

# 4. Передача данных по пропсам

---

## Содержание урока

---

- Обзор;
- Концепция потока данных в React;
- Пропсы;
- Валидация пропсов;
- Дефолтные пропсы;
- Tips and tricks;
- Подведём итоги.

## Обзор

---

Привет! 🖐️ В этом уроке мы рассмотрим модель потока данных в React, а также познакомимся с первым механизмом передачи данных от компонента к компоненту. 🐼

## Концепция потока данных в React

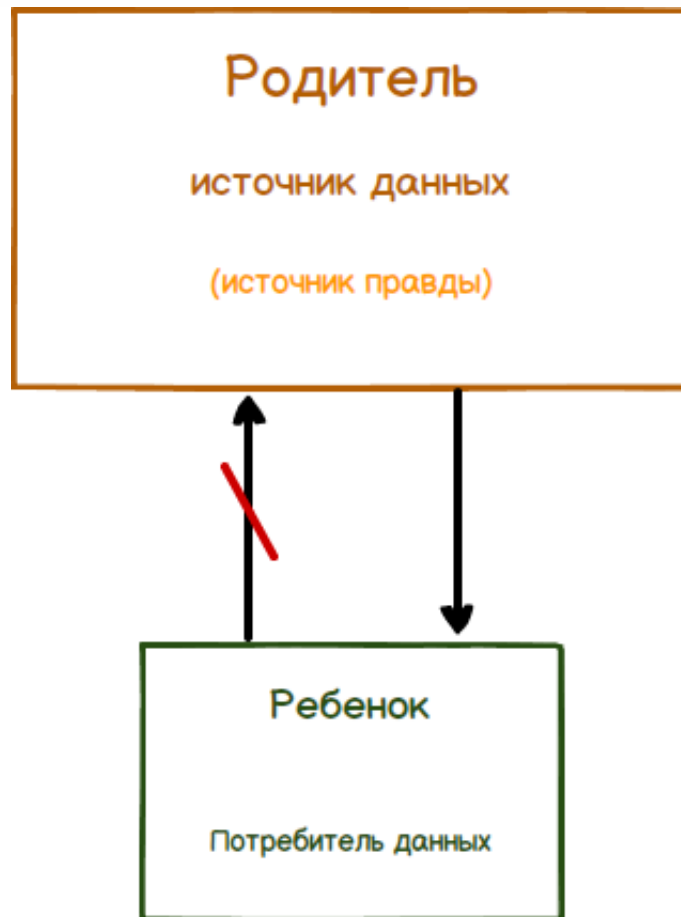
---

React следует концепции `однонаправленного потока данных`.

🔍 Хозяйке на заметку:

Иногда этот подход ещё называют `one-way data flow` или `unidirectional data flow`.

Данная модель описывает поток данных только в одну сторону - от источника данных к их потребителю.



🔍 Хозяйке на заметку:

Данная модель контрастирует с моделями данных других инструментов для постройки UI, в которых данными можно обмениваться в обе стороны. Такая модель называется «two-way data flow» («двусторонний поток данных»). В «двустороннем потоке данных» данные, привязанные к ребенку и к родителю, всегда должны находиться в синхронизированном состоянии, независимо от того, с какой стороны произошло обновление данных: со стороны ребенка или родителя.

Давай рассмотрим, как работает «однонаправленный поток данных» в React на примере мини-приложения чата (в этом нам поможет Оскар). Состоять оно будет из нескольких компонентов:

- `Chat` — родительский компонент
- `ChatMessage` — дочерний компонент

Компонент `Chat` объединит в себе экземпляры компонента `ChatMessage` — и поделится с ними данными посредством их передачи через канал данных «props». 🧑

## Пропсы


«Пропсы» (англ. «props») — это обычный иммутабельный JavaScript-объект, содержащий данные. Данные, доступные в объекте «props», передаются сверху родительским компонентом вниз по иерархии дочернему компоненту, становясь, таким образом, доступными для употребления.

Доступ к «props» можно получить, обратившись к свойству `this.props` классового компонента.

 Пример кода 4.1:

```
1 // Chat.js
2
3 import React, { Component } from 'react';
4 import ChatMessage from './ChatMessage';
5
6 const data = {
7   author: 'Oscar',
8   message: 'Hello there!'
9 };
10
11 export default class Chat extends Component {
12   render () {
13
14     return (
15       <>
16         <ChatMessage
17           author = { data.author }
18           message = { data.message }
19         />
20       </>
21     );
22   }
23 }
```

```
1 // ChatMessage.js
2
3 import React, { Component } from 'react';
4
5 export default class ChatMessage extends Component {
6   render () {
7     const { author, message } = this.props;
8
9     return <span>Author: { author }, message: { message }</span>;
10   }
11 }
```

 Хорошая практика:

Среди комьюнити-разработчиков хорошей практикой является использование приема деструктуризации. Например, вместо многократного обращения к `this.props...` в возвращаемом значении метода `render` мы можем деструктурировать нужные значения в начале тела метода `render` и использовать эти значения повсеместно, избежав дублирования.

Давай разберем `пример кода 4.1` построчно:

Компонент `ChatMessage`:

1. На строке кода 7: используется синтаксис деструктурирующего присваивания ES2015, изъяс и привязав данные к идентификаторам `author` и `message` из соответствующих свойств объекта `this.props` (пропсов компонента `ChatMessage`). Данные доступны в объекте `«props»` по причине их передачи компонентом-родителем `Chat` компоненту-ребенку `ChatMessage`.
2. На строке кода 9: возвращается элемент `<span>` с включенными JSX-выражениями, несущими в себе ссылки на идентификаторы `author` и `message`. При рендере компонента на месте этих идентификаторов окажутся значения, сброшенные родительским компонентом `Chat`.

Компонент `Chat`:

1. На строке кода 6: создан объект с данными, которые необходимо пробросить.
2. На строке кода 16: рендерится компонент `ChatMessage` с передачей ему двух пропсов:
  1. Пропс с именем `author` и значением, взятым из `data.author`.
  2. Пропс с именем `message` и значением, взятым из `data.message`.

В результате такой композиции данные, переданные родительским компонентом `Chat` по пропсам, становятся доступными для употребления из объекта `this.props` компонентом `ChatMessage`. Это похоже на волейбол, когда один передаёт, а второй — ловит. 🏐

! Важно:

Здесь стоит отметить, что для каждого экземпляра компонента всегда создаётся новый и уникальный объект `«props»`. Так происходит с учётом компонентной природы в React. Компонент — это переиспользуемая уникальная сущность, всегда имеющая экземпляр себя при создании.

Учитывая, что функциональный компонент в React — это обычная функция, возвращающая JSX и инкапсулирующая определенную логику, объект `«props»` для них доступен первым аргументом.

🖥️ Пример кода 4.2:

```
1  const ChatMessage = ({ author, message }) => {  
2  
3      return <span>Author: { author }, message: { message }</span>;  
4  }
```

В примере кода 4.2 на строке 1 произведена деструктуризация нужных значений из объекта `«props»`, коим является первый аргумент компонента `ChatMessage`. Данная доступность удобна при чтении `«пропсов»` в функциональных компонентах. 🐱

# Валидация пропсов


«Пропсы» можно пробрасывать вниз по иерархии компонентов манерой «водопада», читая сверху и сбрасывая данные вниз на каждом компонентном уровне. 🌊

Данная особенность приводит к вполне очевидной опасности потерять или перепутать данные на одном из уровней. Для защиты этой уязвимости существует механизм валидации типов данных, передаваемых по пропсам. Валидацию можно осуществить с помощью npm-пакета `prop-types`.

 Пример кода 4.3:

```
1 // Page.js
2
3 import React, { Component } from 'react';
4 import PropTypes from 'prop-types';
5
6 export default class Page extends Component {
7   static propTypes = {
8     spell: PropTypes.string
9   }
10
11   render () {
12     const { spell } = this.props;
13
14     return <h1>Preparing to cast the { spell }!</h1>;
15   }
16 }
```

```
1 // SpellBook.js
2
3 import React, { Component } from 'react';
4 import Page from './Page';
5
6 export default class SpellBook extends Component {
7   render () {
8
9     return <Page spell = 'Arcane fire' />;
10   }
11 }
```

 Хозяйке на заметку:

До версии `React v15.5` объект валидаторов данных `PropTypes` существовал в одноимённом пакете `React` вместе со многими другими вспомогательными инструментами. В следующих версиях логика валидации была вынесена в отдельный npm-пакет `prop-types` для лучшего соблюдения подхода разделения ответственности.

Объект `PropTypes` содержит набор валидаторов данных. В примере кода `4.3` мы используем валидатор `PropTypes.string`, чтобы убедиться, что в свойстве с именем `spell` объекта `props` будет содержаться правильный тип данных — строка. Если передать в пропс `spell` другой тип данных, `prop-types` предупредит разработчика, распечатав сообщение в консоли:

```
1 Warning: Failed prop type: Invalid prop `spell` of type `number` supplied
  to `Spell`, expected `string`.
```

Такое сообщение будет распечатано в консоли разработчика, если передать компоненту `Page` число в виде значения пропса `spell`.

У любого валидатора `prop-types` также существует возможность указать, что присутствие значения для этого пропса обязательно:

```
1 static propTypes = {
2   spell: PropTypes.string.isRequired
3 };
```

Пакет `prop-types` предоставляет широкий выбор доступных валидаторов:

```
1 PropTypesExampleComponent.propTypes = {
2
3   optionalArray:      PropTypes.array,
4   optionalBool:       PropTypes.bool,
5   optionalFunc:       PropTypes.func,
6   optionalNumber:     PropTypes.number,
7   optionalObject:     PropTypes.object,
8   optionalString:     PropTypes.string,
9   optionalSymbol:     PropTypes.symbol,
10  optionalNode:       PropTypes.node,
11  optionalElement:    PropTypes.element,
12  optionalMessage:    PropTypes.instanceOf(Message),
13  optionalEnum:       PropTypes.oneOf([ 'News', 'Photos' ]),
14
15  optionalUnion:       PropTypes.oneOfType([
16    PropTypes.string,
17    PropTypes.number,
18    PropTypes.instanceOf(Message)
19  ]),
20
```

```

21 optionalArrayOf:      PropTypes.arrayOf(PropTypes.number),
22 optionalObjectOf:     PropTypes.objectOf(PropTypes.number),
23
24 optionalObjectWithShape: PropTypes.shape({
25     color:      PropTypes.string,
26     fontSize:   PropTypes.number
27 }),
28
29 requiredFunc:          PropTypes.func.isRequired,
30 requiredAny:           PropTypes.any.isRequired
31
32 };

```

## Дефолтные пропсы

Иногда разработчик не всегда уверен в том, получит ли дочерний компонент пропс от родителя или нет. Для таких случаев существует фолбэчный механизм `defaultProps`:

 Пример кода 4.4:

```

1  // Page.js
2
3  import React, { Component } from 'react';
4  import { string } from 'prop-types';
5
6  export default class Page extends Component {
7      static propTypes = {
8          spell: string
9      }
10
11     static defaultProps = {
12         spell: 'fireball'
13     }
14
15     render () {
16         const { spell } = this.props;
17
18         return <h1>Preparing to cast a { spell }!</h1>;
19     }
20 }

```

Если компоненту `Page` родитель не передаст пропс `spell` явно, сработает фолбэк и свое значение пропс `spell` возьмёт из объекта `defaultProps`.

# Tips and tricks

В некоторых ситуациях дочернему компоненту необходимо передать большое количество пропсов. В таком случае нет необходимости передавать каждый пропс по одному вручную — можно воспользоваться специальным синтаксисом `spread props`, передав все пропсы за один раз.

 Пример кода 4.5:

```
1 // index.js
2
3 import React, { Component } from 'react';
4 import Page from './Page';
5
6 const props = {
7   spell:      'Spike of ice',
8   spellPower: 7
9 };
10
11 export default class SpellBook extends Component {
12   render () {
13
14     return (
15       <section>
16         <Page spell = 'Wall of Fire' spellPower = { 10 } />
17         <Page { ...props } />
18       </section>
19     );
20   }
21 }
```

```
1 // Page.js
2
3 import React, { Component } from 'react';
4 import PropTypes from 'prop-types';
5
6 export default class Page extends Component {
7   static propTypes = {
8     spell:      PropTypes.string.isRequired,
9     spellPower: PropTypes.number.isRequired
10   }
11
12   render () {
13     const { spell, spellPower } = this.props;
14
15     return <h1>Preparing to cast a { spell }! The power of this
spell is: { spellPower }!</h1>;
```



```
16     }  
17 }
```

В примере кода 4.5 компонент `SpellBook` рендерит два экземпляра компонента `Page`:

- На строке кода 16: компоненту `Page` пропсы передаются явно, по одному;
- На строке кода 17: компоненту `Page` пропсы передаются за один раз.

Чем больше пропсов необходимо передать, тем большая польза от данного подхода. Однако следует использовать этот подход выборочно: иногда передать слишком много ненужных пропсов может быть лишним.

! Важно:

Ещё одна особенность «пропсов» — иммутабельность. Для обеспечения максимального уровня предсказуемости и надёжности приложения пропсы нельзя мутировать. При попытке сделать это возникнет ошибка.

## Подведём итоги

«Однонаправленный поток данных» — одна из самых неотразимых характеристик React. Она делает приложение более прозрачным в восприятии ментально, помогает его «охватить».

Из преимуществ следования данному подходу можно выделить:

- При чтении из «`props`» мы всегда знаем, откуда данные пришли;
- «`props`» — достаточно мощный канал передачи данных с учетом возможности использования потенциала языка программирования JavaScript;
- Переиспользование компонентов становится более безопасным и доступным;
- Легко валидировать;
- Легко тестировать.

В следующем уроке мы изучим еще один способ передачи данных в React —

«контекст».

Мы будем очень признательны, если ты оставишь свой фидбек в отношении этой части конспекта на нашу электропочту [hello@lectrum.io](mailto:hello@lectrum.io).