# Часть 1. Введение в React

## Задание 1. Основы React

### Начните новый проект

1. Mkdir project && cd project
2. Запустите **npm init** и ответьте на вопросы
3. Перейдите в терминал и установите react и react-dom:

**npm i react react-dom**

### Привет, Мир! на React

1. Создайте страницу **index.html** с помощью:

<!doctype html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>  
<body>  
  
</body>  
</html>

1. Добавьте скрипты react и react-dom и создайте dom элемент, куда будет помещен результат:

<!doctype html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <script src="node\_modules/react/umd/react.development.js"></script>  
 <script src="node\_modules/react-dom/umd/react-dom.development.js"></script>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>  
<body>  
 <app id="app">  
   
 </app>

<script src="js/app.js"></script>

</body>  
</html>

1. Создайте папку js и добавьте туда файл app.js (mkdir js && cd js)
2. Добавьте следующий контент в app.js:

ReactDOM.render(  
 React.createElement('h1', **null**, 'Hello, world!'),  
 document.getElementById('app')  
);

1. Откройте index.html в браузере. Ура! Мы только что создали React компонент, используя чистый JavaScript: 
2. Окей, хорошо, но давайте сделаем что-то посложнее и создадим простую разметку, например div > h1 ul > li li > h2:

**var** app = React.createElement('div', {},  
 React.createElement('h1', {}, "Hi, I'm header inside div"),  
 React.createElement('ul', {},  
 React.createElement('li', {},  
 React.createElement('h2', {}, "Hi, I'm list item inside list inside div")  
 ),  
 React.createElement('li', {},  
 React.createElement('h2', {}, "Hi, I'm list item inside list inside div")  
 )  
 )  
);  
  
ReactDOM.render(  
 app,  
 document.getElementById('app')  
);

1. Что здесь происходит? Выглядит ли такой код нормально? Как вы считаете, комфортно ли читать такой код? – Нет, нет, нет!
2. Давайте начнем использовать jsx. Для этого нам потребуется WebPack и преобразователь JSX:

npm i webpack -g

npm i webpack-cli -g

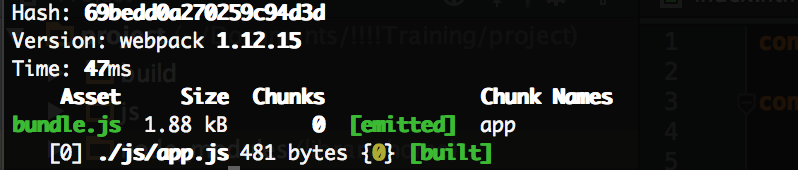
npm i -D webpack

1. Давайте я объясню, что мы сделали. Мы глобально установили webpack globally (флаг -g) чтобы быть уверенными, что он доступен везде (вместо этого мы могли каждый раз обращаться к node\_modules/webpack/.bin/xxx). Зачем это нужно? Давайте пойдем шаг за шагом:
   1. **Webpack**: Упаковщик модулей, также использующий преобразование JSX в JS, а ES6 в ES2015+
   2. **Babel**-**core:** собственно JSX -> JS, ES6 -> ES5
   3. **Webpack Babel loader:** Используется webpack чтобы управлять babel
   4. **Babel Presets:** Различные настройки для разного рода компиляции, которая нам может понадобиться. Мы будем использовать настройку React (JSX -> JS) и ES2015 (ES6-> ES5)
2. Давайте теперь настроим конфигурацию webpack. Создайте файл webpack.config.js в корне вашего проекта и поместите туда код:

**const** path = require('path');  
  
**const** PATHS = {  
 app: "./js/app",  
 build: path.join(\_\_dirname, 'build')  
};  
  
module.exports = {  
 mode: 'development',  
 entry: {  
 app: PATHS.app  
 },  
 output: {  
 path: PATHS.build,  
 filename: 'bundle.js'  
 }  
};

1. Мы говорим webpack, чтобы он произвел сборку с нашей точки входа и назвал выходной файл bundle.js. Хорошо. У нас пока нет файла bundle.js, но предположим, что мы его получили. Обновите наш index.html

<!doctype html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <script src="node\_modules/react/umd/react.development.js"></script>  
 <script src="node\_modules/react-dom/umd/react-dom.development.js"></script>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>  
<body>  
 <app id="app">  
  
 </app>  
   
<script src="build/bundle.js"></script>  
</body>  
</html>

1. Хорошо, похоже, мы готовы собрать файл. Введите webpack в корне вашего проекта. Вы увидите что-то вроде этого:
2. Откройте index.html в браузере. Ничего не изменилось? Потрясающие! Это именно то, что мы искали.

#### Задача 1: Создать компонент React.js с заданным HTML

## Создайте такую структуру html, используя чистый JS и React:

<div>  
 <h1>I'm page header</h1>  
 <div>  
 <p>I'm staying at the begining of the page content</p>  
 <div>  
 <span>I'm user logo container</span>  
 <span>I'm user name container</span>  
 </div>  
 <h2>I'm next section header</h2>  
 <section>  
 <article>I'm awesome article</article>  
 <ul>  
 <li>I'm article item</li>  
 <li>I'm article item with <b>bold element</b></li>  
 </ul>  
 </section>  
 </div>  
</div>

Не забудьте заново сделать build после внесения изменений!

1. Думаю, когда вы набрали webpack 10 раз, чтобы пересобрать bundle, вы устали от него. Вы можете упростить этот процесс: запускайте webpack так:

webpack -w

## Задание 2. JSX

Насколько вам комфортно с React.js разметкой? Это выглядит странно? Да, я так думаю. Давайте заменим его на JSX, чтобы сделать более понятным:

class JsxApp extends React.Component {

render(){

return (

<div>

<div>Hi, I'm div</div>

<h1>Hi, I'm header inside div</h1>

<ul>

<li>Hi, I'm list item inside list inside div</li>

<li>Hi, I'm list item inside list inside div</li>

</ul>

</div>

)

}

}

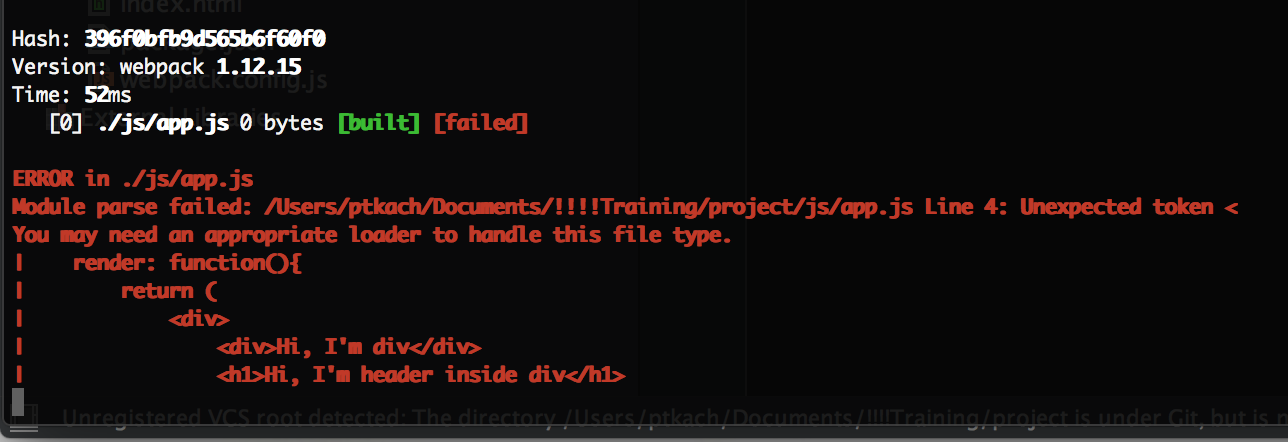
ReactDOM.render(

<JsxApp></JsxApp>,

document.getElementById('app')

);

1. Взгляните в консоль. Вы должны там увидеть следующее:



1. Похоже, нам нужен загрузчик, чтобы позволить webpack работать с JSX. Как вы помните, мы его уже устанавливали, осталось его настроить. Пойдем на webpack.config.js и поместим туда такой код:

**const** path = require('path');  
  
**const** PATHS = {  
 app: "./js/app",  
 build: path.join(\_\_dirname, 'build')  
};  
  
module.exports = {  
 devtool: 'source-map',

mode: 'development',   
 entry: {  
 app: PATHS.app  
 },  
 module: {  
 rules: [  
 {  
 test: /\.js$/,  
 exclude: /node\_modules/,  
 loader: 'babel-loader',  
 query: {  
 presets: ['@babel/preset-react']  
 }  
 }  
 ]  
 },  
 output: {  
 path: PATHS.build,  
 filename: 'bundle.js'  
 }  
};

Также вам надо установить Babel и Babel loader:

npm i @babel/core babel-loader @babel/preset-env @babel/preset-react --save-dev

1. Давайте я объясню, что мы проделали. Мы добавили 2 вещи:
   1. Devtool: мы заставили webpack строить полный source-map за нас (Просто в случае, если нам понадобится воспользоваться отладчиком)
   2. Module: Это самое интересное. Мы попросили webpack использовать загрузчик babel (установленный нами ранее) с настройкой react (также установленной ранее) для любых файлов \*.js, исключая папку node\_modules. Почему нам необходимо исключить node\_modules? Потому что там хранятся библиотечные файлы, которые уже обработаны, и нам нет необходимости их заново компилировать и собирать.
2. Теперь перезапустим webpack и откроем страницу index.html в браузере:



1. Отлично, работает. Таким образом, мы настроили webpack на использование JSX с загрузчиком Babel, мы заменили JS на JSX (и это выглядит намного лучше) и преобразовали JSX в JS, используя webpack.
2. Но что здесь не так? Я думаю, раз мы используем webpack, который предоставлят нам все возможности работы с модулями, нам не нужно загружать React и ReactDOM как глобальные скрипты:

<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <script src="node\_modules/react/umd/react.development.js "></script>  
 <script src="node\_modules/react-dom/umd/react-dom.development.js"></script>  
</head>

1. Давайте удалим скрипты и будем загружать их через webpack.

Добавьте такие строчки в ваш app.js:

**import** React **from** 'react';  
**import** ReactDOM **from** 'react-dom';

1. Если вы посмотрите на сгенерированный код, вы увидите, что webpack сгенерировал код ES2015 (поддерживаемый любыми современными браузерами).
2. Если вы хотите, чтобы сгенерированный код поддерживался любым, даже старым бразуером, вам нужно добавить ‘es2015’ в presets:

presets: ['@babel/preset-env', '@babel/preset-react']

1. И вуаля:



1. Таким образом, мы соединили webpack, babel + babel es2015, react presets, модули react и reactDOM – и вся инфраструктура готова!

### 

1. Давайте сделаем что-то полезное. Мы создадим таблицу, используя стили Bootstrap. Для этого нам нужно установить bootstrap и webpack css loader (Да, webpack также может обрабатывать и css)

npm install --save bootstrap

npm install file-loader css-loader style-loader url-loader --save-dev

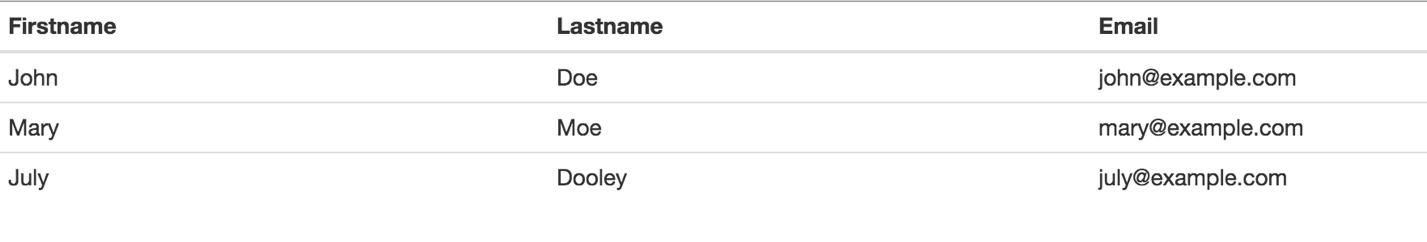
1. Обновите **webpack.config.js** , добавив в него новые загрузчики:

rules: [  
 {  
 test: /\.js$/,  
 exclude: /node\_modules/,  
 loader: 'babel-loader',  
 query: {  
 presets: ['@babel/preset-react']  
 }  
 },{  
 test: /\.css$/,  
 loaders: ['style-loader', 'css-loader']  
 },  
 { test: /\.(png|woff|woff2|eot|ttf|svg)$/, loader: 'url-loader?limit=100000' }  
]

1. Обновите ваш компонент:

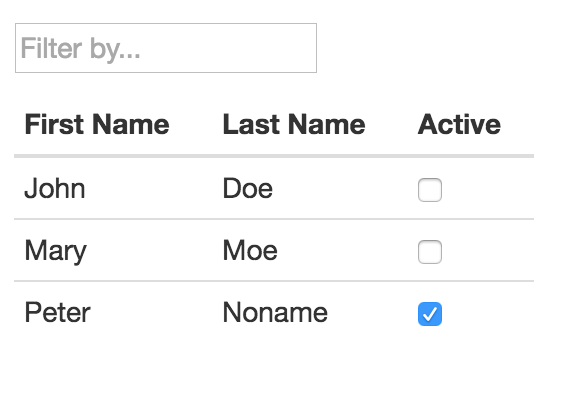
**import** "bootstrap/dist/css/bootstrap.css";  
**import** React **from** 'react';  
**import** {render} **from** 'react-dom';  
  
**class** GridComponent **extends** React.Component {  
 render() {  
 **return** (  
 <table className="table table-condensed">  
 <thead>  
 <tr>  
 <th>Firstname</th>  
 <th>Lastname</th>  
 <th>Email</th>  
 </tr>  
 </thead>  
 <tbody>  
 <tr>  
 <td>John</td>  
 <td>Doe</td>  
 <td>john@example.com</td>  
 </tr>  
 <tr>  
 <td>Mary</td>  
 <td>Moe</td>  
 <td>mary@example.com</td>  
 </tr>  
 <tr>  
 <td>July</td>  
 <td>Dooley</td>  
 <td>july@example.com</td>  
 </tr>  
 </tbody>  
 </table>  
 )  
 }  
}  
  
render(  
 <GridComponent/>,  
 document.getElementById('app')  
);

1. Откройте браузер:



1. Очень хорошо! Теперь у нас есть первый UI компонент, написанный на JSX, CSS стили из bootstrap загружаются через webpack – и все это магическим образом работает.

#### Задача 2: Создайте разметку JSX по данному дизайну:



## Задание 3. State и Props

1. Теперь давайте добавим state и props и удалим заданные значения в таблицах. Прежде всего, давайте создадим объект, эмулирующий источник данных. Вот так будет выглядеть app.js:

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

1. Давайте передадим эти данные как state в наш GridComponent:

constructor(){  
 **super**();  
 **this**.state = {  
 records:[]  
 }  
}  
componentDidMount(){  
 **this**.setState({  
 records:dataSource  
 })  
}

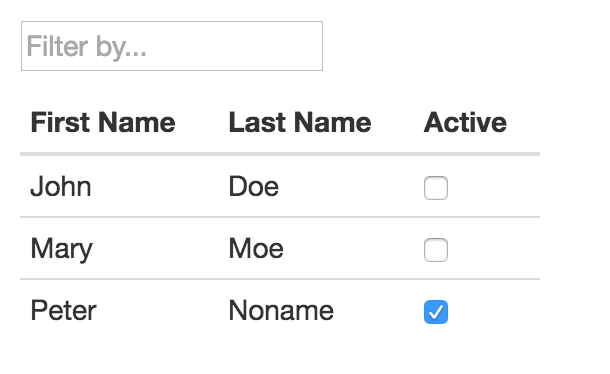
1. Отлично, они там. Теперь давайте запрограммируем, чтобы компонент отображал те props, которые он получает на входе. Для этого прежде всего извлечем Grid Record из компонента. И пусть GridRecord также получает необходимые данные из props:

**class** GridRecord **extends** React.Component {  
 render(){  
 **let** {record} = **this**.props;  
 **return** <tr>  
 <th>{record.firstName}</th>  
 <th>{record.lastName}</th>  
 <th><input type="checkbox" checked={record.active}/></th>  
 </tr>  
 }  
}

1. Теперь поместим этот код в метод render() в GridComponent, чтобы отрерисовать GridRecords вместо использования жестко запрограммированной разметки:

render(){  
 **let** records = **this**.state.records.map((record)=>{  
 **return** <GridRecord record={record}/>  
 });  
 return (  
 <div style={{width:300, height: 300, padding: 20}}>  
 <p>  
 <input type="text" placeholder="Filter by..."/>  
 </p>  
 <table className="table table-condensed">  
 <thead>  
 <tr>  
 <th>First Name</th>  
 <th>Last Name</th>  
 <th>Active</th>  
 </tr>  
 </thead>  
 <tbody>  
 {records}  
 </tbody>  
 </table>  
 </div>  
 )  
 }  
}

1. Окей, убедитесь, что вы понимаете, что мы здесь сделали. Мы передали данные в GridComponent, после чего пробежали по данным data и передали каждую запись record как свойство в компонент GridRecord, после чего просто добавили все записи в массив records, который мы отрендерили в компоненте GridComponent. Откройте код в браузере и убедитесь, что ничего не изменилось:



1. Что происходит на самом деле – мы разделили ответственности и улучшели масштабируемость нашего компонента. И вы можете видеть однонаправленный поток данных (Data Flow) в этом примере. Также этот код можно написать таким образом:

**return** (  
 <div style={{width:300, height: 300, padding: 20}}>  
 <p>  
 <input type="text" placeholder="Filter by..."/>  
 </p>  
 <table className="table table-condensed">  
 <thead>  
 <tr>  
 <th>First Name</th>  
 <th>Last Name</th>  
 <th>Active</th>  
 </tr>  
 </thead>  
 <tbody>  
 {**this**.state.records.map((record)=>{  
 **return** <GridRecord record={record}/>  
 })}  
 </tbody>  
 </table>

</div>  
 )

1. Теперь вы можете увидеть в консоли что-то вроде этого:

../../Desktop/Screen%20Shot%202016-04-14%20at%202.27.06%20PM.p

1. Это происходит потому, что для внутренних оптимизаций в React если вы создаете компоненты динамически, используя итераторы, вам необходимо передавать каждому компоненту какой-то уникальный идентификатор, используя атрибут key:

{**this**.state.records.map((record, index)=>{  
 **return** <GridRecord record={record} key={index}/>  
})}

1. Теперь нам надо дать пользователю возможность менять чекбокс при щелчке мышкой. Теоретически, первое, что приходит на ум – код вроде такого:

toggleActive(){  
 **this**.props.record.active = !**this**.props.record.active;  
 **this**.replaceProps({  
 record: **this**.props.record  
 })  
}

1. Это решение некорректно, потому что свойства должны приходить только от родительского компонента и state – единственное, что может быть изменено внутри компонента. Измнение свойств внутри компонента считается плохой практикой и метод replaceProps объявлен устаревшим в react. Давайте сделаем это правильно, следуя принципу: данные движутся вниз, действия движутся вверх. Передайте обработчик события в компонент GridRecord:

<GridRecord record={record} key={index} toggleActive={**this**.toggleActive.bind(**this**, index)}/>

1. Если вы заметили, мы использовали метод bind(). Почему в React нам надо использовать метод bind? C помощью этого метода мы меняем значение this внутри метода toggleActive(). Дело в том, что по умолчанию у метода toggleActive при вызове в коллбэке контекст this теряется. Использование bind() позволяет его жестко закрепить. Также мы передаем index, который будет передан в метод toggleActive() как первый аргумент.
2. Создайте обработчик события в классе GridComponent:

toggleActive(index){  
 **let** {records} = **this**.state;  
 records[index].active = !records[index].active;  
 **this**.setState({  
 records:records  
 })  
}

1. И активируйте его в GridRecord:

<th><input type="checkbox" checked={record.active} onChange={**this**.props.toggleActive}/></th>

1. Подытоживая:
   1. **GridComponent** отвечает за данные. Он знает, как изменить данные и, так как мы работаем с состоянием (state) – он знает, как перерисовать DOM после того, как данные изменены.
   2. **GridRecord** знает, как создать разметку, основываясь на входящих props. Он знает, что нужно вызвать, когда вы кликаете на него. Но если в какой-то момент вы решите заменить toggleActive() на что-то еще, GridRecord об этом не узнает – это решается на уровень выше – в компоненте GridComponent.

#### Задача 3: Сделайте фамилию в таблице редактируемой.

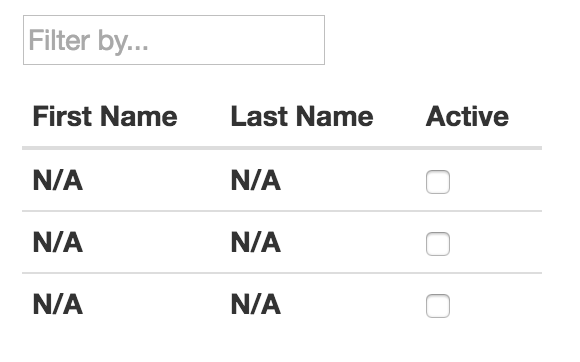
1. Фамилию можно изменить
2. После редактирования и перемещения курсора на другие элементы, значение поля остается измененным.
3. Давайте сделаем наши компоненты более легко переиспользуемыми. Прежде всего, нам понадобится определить defaultProps на случай, если к нам придут отсутствующие данные:

GridRecord.defaultProps = {  
 record: {firstName: "N/A", lastName: "N/A", active: **false**}  
};

1. Давайте проверим:

**return** <GridRecord record={undefined} key={index} toggleActive={**this**.toggleActive.bind(**this**, index)}/>

1. Как вы видите, теперь вместо ошибок в консоли и пустой таблицы мы увидим следующее:



1. Отлично, второе, что нам понадобится – это propTypes:

Установите модуль prop-types: **npm i prop-types**

**import** PropTypes **from 'prop-types'**;

GridRecord.propTypes = {  
 record: PropTypes.shape({  
 firstName: PropTypes.string.isRequired,  
 lastName: PropTypes.string.isRequired,  
 active: PropTypes.bool.isRequired  
 })  
};

Также нужно убедиться, что webpack.config.js работает в режиме development – в нем должна быть настройка режима:

**mode: "development"**

1. Давайте попробуем изменить dataSource, мы поменяем firstName на 123:

**const** dataSource = [  
 {firstName: 123, lastName: "Doe", active: **false**},  
 {firstName: "Mary", lastName: "Moe", active: **false**},  
 {firstName: "Peter", lastName: "Noname", active: **true**}  
];

1. Откроем консоль браузера:

Вы увидите предупреждение:

../../Desktop/Screen%20Shot%202016-04-16%20at%2012.20.44%20PM.p

Таким образом, у нас есть возможность дополнительного контроля типов поступающих данных в компонент. Конечно, при использовании языка TypeScript или подобного (например, Flow) нет необходимости использовать PropTypes. Но если вы используете JavaScript, PropTypes позволит вам быстрее поймать некоторые виды ошибок.

1. Теперь вы можете поменять 123 в dataSource обратно на John.
2. Это очень полезная штука. Хорошо, давайте закончим с нашим фильтром. Нам надо добавить новый слушатель в поле ввода в классе GridComponent:

onChange={**this**.handleFilterChange.bind(**this**)}

1. И сам метод в GridComponent:

handleFilterChange(e){  
 **let** value = e.target.value,  
 records = dataSource.filter((record) => record.firstName.toUpperCase().includes(value.toUpperCase()));  
 **this**.setState({  
 records:records  
 });  
}

1. Очень хорошо, но допустим у нас есть новое требование: сразу после открытия страницы фокус должен быть устновлен на поле фильтра. Как нам это сделать? Нам понадобится ссылка на элемент - ref:

<input type="text" ref="filterInput" placeholder="Filter by..." onChange={**this**.handleFilterChange.bind(**this**)}/>

И

componentDidMount(){  
 **this**.refs.filterInput **&& this**.refs.filterInput.focus();  
 **this**.setState({  
 records:dataSource  
 })  
}

Вот и все:

