**Работа с формами**

HTML компоненты, такие как input, textarea, и option отличаются от других неуправляемых компонентов, потому что они могут быть видоизменены посредством взаимодействия с пользователем.

Существует два типа компонентов:

- Контролируемые компоненты

- Неконтролируемые компоненты

**Контролируемые компоненты**:

Контролируемый компонент обеспечивает свойство prop «value». Контролируемый компонент не поддерживает своего собственного внутреннего состояния; генерация компонента основана на функциях props.

*render*() {  
 **return** (  
 <**input  
 type="text"  
 value="Hello!"** />  
 );  
}

Если вы попытаетесь запустить этот пример, вы увидите, что входные данные не изменяются по мере ввода. Это происходит потому, что компонент признал «value» входных данных всегда выводить «Hello!».

Чтобы обновить значение в ответ на ввод пользователя, вы должны использовать событие «onChange», чтобы сохранить новое значение, а затем передать его в функцию prop «value» входных данных:

**class** Form **extends** React.**Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {**value**: **''**};  
 **this**.handleChange = **this**.handleChange.bind(**this**);  
 **this**.handleSubmit = **this**.handleSubmit.bind(**this**);  
 }  
 handleChange(event) {  
 **this**.setState({**value**: event.**target**.**value**});  
 }  
 handleSubmit(event) {  
 alert(**'Text field value is: '** + **this**.**state**.**value**);  
 }  
 render() {  
 **return** (  
 <**div**>  
 <**input type="text"  
 placeholder="Hello!"  
 value=**{**this**.**state**.**value**}  
 **onChange=**{**this**.handleChange} />  
 <**button onClick=**{**this**.handleSubmit}>  
 Submit  
 </**button**>  
 </**div**>  
 );  
 }  
}  
**ReactDOM**.**render**(  
 <**Form** />,  
 **document**.getElementById(**'root'**)  
);

В этом примере, мы принимаем значение, указанное пользователем и обновляем свойство prop «value» компонента "". Этот шаблон позволяет легко реализовывать интерфейсы, которые реагируют на проверки или взаимодействия с пользователем. Например:

handleChange(event) {  
 **this**.setState({  
 **value**: event.**target**.**value**.substr(0, 5)  
 });  
}

**Неконтролируемые компоненты**:

Форма-компоненты, которые не обеспечивают prop «value», являются неконтролируемыми.

Приведенный ниже пример показывает управление "" при помощи пустого значения. Любой пользовательский ввод будет немедленно отражен посредством представленного элемента. Неконтролируемый компонент управляет своим собственным состоянием.

*render*() {  
 **return** <**input type="text"** />;  
}

Давайте поработаем с формами

Создадим компонент с формой регистрации. Создадим файл RegistrationForm.js



Добавим туда пустую заготовку под компонент

**import** React, { **Component** } **from 'react'**;  
**class** RegistrationForm **extends Component** {  
 render() {  
 **return** (  
 <**div**>Form</**div**>  
 );  
 }  
}  
**export default** RegistrationForm;

И импортируем ее в App.js

**import** React, { **Component** } **from 'react'**;  
**import** RegistrationForm **from './RegistrationForm'**;  
**class** App **extends Component**{  
 render(){  
 **return**(  
 <**div**>  
 <**RegistrationForm**/>  
 </**div**>  
 )  
 }  
}  
**export default** App;

Если мы перейдем в браузер, то увидим нашу форму. Давайте теперь добавим форму и submit метод:

**import** React, { **Component** } **from 'react'**;  
**class** RegistrationForm **extends Component** {  
 render() {  
 **return** (  
 <**form onSubmit=**{**this**.handleSubmit}>  
 </**form**>  
 );  
 }  
}  
**export default** RegistrationForm;

Представим, что у нас в форме будет email, который нужно заполнить. Для этого нам нужно в state указать дефолтное значение. Создадим конструктор, вызываем super и задаем дефолтное значение для email.

constructor(props) {  
 **super**(props);  
 **this**.**state** = {  
 **email**: **''** };  
}

Добавим input в нашу форму. Как значение зададим this.state.email. Также нам нужно задать onChange функцию. Она будет стрелять каждый раз, когда мы изменяем email.

<**form onSubmit=**{**this**.handleSubmit}>  
 <**input  
 type="text"  
 placeholder="E-mail"  
 value=**{**this**.**state**.**email**}  
 **onChange=**{**this**.handleEmailChange}  
 />  
</**form**>

И добавим метод handleSubmit и handleEmailChange

handleSubmit() {  
 **console**.log(**'form is submitted'**);  
}  
  
handleEmailChange() {  
 **console**.log(**'handleEmailChange'**, **this**);  
}

Если мы законсолим this в функции handleEmailChange, то увидим в браузере, что он undefined. Есть два способа, как подставить нужный нам контекст. Это использовать метод bind или стрелочную функцию. Предпочтительней - это использование стрелочной функции, т. к. это более производительный способ

*handleSubmit* = () => {  
 **console**.log(**'form is submitted'**);  
}  
  
*handleEmailChange* = () => {  
 **console**.log(**'handleEmailChange'**, **this**);  
}

Теперь в браузере мы видим правильный this.

Давайте теперь когда стреляет функция handleEmailChange сетить значение в стейт.

*handleEmailChange* = (event) => {  
 **console**.log(**'handleEmailChange'**, **this**);  
 **this**.setState({**email**: event.**target**.**value**});  
}

Теперь каждый раз при изменении инпута у нас будет менятся значение в локальном стейте. Если мы попробуем печатать в форме, мы видим, что у нас меняется значение как в инпуте так и в стейте.

Как вы видите, у нас сразу же в инпуте отображаются данные. Если мы уберем строчку setState, то мы будем что-то печатать и у нас ничего не отображается, поскольку значение this.state.email не изменилось. Как только мы вызываем setState, у нас отображается значение с локального стейта.

Давайте изменим немного сообщение в handleSubmit. В браузере, если мы напишем что-то в input и нажмем enter, console.log у нас выводится, но страница перегружается так как это обычная форма. Для того, чтобы это пофиксить давайте добавим event.preventDefault().

*handleSubmit* = (event) => {  
 event.preventDefault();  
 **console**.log(**'form submitted and email value is'**, **this**.**state**.**email**);  
}

Теперь если мы будем что-то писать в инпуте и нажмем enter, то у нас страница не перезагружается и в консоли мы видим надпись.

Ну и напоследок давайте добавим кнопку для сабмита формы

Так как у button, по умолчанию тип submit, и наша форма отправляется и мы попадаем в функцию handleSubmit.

**import** React, { **Component** } **from 'react'**;  
**class** RegistrationForm **extends Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {  
 **email**: **''** };  
 }  
 *handleSubmit* = (event) => {  
 event.preventDefault();  
 **console**.log(**'form submitted and email value is'**, **this**.**state**.**email**);  
 }  
  
 *handleEmailChange* = (event) => {  
 **console**.log(**'handleEmailChange'**, **this**);  
 **this**.setState({**email**: event.**target**.**value**});  
 }  
 render() {  
 **return** (  
 <**form onSubmit=**{**this**.*handleSubmit*}>  
 <**input  
 type="text"  
 placeholder="E-mail"  
 value=**{**this**.**state**.**email**}  
 **onChange=**{**this**.*handleEmailChange*}  
 />  
 <**button**>Save</**button**>  
 </**form**>  
 );  
 }  
}  
**export default** RegistrationForm;

**Валидация react props с помощью PropTypes**

Идея **PropTypes** в том, чтобы описать какие типы данные и значения может принимать компонента на вход, чтобы если ей передали неправильные значения, вывадилась ошибка.

Это очень хорошо помогает когда над проектом работают несколько человек. Обычно открываешь компонент и непонятно, что в него нужно передать, чтобы он работал правильно и отрисовал то, что нужно. А если описать PropTypes, то вопросов вообще не возникает.

Для начала создадим компонент Header:

**import** React, { **Component** } **from 'react'**;  
**class** Header **extends Component**{  
 render(){  
 **return**(  
 <**div**>  
 {**this**.**props**.**items**.map((item,index) => {  
 **console**.log(**'items'**, item.**label**);  
 **return** <**a href=**{item.**link**} **key=**{index}>{item.**label**}</**a**>  
 })}  
 </**div**>  
 )  
 }  
}  
**export default** Header;

И импортируем его в App.js:

**import** React, { **Component** } **from 'react'**;  
**import** Header **from './Header'**;  
**const** menu = [  
 {  
 **link**: **'/articles'**,  
 **label**: **'Articles'** },  
 {  
 **link**: **'/contacts'**,  
 **label**: **'Contacts'** },  
 {  
 **link**: **'/posts'**,  
 **label**: **'Posts'** }  
]  
**class** App **extends Component**{  
 render(){  
 **return**(  
 <**div**>  
 <**Header items =** {menu}/>  
 </**div**>  
 )  
 }  
}  
**export default** App;  
  
Для начала работы с propTypes необходимо импортировать из реакта:

**import** React, { **Component**, **PropTypes** } **from 'react'**;

Добавим в наш компонент Header PropTypes.  
  
**static** *propTypes* = {  
   
};

Теперь мы хотим описать, что в наш компонент обязательно должен приходить массив items.  
  
**static** *propTypes* = {  
 **items**: **PropTypes**.**array**.**isRequired**}

Мы указали, что items должен быть типа array, а isRequired показывает, что эти данные нужно передавать обязательно.

Если мы посмотрим в браузер, то никаких ошибок сейчас в консоли нет. Давайте попробуем убрать items из параметров компонента.



И мы сразу получили warning, что items не были переданы в компонент.

Давайте попробуем передать items типа string

**class** App **extends Component**{  
 render(){  
 **return**(  
 <**div**>  
 <**Header items = 'string'**/>  
 </**div**>  
 )  
 }  
}  
**export default** App;

Выдаст ошибку



Этот warning говорит нам, что тип должен быть array а не string. Эти сообщение очень помогают, когда ты пытаешься использовать компоненту в первый раз и не знаешь, что ей передать.

Давайте рассмотрим самые популярные проверки типов.

**isLoading**: **PropTypes**.**bool**

Это означает, что компонента может принимать на вход isLoading булевого типа. Но не обязательно переменная должна быть.

**submit**: **PropTypes**.**func**

Здесь мы описали, что на вход нужно обязательно передать функцию submit.

**title**: **PropTypes**.**string**.**isRequired**  
  
Еще можно валидировать сами данные. Например  
**type**: **PropTypes**.oneOf([**'news'**, **'photos'**])

То есть здесь мы описали, что хотим получить type либо значения news, либо photos.  
  
**user**: **PropTypes**.shape({  
 **name**: **PropTypes**.**string**,  
 **age**: **PropTypes**.**number**})

Здесь мы описали, что хотим получить юзера с полями name и age, которые будут типов string и number соответственно.

Если же мы хотим провалидировать массив юзеров, то можно записать это так.

**users**: **PropTypes**.arrayOf(  
 **PropTypes**.shape({  
 **name**: **PropTypes**.**string**,  
 **age**: **PropTypes**.**number** })  
)

Этим мы ожидаем, что у нас будет массив обьектов с правильными данными.

<https://habrahabr.ru/post/319358/>

**REFS**

Мы разобрали с вами два типа данных в React: props и state. Нам осталось разобрать еще один тип данных в React. Это refs. Он позволяет обращатся к конкретному DOM елементу компонента. Давайте создадим инпут и баттон:

**import** React, { **Component** } **from 'react'**;  
**class** App **extends Component** {  
 render() {  
 **return** (  
 <**div**>  
 <**input type="text" placeholder="test"** />  
 <**button**>Submit</**button**>  
 </**div**>  
 );  
 }  
}  
**export default** App;

Если мы посмотрим в браузер, то увидим, что наша форма отрисовалась.

Добавим атрибут onClick и зададим метод this.submit нашему событию.

<**button onClick=**{**this**.*submit*}>Submit</**button**>

И создадим необходимый метод:

*submit*=() => {  
 **console**.log(**'submit'**);  
}

Если мы посмотрим в браузер, то наша функция submit вывела консоль лог при клике.

Теперь для того, чтобы получить значение DOM елемента input нужно навесить на него вот такую конструкцию.

Здесь мы указали атрибут ref, который является функцией и присваивает input в переменную this.testInput. React вызвает ref callback, когда наш компонент рендерится. Это обычный паттерн в React как получить доступ к DOM елементам.

Теперь в нашей функции submit мы можем вывести this.testInput.

*submit*=() => {  
 **console**.log(**'submit'**, **this**.**testInput**);  
}

Как мы видим, в консоль нам вывелся DOM елемент. Теперь мы можем делать с ним что угодно. Например, узнать его значение в данный момент.

**import** React, { **Component** } **from 'react'**;  
**class** App **extends Component** {  
 *submit*=() => {  
 **console**.log(**'submit'**, **this**.**testInput**.**value**);  
 }  
 render() {  
 **return** (  
 <**div**>  
 <**input type="text" placeholder="test" ref=**{(input) => **this**.**testInput** = input} />  
 <**button onClick=**{**this**.*submit*}>Submit</**button**>  
 </**div**>  
 );  
 }  
}  
**export default** App;

Если мы введем что-то в input и нажмем submit, то в консоль выведется значение input.

Ref отличный способ доступа к DOM елементам, но его нужно применять с осторожностью. Нужно помнить, что это не React way, а просто возможность доступа к DOM елементам. Всегда лучше использовать state, props если это возможно вместо refs, так как они поддерживают правильный флоу данных в приложении, а refs нет.

**ROUTERS**

Для начала работы с роутами их необходимо установить

npm install --save react-router@3.0.2

После установки мы переходим в наш файл index.js и прописываем следующее:

**import** React **from 'react'**;  
**import** ReactDOM **from 'react-dom'**;  
**import './index.css'**;  
**import** *App* **from './App'**;  
**import** { Router, Route, browserHistory, IndexRoute } **from 'react-router'**;  
  
ReactDOM.**render**((  
 <**Router history =** {browserHistory}>  
 <**Route path = '/'**>  
 <**IndexRoute component=**{*App*}/>  
 <**Route path = '/page-:id' component=**{*App*}/>  
 </**Route**>  
 </**Router**>  
), **document**.getElementById(**'root'**))

<Router/> - главный компонент, в котором будут прописаны маршруты

browserHistory - объект browserHistory, который хранит путь к текущему location и перерисовывает интерфейс сайта когда происходят какие то изменения пути.

<Route/> - это строительный блок React Router'а. В том случае если вам нужно рендерить элемент в зависимости от pathname URL'ов, то следует использовать компонент <Route/>

path - это свойство, которое отвечает за путь. Реакт сравнивает значение из адресной строки и запускает соответствующий этому адресу роут.

<IndexRoute/> - это компонент, который запускается при пути равному пути родительского роута

component - здесь прописывается компонента, которая должна отрендериться

И меняем App.js:

**import** React **from 'react'**;  
**import** {Link} **from 'react-router'**;  
**const** links = [1,2,3,4,5,6,7,8,9,10];  
**const** *App* = (props) => (  
 <**div**>  
 This is page {props.**params**.**id** !== **undefined** ?  
 **`**${props.**params**.**id**}**`** : **'IndexPage'**}  
 {links.map((item)=>{  
 **if** (item !== props.**params**.**id**)  
 **return**(  
 <**div key =** {item}>  
 <**Link to =** {**`/page-**${item}**`**}>  
 Redirect to {**`page-**${item}**`**}  
 </**Link**>  
 <**br**/>  
 </**div**>)  
 })}  
 </**div**>  
)  
**export default** *App*;

<Link> - это компонент, который мы используем вместо тега <a>. Свойство to - это аналог href

Когда мы прописываем в свойстве path = ':id', мы можем взять значение с адресной строки, которое стоит на месте :id в соответствующем компоненте с помощью props.params.id

<https://maxfarseer.gitbooks.io/react-router-course-ru/content/sozdaem_primitivnii_router.html>

<http://prgssr.ru/development/pogruzhenie-v-react-router.html#heading-section-4>

<https://habrahabr.ru/post/329996/>