GeekBrains

ReactJS. Базовый курс

Children. Роутинг в React

[React 17.0.1]

На этом уроке

1. Познакомимся с использованием children и паттерном render props.
2. Узнаем, как работает роутинг в приложениях на React.
3. Научимся использовать react-router-dom для настройки роутинга в приложении.
4. Узнаем об использовании URL-параметров.
5. Познакомимся с хуками useMatch и useParams.

Оглавление

Теория урока

Children и render props

Children

Каждый компонент может иметь проп children - в него будут записаны все элементы, заключенные в тег данного компонента. К примеру, при следующей записи:

```
<Button onClick={handleClick}>
  <div>CHILDREN</div>
</Button>
```

В свойство children компонента Button попадет CHILDREN .

Компонент Button будет выглядеть приблизительно следующим образом:

```
▼ {$$typeof: Symbol(react.element), key: null, ref: null, props: {…}, type: ƒ, …}
    $$typeof: Symbol(react.element)
    key: null
  ▼ props:
    ▶ children: {$$typeof: Symbol(react.element), type: "div", key: null, ref: null, props: {
    ▶ __proto__: Object
    ref: null
  ▶ type: ƒ App()
    _owner: null
  ▶ _store: {validated: false}
```

Таким образом, все элементы, переданные внутри тега компонента Button, доступны через props.children. Это удобно для создания переиспользуемых компонентов - к примеру, мы можем использовать один компонент для кнопок с иконкой, изображением, текстом с любыми стилями:

```
function Example() {
 <>
   <Button onClick={doSmth}>
     <span style={{ fontStyle: "italic" }}>Child</span>
   </Button>

   <Button>Text second</Button>

   <Button>
    <img
     src="src-for-image"
     style={{ width: 100, height: 50 }}
     alt=""
    />
    </Button>
```

```
    <Button children={<div>Child</div>} />
  </>
}
```

Проп children используется во многих механизмах и компонентах React, а также сторонних библиотек - одним из самых известных примеров является библиотека react-router-dom.

Кроме того, Реакт предоставляет набор утилит для облегчения работы с children - React.Children (подробнее на официальном сайте).

Render props

Другим способом контролировать рендер дочернего компонента является паттерн render props - в этом случае родительский компонент передает дочернему функцию render (функция не обязательно должна называться именно render, но это общепринятый подход). Дочерний компонент, в свою очередь, может вызвать props.render с некоторыми аргументами.

```
const Image = ({ render }) => {
 return <div>{render("imageWrapper")}</div>;
};

const App = () => {
 const [imageSrc, setImageSrc] = useState("someImgSrc");

 return (
   <Image
     render={(className) => (
       <div className={className}>
         <img src={imageSrc} />
       </div>
     )}
   />
 );
};
```

Здесь с помощью проп render родительский компонент полностью контролирует отображение дочернего компонента. В большинстве случаев достаточно использовать передачу данных через пропсы. Render props используется, как правило, для создания гибких переиспользумых компонентов.

Хороший пример использования render props приведен на официальном сайте React.

В отличие от children, render props - не внутренний механизм Реакт, а паттерн. Свойство props.children будет присвоено компоненту автоматически, если внутри тега компонента есть какие-либо элементы. О том, чтобы вызвать свойство props.render, должен позаботиться сам разработчик.

Функция children

Помимо передачи в children компонентов, Реакт позволяет передавать функции:

```
const Image = ({ children }) => {
 return <div>{children("imageWrapper")}</div>;
};

const App = () => {
 const [imageSrc, setImageSrc] = useState("someImgSrc");

 return (
   <Image>
     {(className) => (
       <div className={className}>
         <img src={imageSrc} />
       </div>
     )}
   </Image>
 );
};
```

Такой способ является, в некотором смысле, чем-то средним между children - компонентами и паттерном render props. Аналогично последнему, он позволяет родительскому компоненту полностью контролировать внешний вид дочернего, вместе с тем оставляя гибкость переиспользования дочернего компонента. Подробнее о таком способе создания компонентов можно прочитать здесь.

React Router

Роутинг в Реакт

В веб-приложении роутинг, упрощенно говоря - организация маршрутов, т.е. все, что связано с переходом пользователем по ссылкам, изменении URL и т.п.

На стандартном сайте при переходе по ссылке браузер запрашивает у сервера страницу по соответствующему адресу (href атрибут тега <a>). После этого сервер отправляет в ответ html страницы, а браузер загружает и отрисовывает ее. В случае с одностраничными приложениями при переходе по ссылкам новая страница не загружается, а весь роутинг осуществляется на фронте.

Для настройки маршрутизации используется библиотека react-router-dom. С ее помощью приложение контролирует то, какие компоненты отображаются для каждого URL (каждой страницы). Упрощенно ее использование выглядит следующим образом:

```
export default function Router() {
  return (
    <BrowserRouter>
```

```
      <ul>
        <li>
          <Link to="/profile">profile</Link>
        </li>

        <li>
          <Link to="/chats">chats</Link>
        </li>

        <li>
          <Link to="/">Home</Link>
        </li>
      </ul>

      <Switch>
        <Route path="/profile">
          <Profile />
        </Route>
        <Route
          exact
          path="/chats"
        >
          <Chats />
        </Route>

        <Route exact path="/">
          <Home />
        </Route>
      </Switch>
    </BrowserRouter>
  );
}
```

Обратите внимание - все компоненты, которым необходимо иметь доступ к данным о роутинге, обернуты в компонент BrowserRouter. Это один из видов роутеров, предоставляемых данной библиотекой.

Всего таких вариантов четыре:

1. BrowserRouter - один из наиболее часто используемых роутеров. Адреса страниц при его использовании выглядят следующим образом:

```
http://localhost:3000/chats
```

> Этот вариант - самый привычный и удобный с точки зрения UX, однако может потребовать дополнительной настройки на стороне сервера (сервер должен корректно обрабатывать запрос не только главной страницы, на и страницы любой вложенности)

2. Hash router - использует хэш для хранения адреса. Url той же страницы при его использовании выглядит так:

```
http://localhost:3000#chats
```

   Как правило, используется для навигации по якорям в рамках одной страницы. Не требует дополнительной настройки на стороне сервера.

3. MemoryRouter - не использует адресную строку и хранит маршрут только в памяти. В основном используется при тестировании.
4. StaticRouter - роутер, который никогда не изменяет url. Используется в основном при настройке SSR (server side rendering) и в простых тестах.

   В данном курсе будет использоваться BrowserRouter.

Помимо самих роутеров, react-router-dom предоставляет несколько основных компонентов:

1. Link - обертка над тегом <a>, служит для создания ссылки. Принимает проп to, в который необходимо передать адрес страницы (относительно базового URL). Использование выглядит следующим образом:

```
<Link to="/">Home</Link>
```

2. Route - компонент-обертка для ваших компонентов, принимает проп path. Роутер отобразит компонент, находящийся внутри Ruote, path которого совпадает с текущим URL (хотя бы частично):

```
<Route path="/profile">
// компонент Profile будет отображен, если относительный путь начинается с
// '/profile', т.е., и в случае адреса '/profie123'
  <Profile />
</Route>
```

   Помимо path, Route принимает проп exact - в этом случае дочерний компонент Route будет отрендерен, если URL и path совпадают точно:

```
<Route exact path="/profile">
// компонент Profile будет отображен, только если относительный путь точно
// равен '/profile'
  <Profile />
</Route>
```

Другие способы передачи компонента в Route:

```
<Route path="/path1" render={() => <Component />} />
<Route path="/path2" component={Component} />
```

3. Switch - в данный компонент необходимо обернуть все Route вашего приложения. Работает аналогично switch-case - перебирает по порядку path переданных Route, пока не найдет совпадающий с текущим URL.

```
export default function Router() {
  return (
    <BrowserRouter>
      <ul>
        <li>
          <Link to="/profile">profile</Link>
        </li>

        <li>
          <Link to="/chats">chats</Link>
        </li>

        <li>
          <Link to="/">Home</Link>
        </li>
      </ul>

      <Switch>
        <Route path="/profile">
          <Profile />
        </Route>
        <Route
          exact
          path="/chats"
        >
          <Chats />
        </Route>

        <Route exact path="/">
          <Home />
        </Route>
      </Switch>
    </BrowserRouter>
  );
}
```

4. Redirect - служит для перенаправления пользователя на некоторую страницу. Как только этот компонент будет отрендерен, URL изменится на переданный в проп to. Данный компонент удобно использовать с условным рендером:

```
if (!chat) {
  return (
    <Redirect to="/chats/id1" />
  )
}
```

Внимание!

Все вышеперечисленные компоненты должны находиться внутри одного из роутеров (в данном случае BrowserRouter). Вкладывать один роутер в другой нельзя, однако вкладывать друг в друга Switch, Route, Link и Redirect допустимо (подробнее в разделе useRouteMatch).

Настройка и подключение роутера

Для настройки роутинга, прежде всего, следует установить react-router-dom:

```
npm i --save react-router-dom
```

Затем создадим компонент Routes и все компоненты нашего приложения обернем в BrowserRouter, а также настроим маршруты для каждого из основных компонентов:

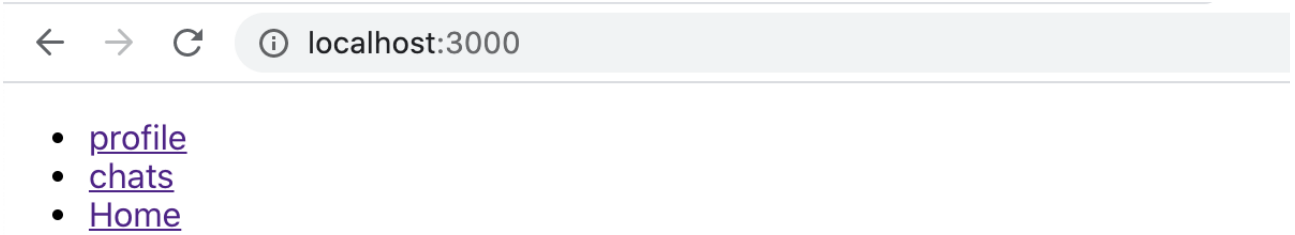
```
export default function Routes() {
 return (
   <BrowserRouter>
      <Switch>

        <Route exact path="/">
          <Home />
        </Route>

        <Route path="/profile">
          <Profile />
        </Route>

        <Route
          exact
          path="/chats"
        >
          <Chats />
        </Route>

      </Switch>
```

```
    </BrowserRouter>
  );
}
```

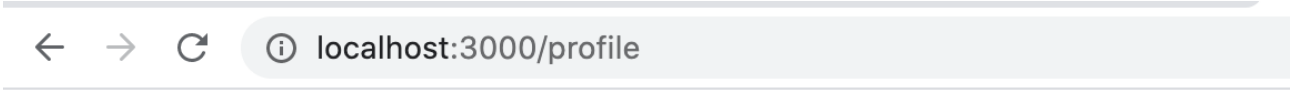
Над Switch добавим Header - отображение этого компонента не зависит от текущей страницы, он будет содержать ссылки на различные URL и будет виден на всех страницах приложения:

```
    <header>
      <ul>
        <li>
          <Link to="/profile">profile</Link>
        </li>

        <li>
          <Link to="/chats">chats</Link>
        </li>

        <li>
          <Link to="/">Home</Link>
        </li>
      </ul>
    </header>
```

Теперь основная страница нашего приложения будет выглядеть следующим образом:



При клике на ссылку My Profile компонент Link позаботится о том, чтобы изменить URL на ‘/profile’, а компонент Switch при этом отобразит компонент Profile:



- profile
- chats
- Home

Profile

Обратите внимание, что Route, отвечающий за главную страницу (path=”/”), имеет проп exact. Если не указать его, то компонент Home будет рендериться всегда - так как любой маршрут будет содержать подстроку ‘/’. Другим способом решения этой проблемы является перемещение такого маршрута в конец списка Route:

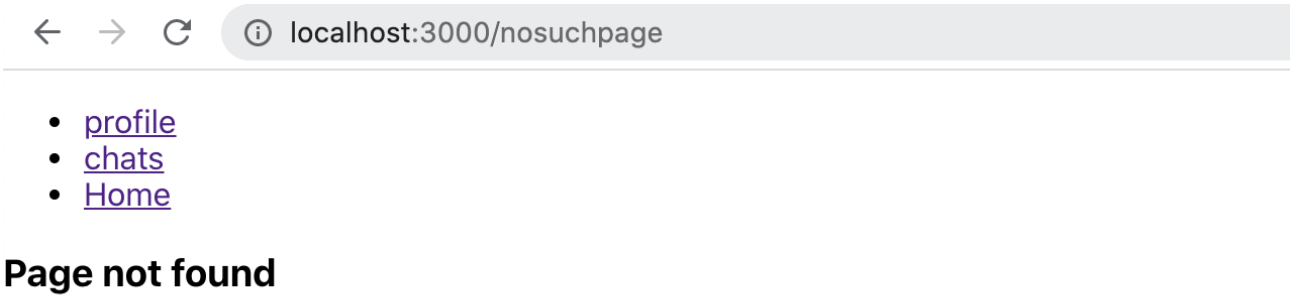
```
export default function Routes() {
 return (
   <BrowserRouter>
      <Switch>

        <Route path="/profile">
          <Profile />
        </Route>

        <Route
          exact
          path="/chats"
        >
          <Chats />
        </Route>

        <Route path="/">
          <Home />
        </Route>

      </Switch>

   </BrowserRouter>
 );
}
```

Теперь при переборе маршрутов внутри Switch выполнение дойдет до <Route path=”/”> только если адрес не совпал ни с одним из предыдущих path.

Внимание!

Если добавить в конце списка Route еще один Route без указания path, то его дочерний компонент будет отрендерен для всех маршрутов, не соответствующих перечисленным ранее. То есть, такой компонент будет работать как страница 404:

```
export default function Routes() {
 return (
   <BrowserRouter>
      <Switch>

        <Route path="/profile">
          <Profile />
        </Route>

        <Route
          exact
          path="/chats"
        >
          <Chats />
        </Route>

        <Route exact path="/">
          <Home />
        </Route>

        <Route>
          <h3>Page not found</h3>
        </Route>


      </Switch>

   </BrowserRouter>
 );
}
```



Параметры URL, useParams, useMatch

useParams

На текущий момент у нас создано приложение с тремя маршрутами - /home, /chats, /profile. До сих пор наш список чатов был неактивным, и отображал статичный список. Теперь мы добавим функционал переключения между чатами с помощью роутера.

Заметьте, что при переключении между чатами у нас будут изменяться только данные (т.е., сообщения чата), при этом сам компонент, отображающий эти сообщения, остается тем же (в данном примере - MessagesList). Это верно и для компонента ChatList - будет изменяться только то, какой чат выбран, а сам компонент при этом остается тем же. В связи с этим устанавливать и определять выбранный чат будем с помощью параметров url.

React-router позволяет добавлять для маршрута т.н. urlParam - параметр URL. При необходимости использовать параметр для маршрута имя этого параметра указывается в конце маршрута после двоеточия:

```
<Route path="/chats/:chatId">
  <Chats />
</Route>
```

При такой записи для компонента Chats будет доступен параметр chatId, причем он будет равен строке - последней части текущего url.

То есть, при таком маршруте:

```
http://localhost:3000/chats/12
```

chatId в компоненте chats будет равен “12” (заметьте что параметр - строка, при строгом сравнении с числом необходимо привести типы!).

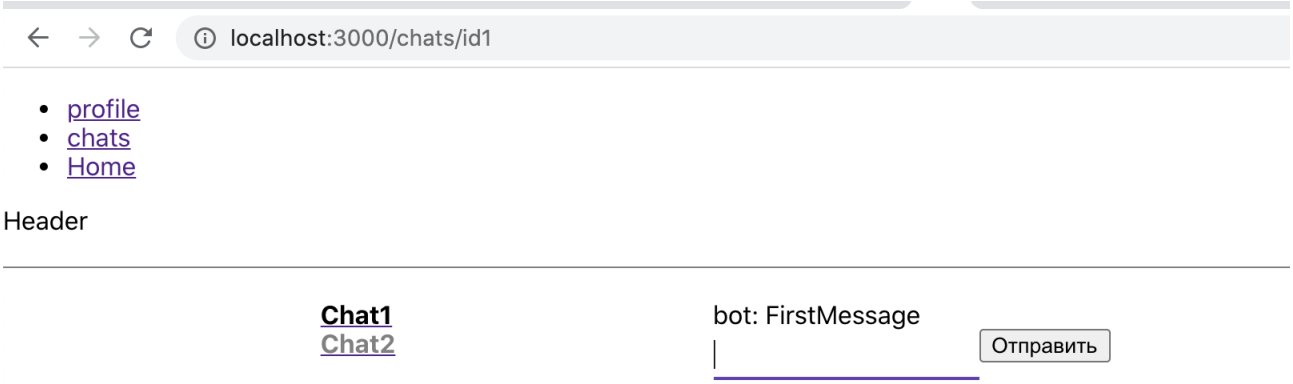
Получение параметра в Chats выглядит следующим образом:

```
export default function Chats() {
 const params = useParams();
 // или
 const { chatId } = useParams();

 return (/* ... */ );
}
```

Хук useParams предоставляет нам доступные параметры url и обеспечивает обновление компонента при их изменении.

Теперь, получив из параметров url айди чата, мы можем получить из объекта chats выбранный чат и отобразить сообщения этого чата:

```
const initialChats = {
 id1: {
   name: "Chat1",
   messages: [{ text: "FirstMessage", author: AUTHORS.BOT }],
 },
 id2: {
   name: "Chat2",
   messages: [{ text: "FirstMessageHereToo!", author: AUTHORS.ME }],
 },
};


export default function Chats() {
 const { chatId } = useParams();

 const [chats, setChats] = useState(initialChats);

 return (
   <>
     <header>Header</header>
     <div className="wrapper">
       <div>
         <ChatList
           chats={chats}
           chatId={chatId}
         />
       </div>
       <div>
         <MessagesList messages={chats[chatId].messages} />
       </div>
     </div>
   </>
 );
}
```



Теперь, вводя в адресную строку айди различных чатов, мы увидим отображение соответствующих сообщений. Однако, если мы введем айди несуществующего чата - мы увидим ошибку:

Это происходит из-за того, что мы обращаемся к свойству несуществующего объекта (selectedChat === undefined).

Это можно исправить различными способами, самый простой - не отображать ничего, если чат не найден.

```
export default function Chats() {
 const { chatId } = useParams();

 const [chats, setChats] = useState(initialChats);
```

```
 if (!chats[chatId]) {
   return null
 }

 return (
   <>
     <header>Header</header>
     <div className="wrapper">
       <div>
         <ChatList
           chats={chats}
           chatId={chatId}
         />
       </div>
       <div>
         <MessagesList messages={chats[chatId].messages} />
       </div>
     </div>
   </>
 );
}
```

Более удобным для пользователя решением, однако, будет отображение соответствующего сообщения. Сделаем это с помощью компонента Redirect. Сперва добавим необходимый маршрут в список Route:

```
<Route path="/nochat">
  <NoChat />
</Route>
```

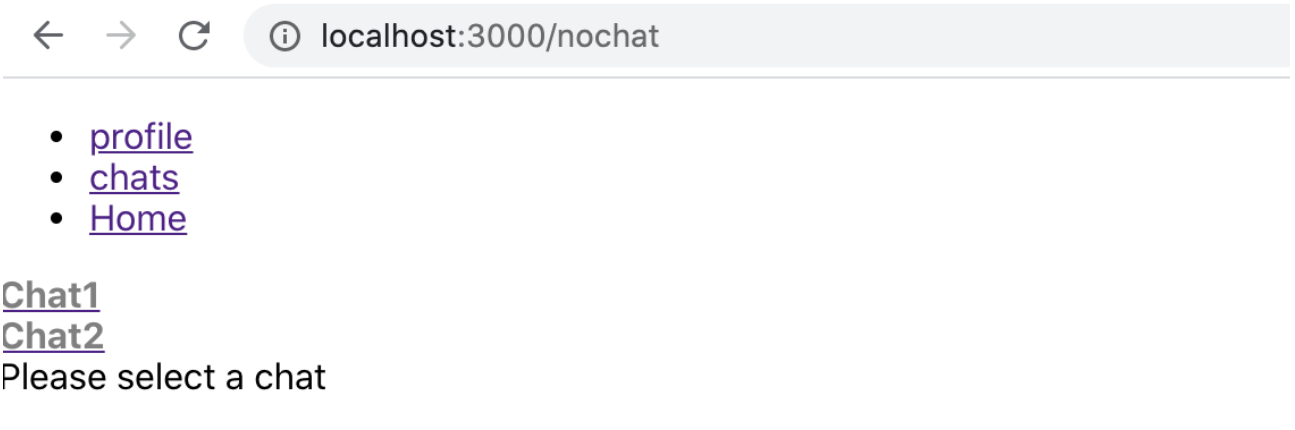
Затем создадим и импортируем компонент NoChats

```
export const NoChat = () => (
 <>
   <ChatList />
   <span>Please select a chat</span>
 </>
)
```

Затем добавим сам редирект:

```
 if (!chats[chatId]) {
   return <Redirect to="/nochat" />;
 }
```

Теперь, если пользователь введет адрес с айди несуществующего чата, он будет перенаправлен на страницу с сообщением о необходимости выбрать чат:



useMatch

React-router также предоставляет хук useRouteMatch. Он возвращает объект (обычно называемый match) со свойствами текущего маршрута:

Данный объект полезен при создании вложенных маршрутов. К примеру, мы можем использовать вложенные Switch и Link:

```
function Chats() {
 const [chats, setChats] = useState(initialChats);
 const { path, url } = useRouteMatch();

  return (
    <div>
      <h2>Chats</h2>
      <ChatList chats={chats} />

      <Switch>
        <Route exact path={path}>
          <h3>Please select a chat.</h3>
        </Route>
        <Route path={`${path}/:chatId`}>
          <MessageList chats={chats} />
        </Route>
      </Switch>
    </div>
  );
```

```
}
```

В этом случае следует устанавливать адреса Link и пути Route относительно текущего пути - для этого необходимо использовать свойства path и url объекта match. При этом следует учитывать, что параметр chatId будет доступен только внутри компонента MessageList, и логика для получения необходимых к отображению сообщений переносится в MessageList.

Для заданий данного курса использование вложенных маршрутов не обязательно.

Список чатов

На текущий момент мы можем переходить к чатам, только изменяя url в адресной строке. Добавим возможность перехода к чатам через ChatList. Для этого обернем каждый из элементов списка в Link и установим проп to, равный id чата:

```
export const ChatList = ({ chats, chatId }) => (
 <div>
   {Object.keys(chats).map((id, i) => (
     <div key={i}>
       <Link to={`/chats/${id}`}>
         <b style={{ color: id === chatId ? "#000000" : "grey" }}>
           {chats[id].name}
         </b>
       </Link>
     </div>
   ))}
 </div>
);
```

Кроме того, компоненту NoChat также требуется список чатов. Поднимем стейт в компонент Routes и будем передавать компонентам Chats и NoChat.

```
export default function Router() {
 const [chats, setChats] = useState(initialChats);

 return (
   <BrowserRouter>
     <header>
       <ul>
         <li>
           <Link to="/profile">profile</Link>
         </li>

         <li>
           <Link to="/chats">chats</Link>
         </li>
```

```
          <li>
            <Link to="/">Home</Link>
          </li>
        </ul>
      </header>

      <Switch>
        <Route path="/profile">
          <Profile />
        </Route>

        <Route
          path="/chats/:chatId"
        >
          <Chats chats={chats} setChats={setChats} />
        </Route>

        <Route path="/nochat">
          <NoChat chats={chats} />
        </Route>

        <Route exact path="/">
          <Home />
        </Route>

        <Route>
          <h3>Page not found</h3>
        </Route>
      </Switch>
    </BrowserRouter>
  );
}
```

Внимание!

Обратите внимание, что и список чатов в компоненте ChatList, и отображаемые сообщения используют в качестве источника данных один и тот же объект chats. Все компоненты, зависящие от некоторых данных, должны получать эти данные из одного и того же источника. В этом случае при изменении данных в этом источнике соответствующим образом изменятся и все зависящие от него компоненты. Иными словами, не стоит хранить отдельно список чатов для отображения в ChatList и список чатов для отображения сообщений в MessageField.

Необязательные параметры

В компоненте Route мы также можем указать необязательные параметры - то есть те, которых может не быть в адресе. Сделаем параметр chatId необязательным. Это делается с помощью указания вопросительного знака после параметра:

```
<Route
  path="/chats/:chatId?"
>
  <Chats chats={chats} setChats={setChats} />
</Route>
```

А в компоненте Chats добавим проверку на существование chatId:

```
if (!chatId || !chats[chatId]) {
  return <Redirect to="/nochat" />;
}
```

Глоссарий

1. Рендер-проп - проп-функция, вызываемая дочерним компонентом, с помощью которой родительский компонент контролирует возвращаемое значение дочерним.
2. Children - проп, с помощью которого компонент может получить доступ к другим компонентам или функциям, помещенным внутри его тега.
3. Единый источник правды (Single Source of Truth) - принцип, согласно которому все компоненты и функции, зависящие от некоторых данных, должны получать их из одного и того же источника.

Домашнее задание

1. Установить react-router-dom. Добавить домашнюю страницу по адресу “/” со списком ссылок на страницу чатов и страницу профиля.
2. Добавить страницу профиля (пока не несет никакой функциональности, можно сделать ее пустой).
3. Настроить разделение приложения на чаты с помощью роутера (использовать параметры url). Приложение должно корректно работать, если пользователь вводит идентификатор несуществующего чата или если идентификатора чата нет (т.е. адрес “/chats/”).
4. * Добавить возможность удаления и добавления чатов.

Дополнительные материалы

1. Подробно о React.Children
2. Функция в качестве children
3. Render props
4. Вложенные маршруты

Используемые источники

1. Официальный сайт react-router-dom
2. props.children - официальный сайт React