Урок 6

Роутинг в ReactJS приложении

Добавляем роутинг в приложение

Что такое роутинг

React это не фреймворк, а библиотека, и, следовательно, он не предназначен для решения всех потребностей приложения. Он отлично работает в создании компонентов и предоставляет систему для управления состоянием, но создание сложного SPA потребует дополнительных модулей.

Роутинг — это маршрутизация: входящий URL разбирается специальным образом и по его результату выполняется определённый код. Маршрутизация определяет, как приложение отвечает на клиентский запрос к конкретному адресу (URI). Приведённый ниже код служит примером одного из самых простых маршрутов.

```
var express =require('express');
var app =express();
// respond with "hello world" when a GET request is made to the homepage
app.get('/',function(req, res){
  res.send('helloworld');
});
```

Реализация маршрутизации

Чтобы понять, как работает маршрутизация, давайте рассмотрим пример. Мы создадим приложение, которое будет использовать GitHub API, получения списка репозиториев. Помимо этого добавим домашнюю страницу и «О проекте». Давайте начнём с главного компонента App:

App.js

```
import React,{Component}from'react';
import{render}from'react-dom';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
classAppextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            route:window.location.hash.substr(1)
        };
    }
    componentDidMount(){
        window.addEventListener('hashchange',()=>{
            this.setState({
                route:window.location.hash.substr(1)
            });
        });
    }
    render(){
        return(
            <h1>Hello World</h1>
        );
```

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

Код довольно прост. В конструкторе компонента получаем текущее значение хеш URL-адрес и присваиваем состояние маршрута. Затем, при создании компонента, добавляем обработчик события на изменение URL, в нём обновляется текущее состояние нашего компонента. Теперь нам надо обновить метод render:

App.js

```
...
    render(){
        varChild;
        switch(this.state.route){
            case'/about':
                Child=About;
                break;
            case'/repos':
                Child=Repos;
                break;
            default:
                Child=Home;
        }
        return(
            <div>
                <header>App</header>
                <menu>
                    <ul>
                        <li><ahref="#/about">About</a></li>
                        <li><ahref="#/repos">Repos</a></li>
                    </ul>
                </menu>
                <Child/>
            </div>
        )
    }
...
```

Проверяется текущее состояние route и в зависимости от него Child присваивается определенный компонент. Все дочерние компоненты, которые представляют собой внутренние страницы приложения, просты и имеют одинаковую структуру:

Home.js

```
import React,{Component}from'react';
classHomeextendsComponent{
    render(){
        return(
            <h1>HOME</h1>
        );
    }
}
export defaultHome;
```

About.js

```
import React,{Component}from'react';
classAboutextendsComponent{
    render(){
        return(
            <h1>ABOUT</h1>
        );
    }
}
export defaultAbout;
```

Repos.js

```
import React,{Component}from'react';
classReposextendsComponent{
    render(){
        return(
            <h1>Github Repos</h1>
        );
    }
}
export defaultRepos;
```

Несмотря на то, что это работает, в данном подходе есть, по крайней мере, две проблемы:

- В примере, содержимое URL заняло центральное место: вместо того, чтобы автоматически обновлялось URL и состояние приложения;
- Код маршрутизации может расти в геометрической прогрессии при сложных нетривиальных сценариях. Представим себе, например, что внутри страницы Repos может быть список репозиториев со ссылкой внутрь, что-то вроде /repos/repo_id.

Для сценариев более сложных, чем одноуровневая маршрутизация, рекомендуемый подход заключается в использовании библиотеки React Router.

Добавляем ReactRouter в наше приложение

React-router – библиотека, обладает возможностями по вложению роутов (nesting). Роутер React использует JSX, который на первый взгляд может показаться немного странным.

Поскольку React Router является внешней библиотекой, то она должна быть установлена с npm:

```
npm install —save react-router
```

React Router предоставляет три компонента для начала работы:

- Router и Route: Используются для декларативного описания карты маршрутов приложения;
- Link: Используется для создания ссылок с заданным href. Это не единственный способ навигации проекта, но все же основной.

Добавим React Router:

App.js

```
import React,{Component}from'react';
import{render}from'react-dom';
import{Router,Route,Link}from'react-router';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
classAppextendsComponent{
   render(){
      return(
         <div>
            <header>App</header>
            <menu>
               <ul>
                  <li><Link to="/about">About</Link></li>
                  <li><Link to="/repos">Repos</Link></li>
               </ul>
            </menu>
            {this.props.children}
         </div>
      );
   }
}
render(<App/>,document.getElementById('root'));
```

Здесь мы импортировали React Router и обновили метод render, добавив туда Link. Обновляем метод render:

App.js

```
import React,{Component}from'react';
import{render}from'react-dom';
import{Router,Route,Link}from'react-router';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
classAppextendsComponent{
   render(){
      return(
         <div>
            <header>App</header>
            <menu>
               <ul>
                  <li><Link to="/about">About</Link></li>
                  <li><Link to="/repos">Repos</Link></li>
               </ul>
            </menu>
            {this.props.children}
         </div>
      );
   }
}
render((
   <Router>
      <Route path="/"component={App}>
         <Route path="about"component={About}/>
         <Route path="repos"component={Repos}/>
      </Route>
   </Router>
),document.getElementById('root'));
```

Если попробовать запустить, то видно, что все работает как раньше, но если зайти «/» то не увидим результат компонента «Home». Указываем путь.

```
...
<Router>
  <Route path="/"component={App}>
    <Route path="???"component={Home}/>
    <Route path="about"component={About}/>
    <Route path="repos"component={Repos}/>
  </Route>
</Router>
...
```

Можно использовать IndexRoute. Надо просто импортировать его и использовать его для настройки маршрут индекса.

Основные возможности ReactRouter

React Router является наиболее популярным решением для добавления маршрутизации React приложению. Он помогает держать в синхронизации UI с URL с помощью компонентов, связанных с маршрутами (на любом уровне вложенности). При изменении URL адреса React Router автоматически монтирует и демонтирует нужные компоненты.

В качестве примера приведу код для рендеринга отдельного компонента:

```
//React JSX
var Home = React.createClass({
  render:function(){
return(<h1>Welcome to the Home Page</h1>);
}
});

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

А вот рендеринг того же компонента Home с роутером React:

```
//React JSX
...
ReactDOM.render((
<Router>
<Route path="/"component={Home}/>
</Router>
), document.getElementById('root'));
```

Отметьте, что <Router> и <Route> - это две разные вещи. Технически это компоненты React, но сами они не создают DOM. Может показаться, что <Router> выводится в 'root', но на самом деле он просто определяен правила работы нашего приложения. Данные компоненты существуют не для создания DOM, а для координации действий других компонентов. В примере <Route> определяет правило, что при посещении домашней страницы / в 'root' будет выводиться компонент Home. Метод route является производным от одного из методов HTTP и присоединяется к экземпляру класса express.

Приведённый ниже код служит примером маршрутов, определённых для методов запросов GET и POST к корневому каталогу приложения.

```
// GET method route
app.get('/',function(req, res){
  res.send('GET request to the homepage');
});

// POST method route
app.post('/',function(req, res){
  res.send('POST request to the homepage');
});
```

Для методов route, преобразуемых в недействительные имена переменных JavaScript, используйте нотацию в квадратных скобках. Например: app['m-search']('/', function ...

Существует особый метод маршрутизации, app.all(), не являющийся производным от какого-либо метода HTTP. Этот метод используется для загрузки функций промежуточной обработки в пути для всех методов запросов.

В приведённом ниже примере обработчик будет запущен для запросов, адресованных “/secret”, независимо от того, используется ли GET, POST, PUT, DELETE или какой-либо другой метод запроса HTTP, поддерживаемый в модуле http.

```
app.all('/secret',function(req, res, next){
  console.log('Accessing the secret section ...');
next();// pass control to the next handler
});
```

Определяем пути маршрута.

Пути маршрутов, в сочетании с методом запроса, определяют конкретные адреса (конечные точки), в которых могут быть созданы запросы. Пути маршрутов могут представлять собой строки, шаблоны строк или регулярные выражения. Строки запросов не являются частью пути маршрута.Ниже приводятся примеры путей маршрутов на основе строк.

Данный путь маршрута сопоставляет запросы с корневым маршрутом, /.

```
app.get('/',function(req, res){
  res.send('root');
});
```

Данный путь маршрута сопоставляет запросы с /about.

```
app.get('/about',function(req, res){
res.send('about');
});
```

Данный путь маршрута сопоставляет запросы с /random.text.

```
app.get('/random.text',function(req, res){
res.send('random.text');
});
```

Ниже приводятся примеры путей маршрутов на основе шаблонов строк.

Приведенный ниже путь маршрута сопоставляет acd и abcd.

```
app.get('/ab?cd',function(req, res){
  res.send('ab?cd');
});
```

Этот путь маршрута сопоставляет abcd, abbcd, abbbcd и т.д.

```
app.get('/ab+cd',function(req, res){
  res.send('ab+cd');
});
```

Дефис (-) и точка (.) интерпретируются буквально в путях на основе строк.

Обработчики маршрутов

Для обработки запроса можно указать несколько функций обратного вызова, подобных middleware. Единственным исключением является то, что эти обратные вызовы могут инициировать next('route') для обхода остальных обратных вызовов маршрута. С помощью этого механизма можно включить в маршрут предварительные условия, а затем передать управление последующим маршрутам, если продолжать работу с текущим маршрутом не нужно.

Обработчики маршрутов могут принимать форму функции, массива функций или их сочетания, как показано в примерах ниже.

Одна функция обратного вызова может обрабатывать один маршрут. Например:

```
app.get('/example/a',function(req, res){
  res.send('Hello from A!');
});
```

Один маршрут может обрабатываться несколькими функциями обратного вызова (обязательно укажите объект next). Например:

```
app.get('/example/b',function(req, res, next){
  console.log('the response will be sent by the next function ...');
next();
},function(req, res){
  res.send('Hello from B!');
});
```

Массив функций обратного вызова может обрабатывать один маршрут. Например:

```
var cb0 =function(req, res, next){
  console.log('CB0');
next();
}
var cb1 =function(req, res, next){
  console.log('CB1');
next();
}
var cb2 =function(req, res){
  res.send('Hello from C!');
}
app.get('/example/c',[cb0, cb1, cb2]);
```

Маршрут может обрабатываться сочетанием независимых функций и массивов функций. Например:

```
var cb0 =function(req, res, next){
  console.log('CB0');
next();
}
var cb1 =function(req, res, next){
  console.log('CB1');
next();
}
app.get('/example/d',[cb0, cb1],function(req, res, next){
  console.log('the response will be sent by the next function ...');
next();
},function(req, res){
  res.send('Hello from D!');
});
```

Методы ответа

Методы в объекте ответа (res), перечисленные в таблице ниже, могут передавать ответ клиенту и завершать цикл “запрос-ответ”. Если ни один из этих методов не будет вызван из обработчика маршрута, клиентский запрос зависнет.

Метод	Описание
res.download()	Приглашение загрузки файла.
res.end()	Завершение процесса ответа.
res.json()	Отправка ответа JSON
res.jsonp()	Отправка ответа JSON с поддержкой JSONP.
res.redirect()	Перенаправление ответа.
res.render()	Вывод шаблона представления.
res.send()	Отправка ответа различных типов.
res.sendFile	Отправка файла в виде потока октетов.
res.sendStatus()	Установка кода состояния ответа и отправка представления в виде строки в качестве тела ответа.

app.route()

Метод app.route() позволяет создавать обработчики маршрутов, образующие цепочки, для пути маршрута. Поскольку путь указан в одном расположении, удобно создавать модульные маршруты, чтобы минимизировать избыточность и количество опечаток. Дополнительная информация о маршрутах приводится в документации Router().

Ниже приведён пример объединенных в цепочку обработчиков маршрутов, определённых с помощью функции app.route().

```
app.route('/book')
.get(function(req, res){
   res.send('Get a random book');
})
.post(function(req, res){
   res.send('Add a book');
})
.put(function(req, res){
   res.send('Update the book');
});
```

Вложенные маршруты

```
//React JSX
ReactDOM.render((
<Router>
<Route component={MainLayout}>
<Route component={SearchLayout}>
<Route path="users"component={UserList}/>
</Route>
</Route>
</Router>
),document.getElementById('root'));
```

Компоненты будут вложены друг в друга соответственно тому, как вложены маршруты. Когда пользователь переходит по адресу /users, роутер размещает компонент UserList внутри SearchLayout, а SearchLayout внутри MainLayout. В итоге при переходе /users будут выведены три вложенных компонента внутри 'root'.

JSX следует правилам XML в том плане, что компонент Route может быть записан как один самозакрывающийся тег:<Route/> или как два тега <Route>...</Route>. Это касается всего JSX, включая пользовательские компоненты и нормальные узлы DOM. Например, запись это валидный JSX, который будет рендериться в стандартный .

Так как у <Route component={SearchLayout}> есть два дочерних маршрута, пользователь может посетить /users или /widgets и соответствующий <Route> загрузит нужный компонент внутрь компонента SearchLayout.

Также обратите внимание, как компонент Home размещается непосредственно внутри MainLayout без задействования SearchLayout. Вы можете представить насколько легко можно изменять вложенность раскладок и компонентов, переставляя маршруты.

Параметры роутинга

URL является важной частью веб-приложений. Изначально задумывался как простой указатель на файл, который лежит на сервере, но с появлением веб-приложений лучше думать о нём как о текущем состоянии приложения. Глядя на него, пользователь может понять, где он в настоящее время находится, а также может скопировать его для последующего использования или кому-то передать.

Маршруты с параметрами

В рассмотренном в первом пункте примере была реализована маршрутизация. Расширяем его извлекая данные из GitHub.

Repos.js

```
import React,{Component}from'react';
import'whatwg-fetch';
classReposextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            repositories:[]
        };
    }
    componentDidMount(){
        fetch('https://api.github.com/users/pro-react/repos')
            .then((response)=>response.json())
            .then((responseData)=>{
                this.setState({repositories:responseData});
            });
    }
    render(){
        let repos=this.state.repositories.map((repo)=>(
            <li key={repo.id}>{repo.name}</li>
        ));
        return(
            <div>
                <h1>GithubRepos</h1>
                <ul>
                    {repos}
                </ul>
            </div>
        );
    }
}
export defaultRepos;
```

Теперь, если попробовать зайти на страницу Repos, мы увидим список репозиториев. Далее нам неплохо бы добавить для каждого элемента списка репозиториев ссылку на подробности, имеющую примерно такой вид /repos/details/repo_name. Создадим для этих целей новый компонент RepoDetails, но перед этим давайте обновим список добавим Link:

Repos.js

```
import React,{Component}from'react';
import'whatwg-fetch';
import{Link}from'react-router';
classReposextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            repositories:[]
        };
    }
    componentDidMount(){
        fetch('https://api.github.com/users/pro-react/repos')
            .then((response)=>response.json())
            .then((responseData)=>{
                this.setState({repositories:responseData});
            });
    }
    render(){
        let repos=this.state.repositories.map((repo)=>(
            <li key={repo.id}>
                <Link to={'/repos/details/'+repo.name}>{repo.name}</Link>
            </li>
        ));
        return(
            <div>
                <h1>GithubRepos</h1>
                <ul>
                    {repos}
                    {this.props.children}
                </ul>
            </div>
        );
    }
}
export defaultRepos;
```

При создании RepoDetails, есть две особенности:

- React Router будет передавать параметр URL через реквизит params значения параметров URL в нашем случае repo_name. Мы может использовать его для получения деталей о репозитории.
- Во вторых мы будем обновлять данные о репозитории не только при создании компонента (componentDidMount), но и при изменении реквизитов (componentWillReceiveProps).

RepoDetails.js

```
import React,{Component}from'react';
import'whatwg-fetch';
classRepoDetailsextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            repository:{}
        };
    }
    fetchData(repo_name){
        fetch('https://api.github.com/repos/pro-react/'+repo_name)
            .then((response)=>response.json())
            .then((responseData)=>{
                this.setState({repository:responseData});
            });
    }
    componentDidMount(){
        // The Router injects the key "repo_name" inside the params prop
        let repo_name=this.props.params.repo_name;
        this.fetchData(repo_name)
    }
    componentWillReceiveProps(nextProps){
        // The Router injects the key "repo_name" inside the params prop
        let repo_name=nextProps.params.repo_name;
        this.fetchData(repo_name)
    }
    render(){
        let stars=[];
        for(vari=0;i<this.state.repository.stargazers_count;i++){
            stars.push('★');
        }
        return(
            <div>
                <h2>{this.state.repository.name}</h2>
                <p>{this.state.repository.description}</p>
                <span>{stars}</span>
            </div>
        );
    }
}
export defaultRepoDetails;
```

Для завершения надо RepoDerails добавить в App.js:

```
import React,{Component}from'react';
import{render}from'react-dom';
import{Router,Route,Link,IndexRoute}from'react-router';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
import RepoDetails from'./RepoDetails';
classAppextendsComponent{...}
render((
  <Router>
    <Route path="/"component={App}>
      <IndexRoute component={Home}/>
      <Route path="about"component={About}/>
      <Route path="repos"component={Repos}>
        {/* Add the route, nested where we want the UI to nest */}
        <Route path="details/:repo_name"component={RepoDetails}/>
      </Route>
    </Route>
  </Router>
),document.getElementById('root'));
```

Здесь импортированы RepoDetails и добавили в Route Repos новый вложенный Route с шаблоном url details/:repo_name и компонентом RepoDetails.

Практика

Задача 1.Реализация маршрутизации

Создадим приложение, которое будет использовать GitHub API, получения списка репозиториев. Помимо этого добавим домашнюю страницу и «О проекте». Давайте начнем с главного компонента App:

App.js

```
import React,{Component}from'react';
import{render}from'react-dom';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
classAppextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            route:window.location.hash.substr(1)
        };
    }
     componentDidMount(){
        window.addEventListener('hashchange',()=>{
            this.setState({
                route:window.location.hash.substr(1)
            });
        });
    }
    render(){
        return(
            <h1>Hello World</h1>
        );
    }
}
render(<App/>,document.getElementById('root'));
```

Код довольно прост. В конструкторе компонента, получаем текущее значение хеш URL-адрес и присваиваем состояние маршрута. Затем, при создании компонента, добавляем обработчик события на изменение URL, в нём обновляется текущее состояние нашего компонента. Теперь нам надо обновить метод render:

App.js

```
...
   render(){
      varChild;
      switch(this.state.route){
         case'/about':
            Child=About;
            break;
         case'/repos':
            Child=Repos;
            break;
         default:
            Child=Home;
      }
      return(
         <div>
            <header>App</header>
            <menu>
               <ul>
                  <li><ahref="#/about">About</a></li>
                  <li><ahref="#/repos">Repos</a></li>
               </ul>
            </menu>
            <Child/>
         </div>
      )
   }
...
```

Проверяется текущее состояние route, и в зависимости от него Child присваивается определённый компонент. Все дочерние компоненты, которые представляют собой внутренние страницы приложения, просты и имеют одинаковую структуру:

Home.js

```
import React,{Component}from'react';
classHomeextendsComponent{
   render(){
      return(
         <h1>HOME</h1>
      );
   }
}
export defaultHome;
```

About.js

```
import React,{Component}from'react';
classAboutextendsComponent{
    render(){
        return(
            <h1>ABOUT</h1>
        );
    }
}
export defaultAbout;
```

Repos.js

```
import React,{Component}from'react';
classReposextendsComponent{
    render(){
        return(
            <h1>Github Repos</h1>
        );
    }
}
export defaultRepos;
```

Задача 2. Подключение React Router

Установка с npm:

```
npm install —save react-router
```

Добавим React Router:

App.js

```
import React,{Component}from'react';
import{render}from'react-dom';
import{Router,Route,Link}from'react-router';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
classAppextendsComponent{
   render(){
      return(
         <div>
            <header>App</header>
            <menu>
               <ul>
                  <li><Link to="/about">About</Link></li>
                  <li><Link to="/repos">Repos</Link></li>
               </ul>
            </menu>
            {this.props.children}
         </div>
      );
   }
}
render(<App/>,document.getElementById('root'));
```

Здесь импортировали React Router и обновили метод render добавив туда Link. Обновляем метод render:

App.js

```
import React,{Component}from'react';
import{render}from'react-dom';
import{Router,Route,Link}from'react-router';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
classAppextendsComponent{
   render(){
      return(
         <div>
            <header>App</header>
            <menu>
               <ul>
                  <li><Link to="/about">About</Link></li>
                  <li><Link to="/repos">Repos</Link></li>
               </ul>
            </menu>
            {this.props.children}
         </div>
      );
   }
}
render((
   <Router>
      <Route path="/"component={App}>
         <Route path="about"component={About}/>
         <Route path="repos"component={Repos}/>
      </Route>
   </Router>
),document.getElementById('root'));
```

Если попробовать запустить, то видно, что всё работает как раньше, но если зайти «/», то не увидим результат компонента «Home». Указываем путь.

```
...
<Router>
  <Route path="/"component={App}>
    <Route path="???"component={Home}/>
    <Route path="about"component={About}/>
    <Route path="repos"component={Repos}/>
  </Route>
</Router>
...
```

Задача 3. Маршруты с параметрами

В рассмотренном в первом примере была реализована маршрутизация. Расширяем его извлекая данные из GitHub.

Repos.js

```
import React,{Component}from'react';
import'whatwg-fetch';
classReposextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            repositories:[]
        };
    }
    componentDidMount(){
        fetch('https://api.github.com/users/pro-react/repos')
            .then((response)=>response.json())
            .then((responseData)=>{
                this.setState({repositories:responseData});
            });
    }
    render(){
        let repos=this.state.repositories.map((repo)=>(
            <li key={repo.id}>{repo.name}</li>
        ));
        return(
            <div>
                <h1>GithubRepos</h1>
                <ul>
                    {repos}
                </ul>
            </div>
        );
    }
}
export defaultRepos;
```

Теперь, если попробовать зайти на страницу Repos, мы увидим список репозиториев. Далее нам неплохо бы добавить для каждого элемента списка репозиториев ссылку на подробности, имеющую примерно такой вид /repos/details/repo_name. Создадим для этих целей новый компонент RepoDetails, но перед этим давайте обновим список добавим Link:

Repos.js

```
import React,{Component}from'react';
import'whatwg-fetch';
import{Link}from'react-router';
classReposextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            repositories:[]
        };
    }
    componentDidMount(){
        fetch('https://api.github.com/users/pro-react/repos')
            .then((response)=>response.json())
            .then((responseData)=>{
                this.setState({repositories:responseData});
            });
    }
    render(){
        let repos=this.state.repositories.map((repo)=>(
            <li key={repo.id}>
                <Link to={'/repos/details/'+repo.name}>{repo.name}</Link>
            </li>
        ));
        return(
            <div>
                <h1>GithubRepos</h1>
                <ul>
                    {repos}
                    {this.props.children}
                </ul>
            </div>
        );
    }
}
export defaultRepos;
```

RepoDetails.js

```
import React,{Component}from'react';
import'whatwg-fetch';
classRepoDetailsextendsComponent{
    constructor(){
        super(...arguments);
        this.state={
            repository:{}
        };
    }
    fetchData(repo_name){
        fetch('https://api.github.com/repos/pro-react/'+repo_name)
            .then((response)=>response.json())
            .then((responseData)=>{
                this.setState({repository:responseData});
            });
    }
    componentDidMount(){
        // The Router injects the key "repo_name" inside the params prop
        let repo_name=this.props.params.repo_name;
        this.fetchData(repo_name)
    }
    componentWillReceiveProps(nextProps){
        // The Router injects the key "repo_name" inside the params prop
        let repo_name=nextProps.params.repo_name;
        this.fetchData(repo_name)
    }
    render(){
        let stars=[];
        for(vari=0;i<this.state.repository.stargazers_count;i++){
            stars.push('★');
        }
        return(
            <div>
                <h2>{this.state.repository.name}</h2>
                <p>{this.state.repository.description}</p>
                <span>{stars}</span>
            </div>
        );
    }
}
export defaultRepoDetails;
```

Для завершения надо RepoDerails добавить в App.js:

```
import React,{Component}from'react';
import{render}from'react-dom';
import{Router,Route,Link,IndexRoute}from'react-router';
import About from'./About';
import Home from'./Home';
import Repos from'./Repos';
import RepoDetails from'./RepoDetails';
classAppextendsComponent{...}
render((
  <Router>
    <Route path="/"component={App}>
      <IndexRoute component={Home}/>
      <Route path="about"component={About}/>
      <Route path="repos"component={Repos}>
        {/* Add the route, nested where we want the UI to nest */}
        <Route path="details/:repo_name"component={RepoDetails}/>
      </Route>
    </Route>
  </Router>
),document.getElementById('root'));
```

Здесь импортированы RepoDetails идобавилив Route Repos новый вложенный Route с шаблоном url details/:repo_name и компонентом RepoDetails.

Домашнее задание

1. Создать хранилище (store), где будут находить все записи нашего блога. При этом, для всех записей блога должна быть возможность: создать, удалить, отредактировать эти блоги.
2. Для каждого метода хранилища, который является методом действия, необходимо создать соответствующий Action либо в виде функции, либо в виде класса. В итоге должен получиться файл, который либо экспортирует набор функций, либо целый класс.
3. Для всех видов действий необходимо создать соответствующие константы, которые будут описывать то действие, которое будет совершаться. Опять же, необходимо положить их в отдельный файл.
4. Последним шагом необходимо в конечном компоненте импортировать хранилище и действия и прописать их использование в необходимых местах.
5. *При этом также следует создать обратную связь между изменением состояния в хранилище и актуальным состоянием в компоненте (мы это делали через EventEmitter).
6. * Реализовать это же задание с применением Redux библиотеки.

Задачи со * повышенной сложности

Дополнительные материалы

1. https://habrahabr.ru/post/249279/
2. https://facebook.github.io/flux/docs/overview.html
3. https://github.com/reactjs/react-redux
4. https://habrahabr.ru/post/269831/
5. http://redux.js.org/docs/basics/UsageWithReact.html

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://habrahabr.ru
2. https://facebook.github.io
3.