Условный рендеринг

Переменные-элементы

Элементы React можно сохранять в переменных. Это может быть удобно, когда какое-то условие определяет, надо ли рендерить одну часть компонента или нет, а другая часть компонента остаётся неизменной.

Рассмотрим два компонента, представляющих кнопки Войти (Login) и Выйти (Logout).

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Войти
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Выйти
    </button>
  );
}
```

Создадим компонент с состоянием и назовём его LoginControl.

Он будет рендерить либо <LoginButton />, либо <LogoutButton /> в зависимости от текущего состояния.

А ещё он будет всегда рендерить <Greeting />

```
function UserGreeting(props) {
  return <h1>С возвращением!</h1>;
}

function GuestGreeting(props) {
  return <h1>Войдите, пожалуйста.</h1>;
}
```

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Попробуйте заменить на isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

Общий код

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }
```

```
  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;
    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginControl />);
```

bind() — это встроенный в React метод, который используется для передачи данных в качестве аргумента функции компонента на основе класса.

Синтаксис:

```
this.function.bind(this,[arg1...]);
```

Параметр: он принимает два параметра, первый параметр — ***это*** ключевое слово this, используемое для привязки, а второй параметр — это последовательность аргументов , которые передаются как параметр и являются необязательными.

```
<button onClick={this.handler.bind(this, 'GeeksForGeeks')}>
  Click Here
</button>
```

```
<button onClick={() => this.handler('GeeksForGeeks')}>
  Click Here
</button>
```

Хуки

Хуки — это функции, с помощью которых вы можете «подцепиться» к состоянию и методам жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают вам возможность использовать React без классов.

Жизненный цикл React

constructor(props): конструктор, в котором происходит начальная инициализация компонента

static getDerivedStateFromProps(props, state): вызывается непосредственно перед рендерингом компонента. Этот метод не имеет доступа к текущему объекту компонента (то есть обратиться к объкту компоненту через this) и должен возвращать объект для обновления объекта state или значение null, если нечего обновлять.

render(): рендеринг компонента

componentDidMount(): вызывается после рендеринга компонента. Здесь можно выполнять запросы к удаленным ресурсам

componentWillUnmount(): вызывается перед удалением компонента из DOM

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Привет, мир!</h1>
        <h2>Сейчас {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Хуки — это функции JavaScript, которые налагают два дополнительных правила:

- Хуки следует вызывать только **на верхнем уровне**. Не вызывайте хуки внутри циклов, условий или вложенных функций.
- Хуки следует вызывать только **из функциональных компонентов React**. Не вызывайте хуки из обычных JavaScript-функций. Есть только одно исключение, откуда можно вызывать хуки — это ваши пользовательские хуки.

С помощью хука эффекта useEffect можно выполнять побочные эффекты из функционального компонента. Он выполняет ту же роль, что и componentDidMount, componentDidUpdate и componentWillUnmount в React-классах, объединив их в единый API.
К примеру, этот компонент устанавливает заголовок документа после того, как React обновляет DOM:

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

function Example() {
  const [count, setCount] = useState(0);

  // По принципу componentDidMount и componentDidUpdate:
  useEffect(() => {
    // Обновляем заголовок документа, используя API браузера
    document.title = `Вы нажали ${count} раз`;
  });

  return (
    <div>
      <p>Вы нажали {count} раз</p>
      <button onClick={() => setCount(count + 1)}>
        Нажми на меня
      </button>
    </div>
  );
}
```

Когда вызывается useEffect, React получает указание запустить вашу функцию с «эффектом» после того, как он отправил изменения в DOM. Поскольку эффекты объявляются внутри компонента, у них есть доступ к его пропсам и состоянию. По умолчанию, React запускает эффекты после каждого рендера, *включая* первый рендер.

Побочными эффектами в React-компонентах могут быть: загрузка данных, оформление подписки и изменение DOM вручную

В качестве параметра в useEffect() передается функция. При вызове хука useEffect по сути определяется "эффект", который затем применяется в приложении. По умолчанию React применяет эффект после каждого рендеринга, в том числе при первом рендеринге приложения. Причем поскольку подобные эффекты определены внутри компонента, они имеют доступ к объекту props и к состоянию компонента.

```
function User() {
  const [name, setName] = React.useState("Tom");

  React.useEffect(() => {
    // Изменяем заголовок html-страницы
    document.title = `Привет ${name}`;
  });


  function changeName(event) {
    setName(event.target.value);
 }
```

```
  return (
    <div>
      <h3>Имя: {name}</h3>

      <div>
        <p>Имя: <input type="text" value={name} onChange={changeName} /></p>
      </div>
    </div>
  );
}

 ReactDOM.createRoot(
    document.getElementById("app")
)
.render(
    <User />
);
```

Здесь мы определяем эффект, который изменяет заголовок страницы. Причем в заголовок выводится значение переменной состояния - переменной name. То есть при загрузке страницы мы увидим в ее заголовке "Привет Tom".

Однако поскольку при вводе в текстовое поле мы изменяем значение в переменной name, и соответственно React будет выполнять перерендеринг приложения, то одновременно с этим будет изменяться и заголовок страни

По умолчанию эффект выполняется при каждом повторном рендеринге на веб-странице, однако можно указать, чтобы React не применял эффект, если определенные значения не изменились с момента последнего рендеринга. Для этого в useEffect в качестве необязательного параметра передается массив арг

```
function User() {
  const [name, setName] = React.useState("Tom");
  const [age, setAge] = React.useState(36);

  React.useEffect(() => {
    // Изменяем заголовок html-страницы
    document.title = `Привет ${name}`;
    console.log("useEffect");
  });

  const changeName = (event) => setName(event.target.value);
  const changeAge =(event) => setAge(event.target.value);
```

В данном случае в компоненте определены две переменных состояния: name и age. При этом эффект использует только переменную name. Однако даже если переменная name останется без изменений, но переменная age будет изменена, в этом случае эффект будет повторно срабатывать:

Чтобы указать, что эффект применяется только при изменении переменной name, передадим ее в качестве необязательного параметра в функцию useEffect:

```
React.useEffect(() => {
  // Изменяем заголовок html-страницы
  document.title = `Привет ${name}`;
  console.log("useEffect");
},
[name]); // эффект срабатывает только при изменении name
```

Чтобы эффект вызывался только один раз при самом первом рендеринге, то в качестве параметра передаются пустые квадратные скобки - []

```
React.useEffect(() => {
    // Изменяем заголовок html-страницы
    document.title = `Привет ${name}`;
    console.log("useEffect");
},
[]); // эффект срабатывает только один раз - при самом первом рендеринге
```

Очистка ресурсов

Нередко в приложении возникает необходимость подисывается на различные ресурсы, а после окончания работы и отписываться от них. В этом случае можно использовать специальную форму хука useEffect():

```
useEffect(() => {
  // код подписки на ресурс
  return () => {
    // код отписки от ресурса
  };
});
```

```
React.useEffect(() => {

    const unmountBtn = document.getElementById("unmountBtn");

     // подписываемся на событие onclick кнопки unmountBtn
     unmountBtn.addEventListener("click", unmount);
     console.log("EventListener added");

    return()=>{
         // отписываемся от события
        unmountBtn.removeEventListener("click", unmount);
        console.log("EventListener removed");
    }
  },
  []); // эффект срабатывает только один раз - при самом первом рендеринге
```

На странице определена кнопка с id=unmountBtn, на событие которой мы будем подписываться. В качестве действия, которое будет выполнять кнопка, в компоненте User определена функция unmount(), которая удаляет данный компонент с веб-страницы:

хук useEffect в данном случае будет срабатывать один раз - при самом первом рендеринге приложения - для этого в функцию в качестве необязательного параметра передается пустой массив. И соответственно функция очистки ресурсов, которая возвращаается оператором return будет выполняться один раз - при удалении компонента с веб-страницы.

Хук **useRef** позволяет сохранить некоторый объект, который можно изменять и который хранится в течение всей жизни компонента. В качестве параметра функция useRef() принимает начальное значение хранимого объекта. А возвращаемое значение - ссылка-объект, из свойства current которого можно получить хранимое значение.

```
const refUser = useRef("Tom");
console.log(refUser.current);    // Tom
```

Например, рассмотрим ситуацию, когда в начале компонент загружает состояние из LocalStorage, а после окончания работы с компонентом при завершении его жизненного цикла он сохраняет состояние обратно в LocalStorage.

```
function UserForm() {
  const [name, setName] = React.useState("Tom");
  const nameRef = React.useRef(name);

  React.useEffect(() => {

    nameRef.current = name;
  }, [name]);

  React.useEffect(() => {
     // извлекаем данные из localStorage
     const userName = localStorage.getItem("userName");
     // если в localStorage есть такой объект
     if(userName!==null) {
        setName(userName);
        console.log("Got!");
     }
```

```
    // сохраняем данные в localStorage
    return()=>{
        console.log(nameRef.current);
        localStorage.setItem("userName", nameRef.current);
        console.log("Saved!");
    }
},
[]); // эффект срабатывает только один раз - при самом первом рендеринге

const changeName = (event) => setName(event.target.value);
const unmount =() => root.unmount();
```

```
  return (
    <div>
      <h3>Имя: {name}</h3>

      <div>
        <p>Имя: <input type="text" value={name} onChange={changeName} /></p>
        <button onClick={unmount}>Unmount</button>
      </div>
    </div>
  );
}

root.render(
    <UserForm />
);
```

Определение маршрутов

В React имеется своя система маршрутизации, которая позволяет сопоставлять запросы к приложению с определенными компонентами. Ключевым звеном в работе маршрутизации является модуль react-router, который содержит основной функционал по работе с маршрутизацией. Но для работы в браузере, то нам также надо использовать модуль react-router-dom, а также history

В приведенном примере имеется 3 страницы, обрабатываемые роутером: главная (home), контакты (about) и страница с пользователями (users). При клике по <Link> (ссылке) роутер рендерит соответствующий <Route> (маршрут, путь).
По умолчанию <Link> рендерит <a> с настоящим href, так что люди, использующие клавиатуру для навигации или экранные считывающие устройства (screen readers), смогут без проблем пользоваться приложением.

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
} from 'react-router-dom';

export const App = () => (
  <Router>
    <header>
      <nav>
        <ul>
          <li>
            <Link to="/">Главная</Link>
          </li>
          <li>
            <Link to="/about">Контакты</Link>
          </li>
          <li>
            <Link to="/users">Пользователи</Link>
          </li>
        </ul>
      </nav>
    </header>
```

```
    <main>
      {/* <Switch> рендерит первый <Route>, совпавший с URL */}
      <Switch>
        <Route path="/about">
          <About />
        </Route>
        <Route path="/users">
          <Users />
        </Route>
        <Route path="/">
          <Home />
        </Route>
      </Switch>
    </main>
  </Router>
);

const Home = () => <h2>Главная</h2>;

const About = () => <h2>Контакты</h2>;

const Users = () => <h2>Пользователи</h2>;
```

Вложенный роутинг

Ниже приводится пример вложенного роутинга. Маршрут /topics загружает компонент Topics, который, в свою очередь, рендерит дальнейшие <Route> на основе значения :id.

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
  useRouteMatch,
  useParams,
} from 'react-router-dom';

export const App = () => (
  <Router>
    <header>
      <nav>
        <ul>
          <li>
            <Link to="/">Главная</Link>
          </li>
          <li>
            <Link to="/about">Контакты</Link>
          </li>
          <li>
            <Link to="/topics">Темы</Link>
          </li>
        </ul>
      </nav>
    </header>
    <main>
      <Switch>
        <Route path="/about">
          <About />
        </Route>
        <Route path="/topics">
          <Topics />
        </Route>
        <Route path="/">
          <Home />
        </Route>
      </Switch>
    </main>
  </Router>
);

const Home = () => <h2>Главная</h2>;

const About = () => <h2>Контакты</h2>;

function Topics() {
  const match = useRouteMatch();

  return (
    <>
      <h2>Темы</h2>
      <nav>
        <ul>
          <li>
            <Link to={`${match.url}/components`}>
              Компоненты
            </Link>
          </li>
          <li>
            <Link to={`${match.url}/props-vs-state`}>
              Пропы против состояния
            </Link>
          </li>
        </ul>
      </nav>

      {/* Страница Topics имеет собственный <Switch> с маршрутами,
          основанными на URL /topics. Вы можете думать о втором
          <Route> как о странице для остальных тем
          или как о странице, отображаемой,
          когда ни одна из тем не выбрана */}
      <div>
        <Switch>
          <Route path={`${match.path}/:topicId`}>
            <Topic />
          </Route>
          <Route path={match.path}>
            <h3>Пожалуйста, выберите тему.</h3>
          </Route>
        </Switch>
      </div>
    </>
  );
}

function Topic() {
  const { topicId } = useParams();
  return <h3>Идентификатор выбранной темы: {topicId}</h3>;
}
```

В React Router существует 3 категории компонентов:

- роутеры (routers), например, <BrowserRouter> или <HashRouter>
- маршруты (route matchers), например, <Route> или <Switch>
- и навигация (navigation), например, <Link>, <NavLink> или <Redirect>

Все компоненты, используемые в веб-приложении, должны быть импортированы из react-router-dom.

Роутеры

Любая маршрутизация начинается с роутера. Для веб-проектов react-router-dom предоставляет <BrowserRouter> и <HashRouter>. Основное отличие между ними состоит в способе хранения URL и взаимодействия с сервером.

- <BrowserRouter> использует обычные URL. В этом случае URL выглядят как обычно, но требуется определенная настройка сервера. В частности, сервер должен обслуживать все страницы, используемые на клиенте. Create React App поддерживает это из коробки в режиме разработки и содержит инструкции для правильной настройки сервера.
- <HashRouter> хранить текущую локацию в хэш-части URL (после символа "#"), поэтому URL выглядит примерно так: http://example.com/#/your/page. Поскольку хэш не отправляется серверу, его специальная настройка не требуется.

Для использования роутера необходимо обернуть в него компонент верхнего уровня:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';

const App = () => <h1>Привет, React Router</h1>;

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('root')
);
```

Маршруты

Существует 2 вида компонентов для поиска совпадений с URL: Switch и Route. При рендеринге <Switch> определяет <Route>, соответствующий текущему URL. При обнаружении такого маршрута, он рендерится, остальные маршруты игнорируются. Это означает, что нужно помещать более специфические маршруты перед менее специфическими.

Если совпадения не найдено, <Switch> ничего не рендерит (точнее, рендерит null).

```
import React from 'react';
import ReactDOM from 'react-dom';
import {
  BrowserRouter as Router,
  Switch,
  Route,
} from 'react-router-dom';

const App = () => (
  <main>
    <Switch>
      {/* Если текущим URL является /about, рандерится данный маршрут,
          остальные игнорируются */}
      <Route path="/about">
        <About />
      </Route>

      {/* Обратите внимание на порядок расположения этих двух маршрутов.
          Более специфический path="/contact/:id" находится перед path="/contact" */}
      <Route path="/contact/:id">
        <Contact />
      </Route>
      <Route path="/contact">
        <AllContacts />
      </Route>
      {/* Если ни один из предыдущих роутеров не совпал,
          рендерится данный маршрут (он является резервным).

          Важно: маршрут с path="/" всегда будет совпадать с
          URL, поскольку все URL начинаются с /. Поэтому
          мы поместили его последним */}
      <Route path="/">
        <Home />
      </Route>
    </Switch>
  </main>
);

ReactDOM.render(
  <Router>
    <App />
  </Router>,
  document.getElementById('root')
);
```

<Route path> ищет совпадение с началом, а не со всем URL. Поэтому <Route path="/"> всегда будет совпадать с URL. Поэтому в <Switch> его помещают последним. Другим решением является использование атрибута exact: <Route exact path="/">, который заставляет роутер искать полное совпадение.

Навигация

React Router предоставляет компонент <Link> для создания ссылок в приложении. При использовании <Link> в HTML рендерится <a>.

```
<Link to="/">Главная</Link>
// <a href="/">Главная</a>
```

<NavLink> - это специальный тип <Link>, позволяющий определять стили для активного состояния ссылки.

```
<NavLink to="/react" activeClassName="hurray">
  React
</NavLink>

// Когда URL является /react, рендерится это:
// <a href="/react" className="hurray">React</a>

// Когда URL является другим:
// <a href="/react">React</a>
```

Для принудительной навигации используется <Redirect>. При рендеринге <Redirect> выполняется перенаправление.

```
<Redirect to="/login">
```

Маршрутизация - https://metanit.com/web/react/4.1.php

Задание

1. С помощью условного рендеринга вывести форму регистрации или авторизации пользователю (верстку форм можно взять из предыдущих заданий)
2. Используя хук UseEffect передать в title страницы имя пользователя. Иначе вывести "не авторизован"
3. Записать данные пользователя в LocalStorage
4. С помощью роутинга сделать меню, через которое можно открыть страницы: Главная, О нас, Товары. На странице товаров разместить 3 ссылки на товары и организовать переходы на эти страницы. (мне сама информация не нужна. Достаточно просто переход и там увидеть сообщение)