# Часть 2. Погружаемся в React

## Задание 1. Работа с вложенными элементами React

1. Хорошо, это работает, и мы находимся на половине пути. Теперь представьте себе, что вам нужно разработать множество таблиц, подобных этой, и вам нужно отображать сводные данные в нижней части таблицы с различной информацией - например, сколько сейчас активных пользователей и сколько пользователей вообще. Что вы будете делать? В этом случае имеет смысл использовать потомков (children) из props:

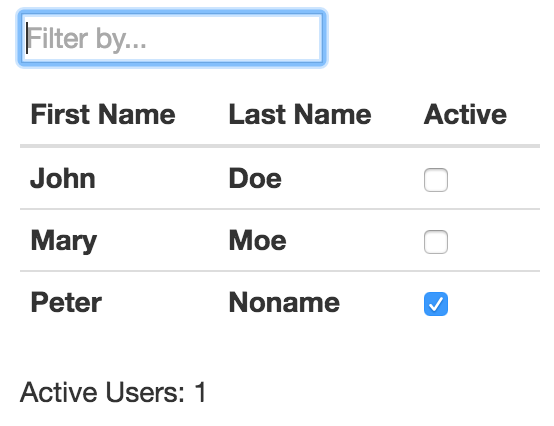
**class** SummaryActive **extends** React.Component {  
 render(){  
 **return** (  
 <div>Active Users: {**this**.props.records.filter((record)=>record.active).length}</div>  
 )  
 }  
}  
  
**class** SummaryUsers **extends** React.Component {  
 render(){  
 **return** (  
 <div>Users Count: {**this**.props.records.length}</div>  
 )  
 }  
}

render(  
 <GridComponent>  
 <SummaryActive/>  
 </GridComponent>,  
 document.getElementById('app')  
);

И добавьте вывод SummaryActive, которые доступны как this.props.children, после вывода таблицы в GridComponent:

</table>  
 <div>{**this**.props.children &&

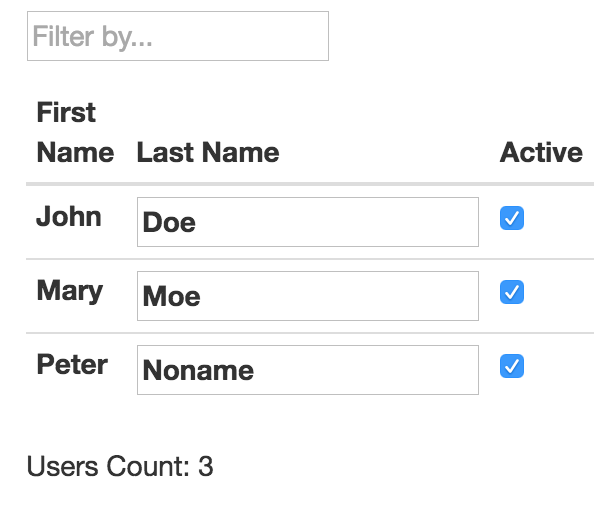
React.cloneElement(**this**.props.children, { records: **this**.state.records })}</div>  
 </div>  
)



1. Теперь допустим в какой-то другой таблице необходимо отображать общее количество пользователей. Это можно сделать следующим образом:

render(  
 <GridComponent>  
 <SummaryUsers/>  
 </GridComponent>,  
 document.getElementById('app')  
);

1. Никаких других изменений в GridComponent не требуется. Это очень удобно, поскольку теперь GridComponent полностью отвязан от информации, выводимой в нижней части таблицы.



1. Теперь давайте наведем порядок в app.js. Во-первых, давайте переместим SummaryActive и SummaryUsers в отдельный файл.Создадим summaries.js и поместим их туда.
2. Теперь создадим grid.js и поместим GridComponent и GridRecord, а также dataSource в этот файл.
3. Теперь, если вы перезагрузите приложение, вы увидите следующее:



1. Не беспокойтесь, это ожидаемо. Мы перенесли часть нашего приложения в отдельный файл, но не связали app.js с этим файлом. Давайте сделаем это. Нам нужно добавить React во все файлы, которые мы создаем:

**import** React **from** 'react';

1. Мы добавим export в компоненты, которые мы планируем использовать снаружи:

**export class** SummaryActive **extends** React.Component {

**export class** SummaryUsers **extends** React.Component {

**export default class** GridComponent **extends** React.Component {

1. Также добавим import в app.js, чтобы у нас была возможность использовать GridComponent и Summaries

**import** GridComponent **from** './grid';  
**import** {SummaryActive, SummaryUsers} **from** './summaries';

1. После всех этих изменений app.js будет выглядеть так:

require("bootstrap/dist/css/bootstrap.css");  
**import** React **from** 'react';  
**import** {render} **from** 'react-dom';  
**import** GridComponent **from** './grid';  
**import** {SummaryActive, SummaryUsers} **from** './summaries';  
  
  
render(  
 <GridComponent>  
 <SummaryUsers/>  
 </GridComponent>,  
 document.getElementById('app')  
);

1. Очень ясное и структурированное приложение.

#### Задача 4: Создайте компонент UserDetail, который должен отображать детали пользователя

1. Создайте файл user-details.js в папке js
2. Создайте компонент react, чтобы рендерить детали пользователя, взяв за основу HTML, приведенный ниже:

<div class="container">  
 <div class="row">  
 <div class="col-md-offset-2 col-md-8 col-lg-offset-3 col-lg-6">  
 <div class="well profile">  
 <div class="col-sm-12">  
 <div class="col-xs-12 col-sm-8">  
 <h2>Nicole Pearson</h2>  
 <p><strong>About: </strong> Web Designer / UI. </p>  
 <p><strong>Hobbies: </strong> Read, out with friends, listen to music, draw and learn new things. </p>  
 <p><strong>Skills: </strong>  
 <span>html5</span>  
 <span>css3</span>  
 <span>jquery</span>  
 <span>bootstrap3</span>  
 </p>  
 </div>  
 <div class="col-xs-12 col-sm-4 text-center">  
 <figure>  
 <img src="http://www.bitrebels.com/wp-content/uploads/2011/02/Original-Facebook-Geek-Profile-Avatar-2.jpg" alt="" class="img-circle img-responsive"/>  
 </figure>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
</div>

Компонент должен показывать данные из этой переменной:

var user = {name:"John Doe",

about:"Nice guy",

hobby:"Likes drinking wine",

skills:["html", "javascript", "redux"]}

1. Отрендерите этот компонент, поместив его на место GridComponent (временно)

#### Задача 5: Создайте компонент UserDetails для отображения детальных данных всех пользователей.

Мы разработали компонент UserDetail, отображающий тестовые данные. Мы будем использовать компонент UserDetails чтобы отобразить всю детальную информацию о каждом пользователе.

1. UserDetails компонент отображает данные из объекта detailsRecords.
2. UserDetails компонент может отображать данные по любому числу пользователей
3. В качестве исходных данных возьмите следующие:

detailsRecords = [{  
 id:1,  
 name:"John Doe",  
 about:"Nice guy",  
 hobby:"Likes drinking wine",  
 skills:["html", "javascript", "redux"]  
},{  
 id:2,  
 name:"Mary Moe",  
 about:"Cute girl",  
 hobby:"Likes playing xbox whole days long",  
 skills:["Fortran", "Lua", "R#"]  
}];

## Задание 2. React Router 4

1) Установите React Router командой

**npm install react-router-dom**

2) Импортируйте необходимые классы из react-router-dom:

**import** {**HashRouter**, **Switch**, **Route**, **Link**} **from "react-router-dom"**;

3) Используйте такую конфигурацию для роутера:

<**HashRouter**>  
 <**div**>  
 <**App** />  
 <**Switch**>  
 <**Route path="/grid" component=**{*GridComponent*}/>  
 <**Route exact path="/details" component=**{UserDetails}/>  
 <**Route path="/details/:id" component=**{UserDetails}/>  
 </**Switch**>  
 </**div**>  
</**HashRouter**>

Обратите внимание, что ссылки <Link> могут быть использованы только внутри роутера (тега <HashRouter> или <BrowserRouter>).

4) Определите компонент App – это будет главное меню приложения:

**const** *App* = ({children}) =>  
 <**div**>  
 <**h1**>Our awesome app</**h1**>  
 <**ul role="nav"**>  
 <**li**><**Link to="/grid"**>Grid</**Link**></**li**>  
 <**li**><**Link to="/details"**>Details</**Link**></**li**>  
 </**ul**>  
 {children}  
 </**div**>;

5) Обновите dataSource путем добавления ids:

**const** dataSource = [  
 {firstName: "John", lastName: "Doe", active: **false**, id: 1},  
 {firstName: "Mary", lastName: "Moe", active: **false**, id: 2},  
 {firstName: "Peter", lastName: "Noname", active: **true**, id: 3}  
];

Также обновите таблицу в GridComponent:

<thead>  
<tr>  
 <th>Id</th>  
 <th>First Name</th>  
 <th>Last Name</th>  
 <th>Active</th>  
</tr>  
</thead>

6) Определите метод в компоненте UserDetails, который будет фильтровать пользователей по id, оставляя только детали пользователей с id из URL параметра или отображая детали всех пользователей:

filterDetails(id) {  
 **if** (id) {  
 **this**.setState({  
 **details**: detailsRecords.filter((record) => {  
 **return** record.id == id;  
 })  
 })  
 } **else** {  
 **this**.setState({  
 **details**: detailsRecords  
 })  
 }  
}

7) Нам необходимо фильтровать список пользователей при монтировании компонента на страницу таким образом, чтобы UserDetails отображала только информацию по определенному id:

componentWillMount() {  
 **this**.filterDetails(**this**.props.match.**params**.id);  
}

8) Также обратите внимание, что компонент UserDetails не пересоздается, когда мы идем с URL “/details” на “/details/:id” и назад – он только обновляется. Метод componentWillMount() не будет запущен в этом случае. Также мы не можем использовать componentWillUpdate() потому что в нем нельзя использовать setState() во избежание зацикливания. Чтобы получить id и перезапустить фильтр в этом случае, мы будем использовать метод componentWillReceiveProps(), который не перезапускается, когда мы вызываем setState(), поэтому setState() безопасно вызывать из этого метода. Мы можем сделать это следующим образом:

componentWillReceiveProps(props) {  
 **this**.filterDetails(props.match.**params**.id);  
}

9) Также вам понадобится объект history, чтобы поместить обновленный URL, когда пользователь щелкает по ссылке. К объекту history можно обратиться в любом компоненте, использующем роутер, через this.props.history. Таким образом, history доступен в GridComponent. Нам необходимо передать его в GridRecord:

**<GridRecord record=**{record} **history=**{**this**.props.history} … />

Затем в компоненте GridRecord необходимо обновить метода showUserDetails:

showUserDetails(e){  
 e.preventDefault();  
 **this**.props.history.push(**`/details/**${**this**.props.record.id}**`**);  
}

Добавьте данную колонку как первую в GridRecord:

<th onClick={**this**.showUserDetails.bind(**this**)}><a href="#">{record.id}</a></th>

10) Теперь вы можете проверить, как работает роутер в вашем приложении.

#### Дополнительное задание

Материал этого модуля сравнительно небольшой, но у вас есть отличная возможность в оставшееся время попрактиковаться в самостоятельном программировании под React.

* Реализуйте возможность редактирования записей, причем это должно делаться в отдельном роуте.
* Переход в окно редактирования должен производиться при нажатии по имени или фамилии.
* При попытке вернуться назад с несохраненными данными спрашивайте подтверждение.
* Также должна быть возможность перемещаться к редактированию следующей/предыдущей записи c поддержкой роутинга (добавьте на форму кнопки Next/Prev).

## Задание 3. Изучаем Flux

Возьмите за основу проект FLUX-initial.zip.

Разберитесь, как он работает.

**Задача: реализовать возможность удалять значения из списка**

Удаленные значения должны показываться как зачеркнутые.

Используйте Store и Dispatcher для этих операций.



Постарайтесь решить эту задачу самостоятельно. Для тех, у кого не получается, ниже приведено пошаговое решение.

**Решение**

1) Создайте кнопку **x** в MyList:

**const** *MyList* = ({items}) => (  
 <**ul**>  
 {items.map(i => (  
 <**li key=**{i}>{i} <**button onClick=**{()=>*remove*(i)}>x</**button**></**li**>  
 ))}  
 </**ul**>  
);

2) Реализуйте функцию remove(val):

**function** *remove*(value) {  
 *removeFromList*(value);  
}

3) Реализуйте removeFromList() в actions:

**export function** *removeFromList*( value ) {  
 **let** payload = {  
 **type**: **'REMOVE\_FROM\_LIST'**,  
 **value**: value,  
 }  
 appDispatcher.dispatch(payload);  
}

4) Добавьте дополнительное значение в начальный Store в filteredList.js:

*setState*(*getState*().merge({  
 **placeholder**: **'Search...'**,  
 **items**: [**'First'**, **'Second'**, **'Third'**, **'Fourth'**],  
 **tempItems**: [],  
 **removedItems**: [],  
}));

5) Зарегистрируйте функцию dispatcher для выполнения действий в Store:

appDispatcher.register(**function**(payload) {  
  
 **switch**(payload.**type**) {  
  
 **case "REMOVE\_FROM\_LIST"**:  
 **let** state = *getState*();  
 **let** removedItems=state.get(**"removedItems"**);  
 *// Updates the state of the store.  
 setState*(state.merge({  
 **removedItems**: removedItems.push(payload.value)  
 }));  
 **break**;  
  
 **default**:  
 }  
});

6) Добавьте стиль style.css:

.**removed** {  
 **text-decoration**: **line-through**;  
}

7) Подсоедините стиль к MyList.js:

**import "./style.css"**;

8) Реализуйте функцию getRemovedClass(value) которая должна возвращать класс «removed» в случае, если значение было удалено:

**function** *getRemovedClass*(value) {  
 **if** (*getState*().get(**"removedItems"**).find(e=>e==value)) {  
 **return "removed"**;  
 } **else return ""**;  
}

9) Используйте getRemovedClass() в компоненте MyList:

**const** *MyList* = ({items}) => (  
 <**ul**>  
 {items.map(i => (  
 <**li className=**{*getRemovedClass*(i)} **key=**{i}>{i}  
 <**button onClick=**{()=>*remove*(i)}>x</**button**></**li**>  
 ))}  
 </**ul**>  
);

10) Перезагрузите страницу и проверьте результат.