из первого, что не хорошо.

Проверим. На сервере:

```
let isEven = true;
router.get('/news/latest', async (ctx, next) => {
  return new Promise(resolve => {
    const response = news.slice(news.length - latestCount, news.length)
      .reverse()
    setTimeout(() => {
      ctx.response.body = response;
      resolve();
    }, isEven ? 10 * 1000 : 0);
    isEven = !isEven;
  });
});
```

USEREF

`useRef`

Мы с вами уже использовали хук `useRef`, для того, чтобы ссылаться на DOM-элементы.

Но на самом деле, это более мощный инструмент. Ключевая его суть состоит в том, что через `useRef` мы всегда можем получить текущее значение (а не то, которое было запомнено в определённом рендере).

`useRef`

Что это значит? На самом деле наши компоненты - это функции, и вызов функции и хуки, которые мы использовали приводят к тому, что при каждом новом рендере функции (большинство) создаются заново, при этом благодаря замыканиям, они запоминают своё окружение, т.е. `state`, `props` и т.д.

Так вот, `useRef` позволяет это обойти и получать не значение, запомненное в closure, а актуальное.

Кроме того, изменение значения, хранящегося в `useRef`, не приводит к перерендерингу.

useRef

```
const timestampRef = useRef();
useEffect(() => {
  const intervalId = setInterval(async () => {
    const timestamp = Date.now();
    timestampRef.current = timestamp;
    try {
      const response = await fetch(process.env.REACT_APP_NEWS_URL);
      if (!response.ok) {
        throw new Error(response.statusText);
      }
      const news = await response.json();
      if (timestampRef.current === timestamp) {
        setNews(news);
      }
    } catch (e) {
      console.error(e);
    }
  }, 5 * 1000);
  return () => clearInterval(intervalId);
}, []);
```

`useRef`

Стоит отметить, что использование `useRef` - не всегда самая лучшая идея при наличии достойных альтернатив.

Так, наш пример вполне можно попробовать переписать с использованием `AbortController` или `xhr`, обеспечив возможность отмены предыдущего незавершившегося запроса.

ПЕРВЫЙ ВЫЗОВ

Вынесем саму функцию получения данных

```
const timestampRef = useRef();
useEffect(() => {
  const fetchNews = async () => {
    const timestamp = Date.now();
    timestampRef.current = timestamp;
    try {
      const response = await fetch(process.env.REACT_APP_NEWS_URL);
      if (!response.ok) {
        throw new Error(response.statusText);
      }
      if (timestampRef.current === timestamp) {
        const data = await response.json();
        setNews(data);
      }
    } catch (e) {
      console.error(e);
    }
  };

  fetchNews();
  const intervalId = setInterval(fetchNews, 5 * 1000);
```

LOADING И ERROR

```
const [isLoading, setLoading] = useState(false);
const [hasError, setError] = useState(null);
const timestampRef = useRef();
useEffect(() => {
  const fetchNews = async () => {
    const timestamp = Date.now();
    timestampRef.current = timestamp;
    setLoading(true);
    try {
      ...
      setError(null);
    } catch (e) {
      setError(e);
    } finally {
      setLoading(false);
    }
  };

  fetchNews();
  const intervalId = setInterval(fetchNews, 5 * 1000);
  return () => clearInterval(intervalId);
}, []);
```

LOADING И ERROR

А дальше уже дела вкуса, как отображать ошибку и индикатор загрузки (неплохо бы установить анимацию, т.к. в противном случае текст будет "скакать"):

```
return (
  <Fragment>
    {isLoading && <p>Loading...</p>}
    <ul>
      {news.map(o => <li key={o.id}>#{o.id} {o.content}</li>)}
    </ul>
  </Fragment>
)
```

ПЕРЕИСПОЛЬЗОВАНИЕ

На самом деле, мы создали довольно шаблонный код, который хотелось бы переиспользовать в других компонентах, если им будет нужен такого рода поллинг.

Можем ли мы это сделать?

КАСТОМНЫЕ ХУКИ

CUSTOM HOOKS

Мы вполне можем вынести логику поллинга в свой собственный хук и затем переиспользовать его в разных компонентах.

Создадим каталог `hooks` и файл `usePolling.js`:

```
1  import {useState, useEffect, useRef} from 'react';
2
3  export default function usePolling(url, interval, initialData) {
4    const [data, setData] = useState(initialData);
5    const [isLoading, setLoading] = useState(false);
6    const [hasError, setError] = useState(null);
7    const timestampRef = useRef()
8
9    useEffect(() => { ... }, []);
10   return [{data, isLoading, hasError}];
11 }
```

Обратите внимание, у `useEffect` нет `deps`, значит ни при изменении url'а, ни при изменении интервала перезапускаться он не будет.

```
useEffect(() => {
  const fetchData = async () => {
    const timestamp = Date.now();
    timestampRef.current = timestamp;
    setLoading(true);
    try {
      const response = await fetch(url);
      if (!response.ok) {
        throw new Error(response.statusText);
      }
      if (timestampRef.current === timestamp) {
        const data = await response.json();
        setData(data);
      }
      setError(null);
    } catch (e) {
      setError(e);
    } finally {
      setLoading(false);
    }
  };
  fetchData();
  const intervalId = setInterval(fetchData, interval);
  return () => clearInterval(intervalId);
}, [])
```

ИСПОЛЬЗОВАНИЕ ХУКА

```
import React, { Fragment } from 'react'
import usePolling from '../../hooks/usePolling';

export default function LatestNews() {
  const [{data: news, isLoading, hasError}] = usePolling(
    process.env.REACT_APP_NEWS_URL,
    5 * 1000,
    []
  );

  return (
    <Fragment>
      {isLoading && <p>Loading...</p>}
      <ul>
        {news.map(o => <li key={o.id}>#{o.id} {o.content}</li>)}
      </ul>
    </Fragment>
  );
}
```

ИСПОЛЬЗОВАНИЕ ХУКА

Условно можно считать, что `useState` и другие хуки, которые мы используем в своём кастомном хуке, фактически "вставляются" в наш компонент, как будто они там изначально были, но при этом (что очень важно) не пересекаются по областям видимости.

Т.е. мы как и `useState` можем использовать наш хук `usePolling` несколько раз в одном и том же компоненте.

БИБЛИОТЕКИ ХУКОВ

Таким образом, через какое-то время вы соберёте собственную библиотеку хуков, доступную для переиспользования.

Можете обратить внимание на библиотеки, собранные сообществом, например: usehooks.com и hooks.guide.

ОСТАВШИЕСЯ ВОПРОСЫ

1. Реагирование на изменение URL
2. Реагирование на изменение Interval
3. Отмена

Начнём с URL и Interval:

```
useEffect(() => {
  ...
}, [url, interval]);
```

Этого будет достаточно, чтобы при изменении хотя бы одного из двух параметров эффект перезапускался (а старый - очищался).

ИСПОЛЬЗОВАНИЕ ХУКА

```
import React, { Fragment } from 'react'
import usePolling from '../../hooks/usePolling';

export default function LatestNews() {
  const [pollingInterval, setPollingInterval] = useState(5 * 1000);
  const [{data: news, isLoading, hasError}] = usePolling(
    process.env.REACT_APP_NEWS_URL,
    pollingInterval,
    []
  );

  return (<Fragment>
    {isLoading && <p>Loading...</p>}
    <ul>
      {news.map(o => <li key={o.id}>#{o.id} {o.content}</li>)}
    </ul>
    <button onClick={()=>setPollingInterval(prev => prev - 1000)}>Faster</button>
  </Fragment>);
}
```

ПОДВИСШИЕ ЗАПРОСЫ

Хотя это и работает, предыдущий эффект очищается и создаётся новый, это не решает проблемы с подвисшими запросами, когда мы после смены url или interval будем получать результаты от старых запросов.

Попробуем обойтись без `useRef` и ещё раз вспомнить ключевые моменты о closure: в них запоминается имя из окружения (именно имя, а не значение) и провернём следующий трюк:

```
useEffect(() => {
  let canceled = false;
  const fetchData = async () => {
    if (canceled) {
      return;
    }
    ...
    fetch here
    ...
      if (!canceled && timestampRef.current === timestamp) {
        const data = await response.json();
        setData(data);
      }
    ...
  };
  fetchData();
  const intervalId = setInterval(fetchData, interval);
  return () => { canceled = true; clearInterval(intervalId); };
}, [url, interval])
```

ОСТАВШИЕСЯ ВОПРОСЫ

На самом деле, наш хук обладает одним оставшимся недостатком - если время ответа будет превышать интервал наших запросов, то мы будем постоянно висеть в состоянии `Loading...`. Подумайте, что с этим можно сделать.

И кроме того, сам fetch-запрос и извлечение данных/генерацию ошибки можно вынести в отдельную фукнцию, не завязанную на `timestamp'ы` и `setData/Error/Loading`.

И ключевое, что стоит отметить - всё-таки polling лучше делать на `setTimeout`, `setInterval` мы взяли специально, чтобы рассмотреть возможности хуков и ключевые проблемы.

DEPS

Стоит отметить, что манипулируя `deps` - т.е. изменяя их, мы можем "эмулировать" программный вызов хука.

ПОДВЕДЁМ ПРОМЕЖУТОЧНЫЕ ИТОГИ

1. Хуки - эффективный способ переиспользования логики работы с состоянием
2. Хуки можно объявлять только на верхнем уровне компонента либо на верхнем уровне другого хука (но не внутри условий и т.д.)
3. Хуки не обычные функции - не работают вне `render`'а.
4. `useState` - хук для работы с состоянием
5. `useEffect` - хук для работы с побочными эффектами
6. `useRef` - хук для работы с текущими значениями, изменение которых не ведёт к перерендерингу

USECONTEXT И CONTEXT API

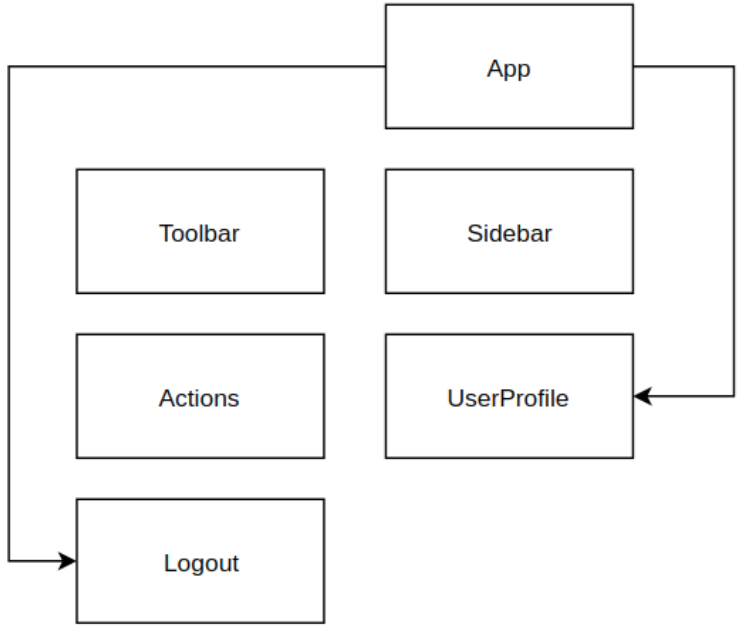
`useContext` и Context API

Итак, из основных хуков - у нас остался последний хук `useContext` и Context API.

При создании достаточно разветвлённого дерева компонентов, состояние, хранящееся в верхних компонентах, приходится пробрасывать достаточно глубоко в нижележащие компоненты:

К ЧЕМУ ЭТО ВЕДЁТ?

Это ведёт к усложнению компонентов и необходимости выполнять лишние действия: объявлять `props`, передавать их, следить за тем, что если вы встроили в дерево новый компонент в промежутке, то ничего не сломалось и т.д.



А хотелось бы как-то так, но при этом, чтобы App ничего не знал про Logout и UserProfile, а они - про App.

`useContext` И CONTEXT API

React нам предлагает концепцию Context'а, когда в одном из компонентов (на рисунке - App) мы объявляем данные, которые собираемся предоставлять другим компонентам. А компоненты, желающие использовать эти данные могут получить их без необходимости передавать `props` через всю цепочку компонентов.

При этом стоит заметить, что данные не обязательно должен предоставлять именно компонент, находящийся в вершине дерева - это может быть любой компонент.

Давайте посмотрим на примере.

АУТЕНТИФИКАЦИОННЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЯ

Рассмотрим эталонный пример - хранение аутентификационных данных пользователя и его профиля.

Здесь стоит немного рассказать об одной из существующих схем в SPA-приложениях, не вдаваясь в детали реализации.

В общем и целом процедура выглядит следующим образом: пользователь вводит свои учётные данные (чаще всего - логин и пароль), в ответ приложение получает токен доступа, который используется для отправки запросов от лица пользователя.

АУТЕНИФИКАЦИЯ И АВТОРИЗАЦИЯ

В задачу фронтенда входит:

1. Обменять учётные данные пользователя на токен
2. Хранить токен (чаще всего в localStorage/sessionStorage)
3. Подставлять его в заголовки запросов (чаще всего `Authorization`)

Напоминаю, что мы очень упрощённо рассматриваем схему, если вы хотите детальнее ознакомиться с этой темой - обязательно изучите Basic-аутентификацию, Session + Cookie, OAuth2 и JWT.

СЕРВЕР

Чтобы пример не был совсем синтетическим, мы напишем небольшой сервер, который позволит проводить аутентификацию

Нам понадобятся следующие зависимости:

```
npm install koa koa-passport koa-router koa-body koa2-cors
npm install uuid bcrypt forever passport-http-bearer
```

```
1  const http = require('http');
2  const Koa = require('koa');
3  const Router = require('koa-router');
4  const cors = require('koa2-cors');
5  const koaBody = require('koa-body');
6  const passport = require('koa-passport');
7  const { Strategy } = require('passport-http-bearer');
8  const uuid = require('uuid');
9  const bcrypt = require('bcrypt');
10
11 const app = new Koa();
12 app.use(cors());
13 app.use(koaBody())
```

```
// пользователей и токены просто храним в Map
const tokens = new Map(); // Redis и аналоги в реальной жизни
const users = new Map(); // БД в реальной жизни
const rounds = 10;

users.set('vasya', {
  id: uuid.v4(),
  login: 'vasya',
  // храним хэш от пароля
  password: bcrypt.hashSync('password', rounds),
  avatar: 'https://i.pravatar.cc/50'
});

// middleware для Аутентификации
passport.use(new Strategy((token, callback) => {
  const user = tokens.get(token);
  if (user === undefined) {
    // неуспешная аутентификация
    return callback(null, false);
  }

  // успешная аутентификация
  return callback(null, user);
}));
// не создаём сессии
const bearerAuth = passport.authenticate('bearer', { session: false });
```

```
// Роутинг
const router = new Router();
router.post('/auth', async (ctx, next) => {
  const {login, password} = ctx.request.body;
  const user = users.get(login);
  if (user === undefined) {
    ctx.response.status = 400;
    ctx.response.body = {message: 'user not found'};
    return;
  }
  const result = await bcrypt.compare(password, user.password);
  if (result === false) {
    ctx.response.status = 400;
    ctx.response.body = {message: 'invalid password'};
    return;
  }

  const token = uuid.v4();
  tokens.set(token, user);
  ctx.response.body = {token, profile: {
    id: user.id, login: user.login, avatar: user.avatar
  }};
});
// Закрываем все роуты с префиксом /private аутентификацией
router.use('/private**', bearerAuth);
router.get('/private', async (ctx, next) => {
  ctx.response.body = {message: 'ok'};
});
```

ЗАПУСК СЕРВЕРА

```
app.use(router.routes())
app.use(router.allowedMethods());

const port = process.env.PORT || 7070;
const server = http.createServer(app.callback());
server.listen(port);
```

Пара моментов:

1. Запрос на авторизацию должен быть POST с body `{"login": "vasya", "password": "password"}`
2. Аутентифицированные запросы должны содержать header: `Authorization: Bearer <token>`

Мы значительно упростили реализацию, отдавая сразу вместе с токеном и профиль пользователя (обычно это приходится делать двумя последовательными запросами).

CONTEXT

Вернёмся в наше React приложение и объявим контекст (файл `/src/contexts/AuthContext.js`):

```
1  import { createContext } from 'react';
2
3  const AuthContext = createContext({
4    token: null,
5    profile: null
6  });
7
8  export default AuthContext;
```

PROVIDER

Provider - это специальный компонент, который используется для того, чтобы передавать данные из контекста вниз по дереву.

Но данные же должны где-то храниться, а ещё мы их хотим менять, так, чтобы на это реагировали другие компоненты.

Значит, данные будут храниться в `state`. Можно, конечно, делать это и в компоненте App, но мы сделаем отдельный компонент `AuthProvider`:

AUTHPROVIDER

```
export default function AuthProvider(props) {
  const [token, setToken] = useState(null);
  const [profile, setProfile] = useState(null);

  const handleLogin = (login, password) => {
    // TODO:
  }

  const handleLogout = () => {
    // TODO:
  }

  return (
    <AuthContext.Provider value={{handleLogin, handleLogout, token, profile}}>
      {props.children}
    </AuthContext.Provider>
  )
}
```

Ключевое здесь - `AuthContext.Provider`, позволяющий "прокидывать" `value` вниз по дереву.

AUTHPROVIDER

Немного отвлечёмся и сразу напишем свой хук, который позволит сохранять и извлекать данные из `localStorage/sessionStorage`:

```
import { useState, useEffect } from 'react';

export default function useStorage(storage, key, jsonify = false) {
  const [value, setValue] = useState(
    jsonify ? JSON.parse(storage.getItem(key)) : storage.getItem(key)
  );

  useEffect(() => {
    if (value === null) {
      storage.removeItem(key);
      return;
    }
    storage.setItem(key, jsonify ? JSON.stringify(value) : value);
  }, [value, storage, key, jsonify]);


  return [value, setValue];
}
```

AUTHPROVIDER

```
export default function AuthProvider(props) {
  const [token, setToken] = useStorage(localStorage, 'token');
  const [profile, setProfile] = useStorage(localStorage, 'profile', true);
  // hardcoded login & pass for quick demo
  const handleLogin = async (login = 'vasya', password = 'password') => {
    try {
      const response = await fetch(process.env.REACT_APP_AUTH_URL, {
        method: 'POST',
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify({login, password}),
      });
      if (!response.ok) {
        throw new Error('Auth failed');
      }
      const {token, profile} = await response.json();
      setToken(token);
      setProfile(profile);
    } catch (e) {
      // Обработать ошибку
    }
  }
  ...
}
```

AUTHPROVIDER

```
export default function AuthProvider(props) {
  const [token, setToken] = useStorage(localStorage, 'token');
  const [profile, setProfile] = useStorage(localStorage, 'profile', true);

  const handleLogin = async (login, password) => {
    ...
  }

  const handleLogout = () => {
    setToken(null);
    setProfile(null);
  }

  return (
    <AuthContext.Provider value={{handleLogin, handleLogout, token, profile}}>
      {props.children}
    </AuthContext.Provider>
  )
}
```

ИСПОЛЬЗОВАНИЕ КОНТЕКСТА

Потреблять `value` мы можем несколькими способами:

1. `Consumer`
2. `useContext` в functional компоненте
3. `static contextType` в class-based компоненте

```
function App() {
  return (
    <AuthProvider>
    <Fragment>
      <ToolbarConsumer />
      <ToolbarFunctional />
      <ToolbarClassBased />
    </Fragment>
    </AuthProvider>
  );
}
```

CONSUMER

```
export default function ToolbarConsumer() {
  return (
    <AuthContext.Consumer>
    {value => (
      <Fragment>
      {value.profile && <Fragment>
        <img src={value.profile.avatar} />
        <button onClick={value.handleLogout}>Logout</button>
      </Fragment>
      }
      {!value.profile && <button onClick={value.handleLogin}>Login</button>}
      </Fragment>
    )}
    </AuthContext.Consumer>
  )
}
```

Не самый удобный вариант, но рабочий, позволяет заворачивать друг в друга несколько контекстов.

USECONTEXT

```
export default function ToolbarFunctional() {
  const {token, profile, handleLogin, handleLogout} = useContext(AuthContext);
  return (
    <Fragment>
    {profile && <Fragment>
      <img src={profile.avatar} />
      <button onClick={handleLogout}>Logout</button>
    </Fragment>
    }
    {!profile && <button onClick={handleLogin}>Login</button>}
    </Fragment>
  )
}
```

Самый удобный из вариантов, позволяет использовать несколько контекстов.

CONTEXTTYPE

```
export default class ToolbarClassBased extends Component {
  static contextType = AuthContext; // вариант 1

  render() {
    const { token, profile, handleLogin, handleLogout } = this.context;
    return (
      <Fragment>
        {profile && <Fragment>
          <img src={profile.avatar} />
          <button onClick={handleLogout}>Logout</button>
        </Fragment>
        }
        {!profile && <button onClick={handleLogin}>Login</button>}
      </Fragment>
    )
  }
}
ToolbarClassBased.contextType = AuthContext; // вариант 2
```

Позволяет использовать только один контекст.

ПОДВЕДЁМ ПРОМЕЖУТОЧНЫЕ ИТОГИ

1. Контекст - эффективный механизм проброса состояний при большой глубине дерева компонентов
2. Provider - обязательный элемент, обеспечивающий проброс `value`
3. `Consumer` - потребление в JSX
4. `useContext` - хук в functional компоненте
5. `static contextType` в class-based компоненте

ОБЩИЕ ИТОГИ

Хуки - достаточно мощный механизм, позволяющий переиспользовать общую логику работы с состоянием.

Мы с вами рассмотрели 4 важнейших: `useState`, `useEffect`, `useRef`, `useContext`.

Разработчики React говорят, что не отказываются от class-based компонентов, но рассматривают хуки в качестве основы будущего React.

Кроме того, сейчас всё чаще можно встретить в описаниях вакансий: React (хуки).

Начиная со следующей лекции вы вправе сами выбирать, какой стиль использовать: functional с хуками или class-based + lifecycle.

НЕТОЛОГИЯ

Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСАНДР РУСАКОВ


a_s_rusakov@mail.ru


@a_s_rusakov