Проект "React Blog". Хук useEffect, кастомные хуки, React Router

Перенос разметки и данных

В **index.js:**
import **reportWebVitals** from './reportWebVitals'; - тесты приложения
Удалим его.
<**React.StrictMode**> - используется при отладке

Создадим **src/index.css, src/Navbar.js, src/Home.js, src/BlogList.js**

В **Home.js** создадим данные для постов:
const **blog** = [...]

В **BlogList.js** создадим

```
const BlogList = ({ blog: posts }) => {  //деструктуризация
 return (
  <div className="blog">
   {posts.map((post) => (   // в jsx вместо фигурных используем круглые скобки
    <div className="post-preview" key={post.id}>
     <h2>post.title</h2>
     <p>post.author</p>
     <button className="btn-delete">Delete</button>
    </div>
   ))}
  </div>
 );
};

export default BlogList;
```

Состояние компонента Home. Удаление постов

```
const Home = () => {
 const [posts, setPosts] = useState([
  {
   title: "My First Blog",
   body: "Lorem ipsum dolor sit amet consectetur adipisicing elit. Rem fuga nisi cum earum odio assumenda provident laborum quisquam ipsum,
eaque accusantium maiores incidunt atque porro dicta dolores ad soluta modi?",
   author: "John Doe",
   id: 1,
  } ...
 ]);

 const handleDelete = (id) => {
   const newPosts = posts.filter((post) => post.id !== id);
  setPosts(newPosts);
 };

 return (
  <div className="home">
   <BlogList posts={posts} handleDelete={handleDelete} />>
  </div>
 );
};
```

React hook useEffect

useEffect-tab - делается импорт из react

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

useEffect(() => {
 })
```

Функция внутри хука вызывается каждый раз, когда происходит **отрисовка данного компонента** на странице

Заметим, что в строгом режиме в **index.js** все компоненты первично **рендерятся два раза** и только в режиме разработки:
<React.StrictMode> <App /> **</React.StrictMode>**

useEffect и его зависимости

При вызове хука useEffect мы можем передавать **зависимости** (второй аргумент)

1. Рассмотрим простой случай, когда хук вызывается только **один раз при первичном рендере** данного компонента. Для этого вторым аргументом передаем пустой массив:

```
useEffect(() => {
  console.log("useEffect run");
}, []);
```

2. Случай, когда хук вызывается **при первичном рендере и при изменении определенного значения состояния posts**

```
useEffect(() => {
  console.log("useEffect run");
}, [posts] );
```

Этот хук используется часто, чтобы вставлять в него **код для получения данных с сервера**, н-р, с помощью **fetch**

! Частая ошибка, когда **useEffect** выполняется **бесконечно** - это, когда внутри него **обновляется состояние.** В этом случае нужно грамотно задать зависимости

json-server

Используется, чтобы поднять локальный сервер из json-файла и взаимодействовать с ним.

Установим сервер. Запустим новое окно терминала. В нем запустим команду:

npm i -g json-server

Создадим для него json-файл в корне приложения react-app/data/data.json и передадим туда данные по постам в таком виде:

```
{
 "posts": [
  {
   "title": "My First Blog",
   "body": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Rem fuga nisi cum earum odio assumenda provident laborum quisquam ipsum, eaque accusantium maiores incidunt atque porro dicta dolores ad soluta modi?",
   "author": "John Doe",
   "id": 1
  },
  {
   "title": "Second Post",
   "body": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Rem fuga nisi cum earum odio assumenda provident laborum quisquam ipsum, eaque accusantium maiores incidunt atque porro dicta dolores ad soluta modi?",
   "author": "Mary Jane",
   "id": 2
  },
  {
   "title": "Third Post",
   "body": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Rem fuga nisi cum earum odio assumenda provident laborum quisquam ipsum, eaque accusantium maiores incidunt atque porro dicta dolores ad soluta modi?",
   "author": "Tom Soyer",
   "id": 3
  }
 ]
}
```

Запускаем сервер следующей командой:

npx json-server --watch data/data.json --port 8000

При этом в терминале будут ссылки:

Resources
http://localhost:8000/posts

Home
http://localhost:8000

useEffect и fetch для получения данных

Используем useEffect для получения данных с сервера json

```
 useEffect(() => {
   console.log("useEffect run");
   fetch("http://localhost:8000/posts")
     .then((res) => {
       return res.json();
     })
     .then((data) => {
       console.log("data: ", data);
     });
 }, []);
```

Запишем их в state. изначально в state передадим **null**. У нас асинхронный запрос, и нам нужно сделать рендеринг уже после того, как получены данные с сервера. Для этого используем условный рендеринг:

```
const Home = () => {
 const [posts, setPosts] = useState(null);  // начальное значение state
 useEffect(() => {
   console.log("useEffect run");
   fetch("http://localhost:8000/posts")
     .then((res) => {
       return res.json();
     })
     .then((data) => {
       setPosts(data);  // записываем данные с сервера в state
     });
 }, []);

 return <div className="home">{posts && <BlogList posts={posts} />}</div>;  // используем условный рендеринг
};
```

Прелоадер для загрузки данных

```
const Home = () => {
 const [posts, setPosts] = useState(null);
 const [isLoading, setLoading] = useState(true);  // начальное значение state

 useEffect(() => {
   fetch("http://localhost:8000/posts")
     .then((res) => {
       return res.json();
     })
     .then((data) => {
       setPosts(data);
       setLoading(false);  // После получения ответа с сервера уберем прелоадер
     });
 }, []);

 return (
   <div className="home">
     {isLoading && <h3>Loading ...</h3>} // отображаем на странице до получения ответа с сервера
     {posts && <BlogList posts={posts} />}
   </div>
 );
};
```

Обработка ошибок в fetch

1. Ошибка **fetch**. Ловим с помощью **catch**
2. Статус результата **res отрицательный**. Бросаем ошибку сами

```
const Home = () => {
 const [posts, setPosts] = useState(null);
 const [isLoading, setLoading] = useState(true);
 const [error, setError] = useState(null); // состояние для ошибки (чтобы отображать текст на странице)
```

```
 useEffect(() => {
  fetch("http://localhost:8000/posts")
    .then((res) => {
      if (res.ok !== true) {  // Статус результата res отрицательный
        throw Error("Could not fetch the data from this resource");
      }
      return res.json();
    })
    .then((data) => {
      setPosts(data);
      setLoading(false); // После получения ответа с сервера уберем прелоадер
      setError(null);  // Уберем ошибку, если данные загрузились
    })
    .catch((err) => {
      console.log("err: ", err.message);
      setError(err.message); // выведем ошибку  в состояние
      setLoading(false); // уберем прелоадер
    });
 }, []);

 return (
  <div className="home">
    {error && <div>{error}</div>}  // выведем текст ошибки на странице
    {isLoading && <h3>Loading ...</h3>}
    {posts && <BlogList posts={posts} />}
  </div>
 );
};
```

Кастомный hook useFetch

Код с fetch может использоваться во многим частях проекта. Создадим для этого отдельный кастомный хук useFetch
Создадим файл src/useFetch.js. Перенесем туда код из Home.js. Сделаем некоторые правки, чтобы сделать код универсальным. Также сделаем возврат переменных состояний.

useFetch.js

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

const useFetch = (url) => {  // url вынесем в параметр
 const [data, setData] = useState(null);  // переименуем posts в data
 const [isLoading, setLoading] = useState(true);
 const [error, setError] = useState(null);

 useEffect(() => {
  fetch(url)
    .then((res) => {
      if (res.ok !== true) {
        throw Error("Could not fetch the data from this resource");
      }
      return res.json();
    })
    .then((data) => {
      console.log("data: ", data);
      setData(data);
      setLoading(false);
      setError(null);
    })
    .catch((err) => {
      setError(err.message);
      setLoading(false);
    });
 }, []);

 return { data, isLoading, error };  // Вернем переменные состояния
};
export default useFetch;
```

Home.js:

```
import useFetch from "./useFetch";
 const Home = () => {
  const { data, isLoading, error } = useFetch("http://localhost:8000/posts");
  return (
    <div className="home">
      {error && <div>{error}</div>}
      {isLoading && <h3>Loading ...</h3>}
      {data && <BlogList posts={data} />}
    </div>
  );
};
```

React Router

Откроем еще один терминал. Установим **React Router**:
npm i react-router-dom

Будем использовать внутри **App.js**. Сначал сделем импорт.

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

function App() {
  return (
    <Router>
      <div className="app">
        <Navbar />
        <main className="main">
          <Routes>
            <Route path="/" element={<Home />} />
          </Routes>
        </main>
      </div>
    </Router>
  );
}
```

Новая страница New Post

Добавим в Navbar.js ссылки на страницы:

```
<div className="links">
   <a href="/">Home</a>
   <a href="/create">New post</a>
 </div>
```

Создадим новый компонент для страницы New Post в файле src/Create.js:

```
const Create = () => {
      return (
            <div className='create'>
                  <h2>Add a new blog!</h2>
            </div>
      );
};

export default Create;
```

В App.js добавим новый **Route**:

```
<Routes>
   <Route path="/" element={<Home />} />
   <Route path="/create" element={<Create />} />
</Routes>
```

Сейчас при переходе по ссылке все равно совершается запрос в браузере и страница загружается заново. Чтобы React перехватывал запрос и обрабатывал его самостоятельно, необходимо использовать **специальные компоненты для ссылок Link** которые есть в react router.

Новая страница New Post

Добавим специальные компоненты Link для ссылок в Navbar.js:

```
import { Link } from "react-router-dom";

const Navbar = () => {
    return (
        <nav className='navBar'>
            <h1>React Blog</h1>
            <div className='links'>
                <Link to="/">Home</Link>
                <Link to="/create">New post</Link>
            </div>
        </nav>
    );
};

export default Navbar;
```

useEffect Cleanup

Сделаем так чтобы в консоль выводились полученные данные из useFetch. Теперь сделаем быстрое переключение между страницами New Post и Home и снова New Post (пока не сработала задержка в **setTimeOut**). При возвращении на New Post мы увидим вывод в консоли полученных данных, но они нам больше не нужны. Такое поведение может провоцировать ошибки и утечки памяти. Поэтому нам надо остановить fetch когда компонент будет **размонтирован**.

В хуке useEffect можно вернуть функцию, и эта функция будет вызываться **в момент размонтирования состояния компонента**:

```
 useEffect(() => {
...
   return () => {
    console.log("cleanup");  //возвращаем функцию
   };
 }, []);
```

Создадим abortController Это специальный встроенный объект, который можно использовать для отмены fetch. Контроллер имеет единственный **метод abort()** и единственное **свойство signal**. При вызове **abort():**

- генерируется **событие** с именем **abort** на объекте
- controller.signal свойство **controller.signal.aborted** становится равным **true**.

Создадим abortController и привяжем его к определенному fetch запросу, в fetch передаем второй аргумент:

```
const abortCont = new AbortController();
...
fetch(url, {signal: abortCont.signal})
```

и будем вызывать данный abortController в cleanup функции

```
return () => {
    console.log("cleanup");
    abortCont.abort()
 }
```

Когда мы **останавливаем fetch**, он выбрасывает ошибку. И в этом случае наш хук все равно будет **продолжать обновлять state**. Поэтому в блоке с catch добавим условие. Если ошибка возникла в результате abort fetch() который мы запускаем сами в cleanUp функции, тогда мы не будем обновлять state:

```
if (error.name === "AbortError") {
   console.log('fetch aborted');
} else {
   setLoading(false)
   setError(error.message)
}
```

Сделаем пару быстрых переходов и увидим сообщение в консоли **fetch aborted**

Итого в UseFetch.js:

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

const useFetch = (url) => {
```

```
  const [data, setData] = useState(null);
  const [isLoading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    console.log("useEffect run");

    const abortCont = new AbortController();

    setTimeout(() => {
      fetch(url, { signal: abortCont.signal })
        .then((res) => {
          if (res.ok !== true) {
            throw Error("Could not fetch the data from this resource");
          }
          return res.json();
        })
        .then((data) => {
          console.log(data);
          setData(data);
          setLoading(false);
          setError(null);
        })
        .catch((err) => {
          if (err.name === "AbortError") {
            console.log("fetch aborted");
          } else {
          setError(err.message);
          setLoading(false);
          }
        });
    }, 1000);

    return () => {
      console.log("cleanup");
      abortCont.abort();
    };
  }, []);

  return { data, isLoading, error };
};

export default useFetch;
```

Route Parameters

Примеры запросов с параметрами:
/blogs/**123**
/blogs/**456**

Создадим новый компонент для отдельных постов src/BlogDetails.js:

```
const BlogDetails = () => {
    return (
        <div className="blog-details">
            <h2>Blog Details</h2>
        </div>
    );
}

export default BlogDetails;
```

Добавим новый маршрут в App.js

```
<Route path="/blogs/:id" element={<BlogDetails />}/>
```

Здесь id - параметр Далее мы можем поочереди зайти по адресам
http://localhost:3000/blogs/2
http://localhost:3000/blogs/3

И увидеть страницу с компонентом `BlogDetails.js`

Добавим получение параметров в компоненте `BlogDetails.js`.
Импортируем useParams из "react-router-dom". Получим параметры с помощью функции useParams(). Выведем параметр id в заголовок:

```
import { useParams } from "react-router-dom";

const BlogDetails = () => {
 const { id } = useParams();
 return (
  <div className="blog-details">
   <h2>Blog Details - {id}</h2>
  </div>
 );
};
export default BlogDetails;
```

Теперь добавим ссылки на каждый пост. Открываем компонент BlogList.js и добавляем компонент Link, сделав его импорт

```
import { Link } from "react-router-dom";

const BlogList = ({ posts }) => {
    return (
        <div className='blog'>
            {posts.map((post) => (
                <div className='post-preview' key={post.id}>
                    <Link to={`/blogs/${post.id}`}>
                        <h2>{post.title}</h2>
                        <p>{post.author}</p>
                    </Link>
                </div>
            ))}
        </div>
    );
};
```

Вывод полного поста. Использование кастомного хука useFetch()

Сделаем получение данных для постов в `BlogDetails.js`

1. Импортируем хук `**useFetch**`
2. Запустим `**useFetch**`
3. Выведем в шаблоне загрузку `**isLoading**`
4. Выведем в шаблоне возможную ошибку `**error**`
5. Сделаем **вывод блога**

```
import { useParams } from "react-router-dom";
import useFetch from "./useFetch";

const BlogDetails = () => {
 const { id } = useParams();
 const { data: blog, isLoading, error } = useFetch("http://localhost:8000/posts/" + id);

 return (
  <div className="blog-details">
   {isLoading && <h3>Loading ...</h3>}
   {error && <div>{error}</div>}
   {blog && (
    <article>
     <h2>{blog.title}</h2>
     <p className="author">Written by:{blog.author}</p>
     <div>{blog.body}</div>
    </article>
   )}
  </div>
 );
};

export default BlogDetails;
```

Форма для добавления нового поста

В Create.js заверстаем форму. Создаем state для того чтобы связать его с формой

```
 const [title, setTitle] = useState("");
 const [body, setBody] = useState("");
 const [author, setAuthor] = useState("Mary Jane");
<form>
    <label htmlFor="">Post title</label>
    <input type="text" required value={title} onChange={(e)=> setTitle(e.target.value)}/>

    <label htmlFor="">Post content</label>
    <textarea required value={body} onChange={(e)=> setBody(e.target.value)}></textarea>

    <label>Author</label>
    <select value={author} onChange={(e) => setAuthor(e.target.value)}>
        <option value="john doe">John Doe</option>
        <option value="mary jane">Mary Jane</option>
        <option value="tom soyer">Tom Soyer</option>
    </select>

    <button>Create Post</button>
</form>
```

Добавление нового поста. fetch запрос

В Create.js вешаем обработку на отправку формы **handleSubmit** и пишем **post запрос fetch**

```
 const handleSubmit = (e) => {
   e.preventDefault();
   const blog = { title, body, author };
   fetch("http://localhost:8000/posts", {
     method: "POST",
     headers: { "Content-Type": "application/json" },
     body: JSON.stringify(blog),
   }).then(() => {
     console.log("New post was added!");
   });
 };
...
<form onSubmit={handleSubmit}> ...
```

Добавление поста. Блокирование кнопки на время отправки данных

Далее добавим поведение для кнопки **Add Post**. Сделаем ее неактивной на время добавления нового поста. Для этого создадим отдельный state:

```
const [isPending, setIsPending] = useState(false)
```

При добавлении нового поста (в функции отправки формы после preventdefault) будем менять этот state на **true**. А когда известно что пост добавлен (в then после ответа с сервера), обратно в **false**.

Сделаем **условный вывод** для кнопки добавления поста:

```
{isPending && <button disabled>Adding Post...</button>}
{!isPending && <button>Create Post</button>}
```

Чтобы протестировать, можно добавить **setTimeout** с задержкой для **fetch**. Также нужно **очистить форму**
Вся функция **handleSubmit** будет выглядеть следующим образом:

```
 const handleSubmit = (e) => {
   e.preventDefault();
   const blog = { title, body, author };
   setIsPending(true);

   setTimeout(() => {
     fetch("http://localhost:8000/posts", {
       method: "POST",
       headers: { "Content-Type": "application/json" },
       body: JSON.stringify(blog),
```

```
    }).then(() => {
      console.log("New post was added!");
      setIsPending(false);
      setTitle('');
      setBody('');
      setAuthor("Mary Jane");
    });
  }, 2000);
};
```

Для **select** лучше задать значение по умолчанию

Редирект

Сделаем так, чтобы после добавления нового поста пользователь автоматически перемещался на главную страницу объекта В Create.js импортируем навигацию, запустим функцию **useNavigate(),** записав результат который она вернет в **navigate**, ,будем передавать в нее новый адрес, после успешного добавления поста сделаем редирект:

```
const navigate = useNavigate();
...
navigate("/");
```

Удаление поста

Откроем файл BlogDetails.js В текст поста добавим кнопку и событие клик на нее:

```
<button onClick={deletePost} className='btn-delete'>Delete</button>
```

Опишем функцию deletePost:

```
const deletePost = () => {
        fetch('http://localhost:8000/posts/' + blog.id,{
            method: 'DELETE'
        }).then(()=>{
            navigate('/');
        })
    }
```

И не забудем подключить **навигацию**

```
import { useNavigate } from 'react-router-dom';
...
const navigate = useNavigate();
```

Удаление постов с главной страницы. Удаление и обновление страницы

Добавим возможность удалять посты не только со страницы поста, но и с главной страницы блога.

В начале вынесем функцию **deletePost()** в отдельный файл deletePost.js, чтобы ее можно было вызывать и с главной страницы при выводе постов. Удалим из deletePost.js все сторонние эффекты, а именно редирект на главную. А также добавим в нее параметр id, так как при вызове на главной, из компонента BlogList функция не знает по умолчанию id поста который нужно удалить, поэтому мы передадим его в качестве параметра.

```
const deletePost = (id) => {
  fetch("http://localhost:8000/posts/" + id, {
    method: "DELETE",
  }).then(() => {
    console.log("post deleted");
  });
};

export default deletePost;
```

Импортируем **deletePost** в компонент BlogList.js и укажем ее в качестве обработчика для кнопки удаления поста. Также передадим в неё id поста который нужно удалить.

```
import deletePost from './deletePost'
...
```

```
<button onClick={()=>{deletePost(post.id)}} className='btn-delete'>Delete</button>
```

Отлично, посты удаляются. Но увидеть это мы можем только после **перезагрузки страницы**. А нам необходимо чтобы оставаясь на главной странице, у нас происходил повторный запрос **fetch** на сервер, получения обновленного списка постов, в которых уже нет удаленного и повторный ре-рендер компонента **Home** а вместе с ним и компонента **BlogList**. Ре-рендер запустится после **изменения состояния в useFetch**

Проблема состоит в том что данные мы получаем из кастомного хука **useFetch** и принудительно запустить повторно мы его не можем. К тому же результат первого запуска useFetch уже записан в константы data, isLoading, error и соответственно переопределить их мы также не можем.

Решение! Внутри хука useFetch мы используем хук **useEffect** в который мы определили **в зависимости пустой массив**, то есть он запускается только один раз, при отрисовке компонента Home. Если убрать второй аргумент, то будет срабатывать при любом изменен ии состояния. Это нам тоже не нужно.

Мы можем добавить отдельный **state** для компонента Home, который передадим **в зависимости для useEffect** (внутри useFetch). И при изменении данного элемента состояния компонента Home. useEffect будет запущен повторно, и соответственно заново сработает fetch и будет получен обновленный список постов.

Для начала удалим второй аргумент в useEffect в файле useFetch.js:

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

Добавим отдельное состояние в компонент Home.js. Это будет специальный **флаг**, при изменении значения которого, необходимо будет перезапускать **useEffect**

```
const [updateFlag, setUpdateFlag] = useState(false)
```

Теперь при удалении поста с главной, нам необходимо изменять этот флаг. Пробросим сам флаг и функцию для его обновления в компонент **BlogList**

```
{ data && <BlogList posts={data} setUpdateFlag={setUpdateFlag} updateFlag={updateFlag} /> }
```

После нам нужно чтобы функция `deletePost` после удаления поста совершала для нас дополнительное действие. В данном случае запускала `setUpdateFlag. Но это уже будет выходить за рамки ответственности данной функции. К тому же, при удалении поста со страницы поста, нам нужно будет совершать другое действие - редирект на главную страницу сайта. Поэтому поступим следующим образом. Доработаем функцию `**deletePost()**` таким образом, чтобы она могла принимать вторым аргументом функцию колбек, и запускать ее, если она была передана, после удаления поста.
Файл deletePost.js:

```
const deletePost = (id, callback) => {
  fetch('http://localhost:8000/posts/' + id,{
    method: 'DELETE'
  }).then(()=>{
    console.log('Post Deleted!');
    if( typeof callback === "function") callback()
  })
}

export default deletePost;
```

Опишем функцию колбек в BlogList.js и передадим ее вторым аргументом в deletePost

```
const BlogList = ({ posts, setUpdateFlag, updateFlag }) => {// Достанем флаг и ф-ю для его обновления
 return (
  <div className="blog">
   {posts.map((post) => (
    <div className="post-preview" key={post.id}>
     <Link to={`/blogs/${post.id}`}>
      <h2>{post.title}</h2>
      <p>{post.author}</p>
     </Link>
     <button
      className="btn-delete"
      onClick={() => {
```

```
          deletePost(post.id, setUpdateFlag(!updateFlag)); // Передадим ф-ю колбек в deletePost
        }}>
        Delete
       </button>
      </div>
    ))}
  </div>
 );
};
```

Зависимость useEffect от конкретного состояния компонента Home

Вспомним, что внутри useFetch.js в **useEffect** мы удалили второй аргумент (пустой массив - т.е. срабатывание только при старте). Сейчас useEffect будет срабатывать при **любом изменении** состояний компонента. Здесь это не проблема, так как состояние только одно. В общем случае сделаем это для конкретного состояния.

Определим параметр внутри useFetch, чтобы указать его в зависимости для useEffect. Файл useFetch.js:

```
const useFetch = (url, updateFlag) => {
...
useEffect(()=>{
...
}, [updateFlag]) // в массив передаем конкретные состояния, при изменении которых будет запускаться useEffect
```

И передадим его в useFetch при его запуске внутри Home.js

```
const {data, isLoading, error} = useFetch('http://localhost:8000/posts', updateFlag)
```

Переиспользование функции deletePost внутри BlogDetails

В файле **BlogDetails.js** импортируем **deletePost**. передадим **navigate** внутри колбека ф-ции **deletePost**. Файл BlogDetails.js

```
import { useParams } from "react-router-dom";
import useFetch from "./useFetch";
import { useNavigate } from "react-router-dom";
import deletePost from "./deletePost";

const BlogDetails = () => {
 const { id } = useParams();
 const { data: blog, isLoading, error } = useFetch("http://localhost:8000/posts/" + id);
 const navigate = useNavigate();  // Включим навигацию

 const afterDelete = () => { // 1. Напишем ф-ю колбек
  navigate("/");
 };

 return (
  <div className="blog-details">
   {isLoading && <h3>Loading ...</h3>}
   {error && <div>{error}</div>}
   {blog && (
    <article>
     <h2>{blog.title}</h2>
     <p className="author">Written by:{blog.author}</p>
     <div>{blog.body}</div>
     <button        {/* // 2. Передадим id поста и ф-ю колбек в deletePost */}
      onClick={() => {
       deletePost(blog.id, afterDelete);
      }}
      className="btn-delete">
      Delete
     </button>
    </article>
   )}
  </div>
 );
};
export default BlogDetails;
```

Страница 404

Создадим компонент src/NotFound.js который будет отображаться для страницы 404

```
import { Link } from "react-router-dom";

const NotFound = () => {
  return (
    <div className="not-found">
      <h2>Sorry</h2>
      <p>That page cannot be found</p>
      <Link to="/">Go to Homepage</Link>
    </div>
  );
}

export default NotFound;
```

В **App.js** добавим **самый последний** маршрут для адресов которые не соответствуют предыдущим маршрутам <Route path="*" element={<NotFound />} />

Полный код роутера в App.js

```
<Routes>
    <Route path="/" element={<Home />}  />
    <Route path="/create" element={<Create />}  />
    <Route path="/blogs/:id" element={<BlogDetails />} />
    <Route path="*" element={<NotFound />} />
</Routes>
```

Заметим, что мы не попадем на страницу 404, если введем номер поста, которого нет, н-р:

http://localhost:3000/blogs/500

Рефакторинг проекта. CSS и clearForm()

1. Очищение формы лучше вынести в отдельную функцию. В файле Create.js:

```
 const clearForm = () => {
   setTitle("");
   setBody("");
   setAuthor("Mary Jane");
 };
...
clearForm();
```

Почему не стоит объединять все вызовы fetch в кастомный хук useFetch

Внутри **useFetch** мы используем **useEffect**, который делает первичную отрисовку или отрисовку при изменении состояния. А когда мы создаем или удаляем пост, fetch запрос выполняется по событиям submit формы или клику по кнопке Удалить.