Глава 1

Все что нужно знать о React

Глава 2

Делаем код чище

Чтобы использовать JSX без проблем, необходимо понимать как он работает под капотом и почему его удобно использовать для создания интерфейса.

Наша цель - писать чистый и поддерживаемый JSX код, и для этого мы должны знать, как JSX транслируется в JavaScript код и какие предоставляет фичи.

В этом блоке мы разберем:

- Что такое JSX и почему мы должны его использовать
- Что такое Babel и как он используется в современной разработке
- Основные фичи JSX и его отличие от HTML
- Лучшие практики при написании кода на JSX
- Статическую проверку кода с ESLint
- Основы функционального программирования и его применение в создании React компонент

2.1 JSX

**Example of JSX

React предлагает два основных способа описания элементов: использование библиотечных JavaScript функций и использование JSX разметки, похожей на XML.

На первый взгляд может показаться, что JSX - это странная смесь из HTML и JavaScript. Но на деле, JSX разметка перед попаданием в браузер конвертируется в чистый JavaScript. Этот прием позволяет достаточно лаконично и визуально понятно описывать компоненты.

Babel

Чтобы использовать JSX(а также фичи ES2015) нам необходимо установить Babel.

Важно понимать, почему мы добавляем Babel в наш процесс разработки. Основная причина в желании использовать фичи языка, которые еще не доступны в браузере. Новые фичи языка часто помогают нам писать код чище и понятнее, но браузер не может их исполнить.

Решение - писать код с JSX и ES2015, а затем транслировать в ES5, который сейчас могут запускать большинство браузеров, используя Babel.

Чтобы использовать Babel, его необходимо установить:

```
npm install --global babel-cli
```

Чтобы не загромождать npm пакетами систему, можно установить babel в конкретный проект и использовать через npm скрипты. Но для учебных целей установим его глобально.

После установки мы можем транслировать любой JavaScript файл:

```
babel source.js -o output.js
```

Одна из сильных сторон Babel - возможность его гибкой конфигурации. Babel всего лишь транслирует один файл в другой, а чтобы были произведены изменения содержимого, нужно сконфигурировать этот процесс.

Для Babel уже создано множество пресетов, в том числе и для JSX и ES2015. Чтобы установить их, необходимо выполнить:

```
npm install --global babel-preset-es2015 babel-preset-react
```

а также добавить в домашнюю директорию (или в папку с проектом) файл .babelrc с содержимым:

```
{
  "presets": [
    "es2015",
    "react"
  ]
}
```

С этого момента мы можем спокойно использовать все фичи ES2015 и JSX, а потом запускать в браузере транслированный Babel'ем код.

Hello, World

Посмотрим на простейший пример создание элемента в React.

Мы можем создать *div* элемент с помощью метода *createElement* библиотеки React:

```
React.createElement('div')
```

Также мы можем создать его, используя JSX:

```
<div />
```

В данном примере JSX код выглядит как HTML. Но нужно понимать одну важную вещь, оба варианта, написанные выше, эквивалентны.

На самом деле, если мы транслируем в JavaScript при помощи babel, то мы получим React.createElement('div'). Это необходимо всегда держать в голове при описании интерфейса на React.

DOM элементы и React компоненты

С JSX мы можем создавать и HTML элементы и React элементы. Разница лишь в том, пишем мы название элемента с заглавной буквы или нет.

Например, в JSX мы можем создать <button /> и <Button /> элементы.

В первом случае результатом трансляции будет:

```
React.createElement('button')
```

Во втором случае:

```
React.createElement(Button)
```

Разница в том, что в первом случае тип DOM элемента передается как строка, а во втором случае мы передаем название переменной, которая должна быть как минимум определена в области видимости данного кода.

Props

JSX очень удобен, если у DOM или React компонентов есть параметры. В XML в целом гораздо нагляднее передавать параметры элементам:

```
<img src="https://facebook.github.io/react/img/logo.svg"
   alt="React.js" />
```

Аналогичный код на JavaScript:

```
React.createElement("img", {
     src: "https://facebook.github.io/react/img/logo.svg",
     alt: "React.js"
});
```

Читаемость резко падает даже с небольшим количеством параметров.

Children

JSX позволяет определять дочерние элементы, чтобы создавать комплексные древовидные разметки.

Простой пример дочернего элемента - текст внутри тега:

```
<a href="https://facebook.github.io/react/">Click me!</a>
```

Этот пример будет транслирован в:

```
React.createElement(
  "a",
  { href: "https://facebook.github.io/react/" },
  "Click me!"
);
```

Если этот тег будет обернут в другой, например div, то JSX будет выглядеть как:

```
<div>
  <a href="https://facebook.github.io/react/">Click me!</a>
</div>
```

JavaScript эквивалентный этому JSX:

```
React.createElement(
  "div",
  null,
  React.createElement(
    "a",
    { href: "https://facebook.github.io/react/" },
    "Click me!"
```

```
  )
);
```

Достаточно очевидно, что чем сложнее разметка, тем больше JSX улучшает читаемость кода. Однако не следует забывать, что каждому JSX коду соответствует однозначно определенный код на JavaScript.

Так как JSX всего лишь удобный синтаксис для JavaScript, вполне логично, что в нем можно использовать JavaScript выражения.

Для того, чтобы сделать это, выражение должно быть обернуто в фигурные скобки:

```
<div>
  Hello, {variable}.
  I'm a {function()}.
</div>
```

Или например в параметрах элемента:

```
<a href={this.makeHref()}>Click me!</a>
```

Differences with HTML

Мы посмотрели, чем похожи JSX и HTML. Теперь посмотрим в чем они отличаются и в чем причины этих различий.

Аттрибуты

Мы должны помнить, что JSX не стандарт языка, и транслируется в JavaScript. Из-за этого некоторые атрибуты не доступны для использования.

Например, вместо атрибута class мы вынуждены использовать className, а вместо for использовать htmlFor:

```
<label className="awesome-label" htmlFor="name" />
```

Причина этого в том, что слова class и for зарезервированы в языке JavaScript.

Стили

Стили - пример значительных различий между HTML и JSX. Подробнее мы посмотрим на них в одной из следующих глав.

Сейчас отметим, что через JSX в атрибуте style не поддерживается CSS строка. Вместо нее React ожидает JavaScript объект, в котором имена стилей переданы в camelCase:

```
<div style={{ backgroundColor: 'red' }} />
```

Root

Стоит отметить важное отличие JSX от HTML, которое заключается в том, что нельзя создать несколько элементов на одном уровне без оборачивания другим элементом:

```
<div />
<div />
```

Этот пример вызовет следующую ошибку:
Adjacent JSX elements must be wrapped in an enclosing tag
Проблема решается добавлением общего элемента:

```
<div>
  <div />
  <div />
</div>
```

Это происходит из-за того, что каждый элемент в JSX транслируется в React.createElement в JavaScript, а в JavaScript нельзя вернуть из функции результат работы двух последовательных вызовов какой-либо функции.

Сейчас React предоставляет возможность использовать пустой тег, чтобы отрисовывать несколько элементов на одном уровне:

```
render() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  );
}
```

Spaces

Есть еще одна мелочь, которая может вводить в ступор новичков, которая заключается в различной обработке пробельных символов в HTML

и JSX.

Например, рассмотрим следующий пример кода, который корректен и в HTML и в JSX:

```
<div>
  <span>foo</span>
  bar
  <span>baz</span>
</div>
```

Если открыть этот кусочек напрямую в браузере как HTML файл, то мы увидим foo bar baz. А если мы добавим его в JSX, то после отрисовки будет foobarbaz.

Это происходит из-за того, что JSX воспримет три строки внутри div, как три дочерние элементы и проигнорирует пробельные символы, а после отрисовке все три дочерних элемента будут отрисованы один за другим.

Основной способ исправить это, добавить еще дочерние элементы, которые будут также являться дочерними элементами:

```
<div>
  <span>foo</span>
  {' '}
  bar
  {' '}
  <span>baz</span>
</div>
```

Таким образом мы добавляем пустые строки, которые являются JavaScript выражениями, чтобы заставить компилятор добавить новые дочерние элементы.

Boolean атрибуты

Также есть небольшое отличие в использовании boolean атрибутов в JSX. Если передать какой-либо атрибут без значения, то JSX поймет, что это boolean атрибут со значением true:

```
<button disabled />

React.createElement("button", { disabled: true });
```

Но в отличие от HTML, чтобы передать атрибут со значением false, необходимо сделать это в явном виде. Если не передать атрибут совсем,

то он не попадет в передаваемый объект с атрибутами, и при дальнейшей попытке использования может быть получен undefined вместо false, что приводит к потенциальным ошибкам:

```
<button disabled={false} />

React.createElement("button", { disabled: false });
```

Эта особенность может вводить в заблуждение, так как в HTML принято отсутствие атрибута считать как false значение для этого атрибута. В React следует всегда явным образом указывать значение boolean атрибутов.

Spread атрибут

Важная особенность JSX - **spread атрибуты**, которая приходит из стандарта ECMASript (https://github.com/sebmarkbage/ecmascript-re st-spread) и очень удобно в случае, когда нам нужно передать все атрибуты JavaScript объекта в параметры элементу.

Распространенная практика - избежать передачи объекта дочерним элементам по ссылке во избежании ошибок, связанных с изменяемостью таких объектов.

В качестве примера можем посмотреть на код:

```
const foo = { id: 'bar' }
return <div {...foo} />
```

который будет транслирован в следующий JSX код:

```
var foo = { id: 'bar' };
return React.createElement('div', foo);
```

Шабоны JavaScript

Мы начали с предположения, что одно из преимуществ использования шаблонов внутри наших компонент вместо использования сторонних библиотек шаблонов(прим.пер. видимо имеются ввиду библиотеки-шаблонизаторы как handlebars) в использовании всей мощи языка JavaScript внутри шаблонов.

Spread оператор один из примеров использования JavaScript внутри JSX. Но в целом любое JavaScript выражение может быть использовано

как атрибут элемента, для этого достаточно обернуть его в фигурные скобки:

```
<button disabled={errors.length} />
```

Основные паттерны

Мы разобрались с тем, как работает JSX. Теперь мы можем можем подумать детальнее, как использовать JSX, следуя полезным соглашениям и практикам.

Многострочный JSX код

Начнем с простого примера. Одна из причин использовать JSX вместо React.createElement - наглядность XML-like синтаксиса, а также потому что такая структура из отрывающих и закрывающих тегов идеально подходит для описания древовидных структур.

Например, если у нас есть JSX код с множеством вложенных элементов, мы должны предпочитать многострочную запись JSX кода однострочной:

```
<div>
  <Header />
  <div>
    <Main content={...} />
  </div>
</div>
```

Такой вариант гораздо предпочтительнее, чем:

```
<div><Header /><div><Main content={...} /></div></div>
```

Однако, если дочерний элемент - не React элемент, а текст или переменная, то разумнее будет записать весь тег в одной строке:

```
<div>
  <Alert>{message}</Alert>
  <Button>Close</Button>
</div>
```

Также рекомендуется оборачивать все JSX блоки в круглые скобки. Это необходимо делать, чтобы не было проблем с автоматической вставкой точки с запятой. Проблемы могут возникнуть, если например, немного не аккуратно вернуть. JSX разметку из функции.

Следующий пример отработает корректно, так как return и div находятся на одной линии:

```
return <div />
```

Но следующий уже отработает непредвиденным образом:

```
return
     <div />
```

Проблема кроется в том, что во втором случае JSX код будет транслирован в следующий JavaScript код:

```
return;
React.createElement("div", null);
```

Чтобы избежать таких проблем, рекомендуется всегда оборачивать многострочные JSX элементы в круглые скобки:

```
return (
  <div />
)
```

Multi-properties

Небольшой проблемой является также множество атрибутов у элемента. Если записывать атрибуты в одну строку, то она может начать занимать много места в ширину и быть неудобной для чтения.

Поэтому стоит стараться писать каждый атрибут в новой строке, а закрывающий тег выравнивать с открывающим (прим.пер. также хороший вариант - оставлять закрывающий тег на одной строке с последним атрибутом):

```
<button
  foo="bar"
  veryLongPropertyName="baz"
  onSomething={this.handleSomething}
/>
```

Условные операторы

Все гораздо интереснее с использование **условий** в JSX, например, если нужно отрисовать какой-то компонент только при каком-то условии. С

одной стороны возможность использовать JavaScript внутри JSX - большой плюс, но с другой стороны у нас появляется множество вариантов использования условий и нужно знать об их плюсах и минусах.

Предположим, что нам нужно отрисовать кнопку logout только для авторизованного пользователя. Простой вариант решения этой проблемы может выглядеть следующим образом:

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
}
return <div>{button}</div>
```

Это работает, но этот вариант плохо читается, если у нас есть множество условий и множество элементов.

Также мы можем использовать ленивую проверку логических выражений в JavaScript:

```
<div>
  {isLoggedIn && <LoginButton />}
</div>
```

Это работает, так как в случае false в isLoggedIn JavaScript не будет проверять остальное выражение, а если true, тогда вызовется createElement для LoginButton и результат ее работы вернется как результат всего выражения.

Если мы хотим, чтобы в условии была также альтернативная ветка, например чтобы показать разные кнопки для авторизованного и неавторизованного пользователей, то мы может использовать if...else в JavaScript:

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
} else {
  button = <LoginButton />
}
return <div>{button}</div>
```

Помимо этого мы можем использовать тернарный оператор, чтобы сделать код компактнее:

```
<div>
  {isLoggedIn ? <LogoutButton /> : <LoginButton />}
</div>
```

Можно найти множество примеров использования тернарных операторов в известных репозиториях, например в Redux(https://github.com/reactjs/redux/blob/world/src/components/List.js#L25), где он используется для показа разного текста в зависимости от того, грузятся ли данные по сети:

```
<button [...]>
  {isFetching ? 'Loading...' : 'Load More'}
</button>
```

Посмотрим, что произойдет, если логическое выражение будет становиться сложнее и в нем будут задействованы разные переменные и логические операции:

```
<div>
  {dataIsReady && (isAdmin || userHasPermissions) &&
    <SecretData />
}
</div>
```

Это решение все еще может быть неплохим, но читаемость его начинает падать. Для решения этой проблемы можно создать вспомогательную функцию внутри компонента и использовать ее название для пояснения логики, сокрытой в теле:

```
canShowSecretData() {
  const { dataIsReady, isAdmin, userHasPermissions } = this.
     props
  return dataIsReady && (isAdmin || userHasPermissions)
}
<div>
  {this.canShowSecretData() && <SecretData />}
</div>
```

Код стал более читаемым, и даже если вернуться к нему через полгода, достаточно будет прочесть название функции и опустить чтение логики внутри.

Если вы не любите использовать функцию, то ее можно заменить геттеро, который сделат код более элегантным (прим.пер. а может быть и нет...):

```
get canShowSecretData() {
  const { dataIsReady, isAdmin, userHasPermissions } = this.
     props
  return dataIsReady && (isAdmin || userHasPermissions)
}
<div>
```

```
  {this.canShowSecretData && <SecretData />}
</div>
```

То же самое относится к вычисляемым значениям. Например, если у нас есть два значения, валюта и стоимость, и нам нужно объединить их в одну строку, то мы можем вынести эту операцию в отдельный метод.

```
getPrice() {
  return `${this.props.currency}${this.props.value}`
}
<div>{this.getPrice()}</div>
```

Это также удобнее тестировать, если внутри этого метода есть дополнительная логика.

То же самое можно сделать с помощью геттеров аналогично предыдущим примерам:

```
get price() {
  return `${this.props.currency}${this.props.value}`
}
<div>{this.price}</div>
```

Есть еще множество решений проблемы ветвления внутри React компонент, которое требует использования сторонних библиотек. В общем случае стоит аккуратно вносить новые зависимости в проект, так как они могут увеличить размер бандла, принести проблему на смене версий и увеличить порог входа в проект, но на что не пойдешь ради увеличения читаемости кода (прим.пер. используемые тут библиотеки довольно просты и имеет смысл реализовать их самостоятельно в учебных целях).

Первый вариант - использовать библиотеку render-if, которую можно установить командой:

```
npm install --save render-if
```

Мы можем легко использовать ее в проекте по аналогии с примером ниже:

```
const { dataIsReady, isAdmin, userHasPermissions } = this.
    props
const canShowSecretData = renderIf(
  dataIsReady && (isAdmin || userHasPermissions)
)
<div>
  {canShowSecretData(<SecretData />)}
</div>
```

Мы оборачиваем наше условии внутри renderIf функции.

Результатом вызова функции renderIf является функция, которая принимает аргументом React элемент, который она вернет при вызове в случае истинности условия.

Самое главное, что мы не должны забывать в данном контексте, это то, что компоненты должны оставаться простыми и глупыми настолько насколько это возможно. Иначе в этих компонентах будет сложно разбираться, править баги и расширять.

Чтобы сделать компонент чище, мы можем вынести из него логику в Компонент более высокого порядка (Higher-Order Component, HOC) с библиотекой react-only-if. Компоненты выского порядка мы рассмотрем далее в Главе 4, сейчас для нас важно только то, что HOC это функция, которая принимает аргументом компонент, расширяет его или изменяет его поведения и возвращает при вызове.

Данная библиотека устанавливается командой:

```
npm install --save react-only-if
```

После установки мы можем использовать ее внутри нашего приложения следующим образом:

```
const SecretDataOnlyIf = onlyIf(
  ({ dataIsReady, isAdmin, userHasPermissions }) => {
    return dataIsReady && (isAdmin || userHasPermissions)
  }
)(SecretData)
<div>
  <SecretDataOnlyIf
    dataIsReady={...}
    isAdmin={...}
    userHasPermissions={...}
  />
</div>
```

Как можно заметить, в этом случае внутри исходного компонента нет логики совсем, что повышает его тестируемость.

Мы передаем условие как параметр в функцию onlyIf, которая меняет поведение нашего компонента таким образом, чтобы оно отображалось только в случае истинности логического выражения.

Циклы

Отображения списков элементов - очень распространенная операция. И в данном случае JavaScript показывает себя с хорошей стороны.

Если мы поместим внутрь JSX массив с React элементами, то все они будут отрисованы на одном уровне вложенности. В целом для нас не важно, как будет получен этот массив, главное, чтобы как и любое другое выражение, было помещено в фигурные скобки.

Самым распространенным способом создать массив элементов является использование операций над множествами объектов языка JavaScript:

```
<ul>
  {users.map(user =><li>{user.name}</li>)}
</ul>
```

Этот пример прост, но показывает большую гибкость использования JSX и JavaScript для генерации HTML.

Control statements

Условные операторы и циклы часто используются для описания верстки и, возможно, вам может показаться, что вносить в JSX блоки JavaScript кода для таких базовых операция - не лучшая практика. Но JSX был разработан лишь как инструмент генерации элементов, оставляя работы с логикой программы на JavaScript.

В общем и целом не стоит держать большой объем логики внутри компонент, но тем не менее время от времени нам нужно скрывать или показывать элементы в зависимости от состояния или итерировать коллекции объектов для отображения списков.

Если вы чувствуете, что JSX должен также позволять использовать условия и циклы, и это сделает код более читаемым, то вы можете попробовать библиотеку: jsx-control-statements.

Эта библиотека не приносит никакого нового функционала в JSX и являешься лишь синтаксическим сахаром, который компилируется в JavaScript.

Прежде всего нам нужно добавить ее в проет:

```
npm install --save jsx-control-statements
```

Также его необходимо добавить в .babelrc, чтобы babel знал, что у нас появились новые правила компиляции:

```
"plugins": ["jsx-control-statements"]
```

С этого момента мы можем использовать новый синтакс и babel будет транслировать его вместе со стандартным JSX.

Условный оператор, написанный с использование этого плагина, будет выглядеть следующим образом:

```
<If condition={this.canShowSecretData}>
  <SecretData />
</If>
```

Этот код будет транслирован в обычный тернарный оператор в JavaScript:

```
{canShowSecretData ? <SecretData /> : null}
```

Для ситуации, когда нам нужно иметь возможность выбрать элемент из нескольких в зависимости от различных условий, в данной библиотеке есть компонент Choose:

```
<Choose>
  <When condition={...}>
    <span>if</span>
  </When>
  <When condition={...}>
    <span>else if</span>
  </When>
  <Otherwise>
    <span>else</span>
  </Otherwise>
</Choose>
```

Не стоит забывать, что компоненты Choose, When и Otherwise не являются React компонентами в привычном для нас понимании, это всего лишь синтаксис, который будет скомпилирован в большой набор тернарных операторов.

Также есть специальный компонент For (который также будет скомпилирован в JavaScript код) для работы с коллекциями объектов:

```
<ul>
  <For each="user" of={this.props.users}>
    <li>{user.name}</li>
  </For>
</ul>
```

Тут тоже никакой магии, после компиляции этот блок превратится в вызов метода map.

Если вы используете какой-либо линтер, то он может ругаться в последнем случае, так как переменная user по сути не определена. Это происходит из-за того, что объявление этой переменной будет сгенерировано после компиляции.

Если вы используете eslint, то для исключения данной ошибки проверки кода можно использовать библиотеку eslint- plugin-jsx-control-statements.

Если у вас еще нет опыта использования линтеров, то не беспокойтесь, они будут разобраны чуть позже.

Sub-rendering

В общем случае стоит стараться делать компоненты маленькими и простыми насколько это возможно, но тем не менее компоненты могут начать разбухать, особенно если разработка идет итеративно и функционал наращивается понемногу на каждой итерации.

Что мы можем сделать, если наши методы отображения компонент становятся слишком большими. Один из вариантов - разделить большой метод render на небольшие функции внутри одного компонента:

```
renderUserMenu() {
    // JSX for user menu
}
renderAdminMenu() {
    // JSX for admin menu
}
render() {
  return (
    <div>
      <h1>Welcome back!</h1>
      {this.userExists && this.renderUserMenu()}
      {this.userIsAdmin && this.renderAdminMenu()}
    </div>
  )
}
```

Это далеко не идеальное решение, но на практике, если нет возможности разделить компонент на более мелкие, это позволяет сохранять метод render чище.

Теперь мы должны начать чувствовать себя посвободнее в использовании JSX. Можно перейти к вопросу, как следовать единому стилю кода внутри всего проекта.

2.2 ESLint

Мы всегда пытаемся писать лучший код, на который способны, но все равно время от времени делаем ошибки и тратим часы на их поиск и исправление.

К счастью для нас, есть инструменты, которые позволяют искать ошибки еще на стадии написания кода. Такие инструменты не скажут, делает ли код то, что должен, но как минимум помогут избежать синтаксических ошибок.

Если вы пришли из мира языков со статической типизацией, таких как C#, то вы привыкли, что множество синтаксических ошибок можно обнаружить в IDE во время написания кода.

Дуглас Крокфорд (Douglas Crockford) сделал линтинг (статическую проверку кода) популярным в мире JavaScript с инструментом JSLint(первый релиз в 2002). Дальше этот инструмент перерос в JSHint, а затем в ESLint, став основным инструментом статической проверки кода в JavaScript мире в целом и в React разработке в частности.

ESLint - инструмент с открытым исходным кодом, вышедший в 2013 году и быстро набравший популярность за счет гибкости в настройке и расширении.

В мире JavaScript, где постоянно меняются библиотеки и подходы, возможность гибкой конфигурации стала одной из главных причин быстрого распространения ESLint.

Помимо этого сейчас мы транслируем код с помощью babel и используем новые возможности языка и сторонние расширения, такие как JSX. Плюс ESLint в том, что он позволяет дописывать расширения для проверки любого нового синтаксиса.

Помимо этого, так как ESLint позволяет создавать правила для проверки синтаксиса, у нас появляется возможность определить единый стиль кода внутри больших команд.

Установка

Прежде всего нам нужно установить ESLint:

```
npm install --global eslint
```

После этого мы можем запустить его следующей командой:

```
eslint source.js
```

На выходе мы получим информацию об ошибках внутри файла с кодом.

Но при первом запуске мы не должны увидеть никаких ошибок, потому ESLint не содержит никаких правил проверки по умолчанию.

Настройка

Для настройки ESLint используется файл *.eslintrc* в домашней директории проета или пользователя.

Начнем с простого и запретим использовать символ точки с запятой. Для этого добавим в .eslintrc следующий JSON:

```
{
  "rules": {
    "semi": [2, "never"]
  }
}
```

Эта запись определенно требует некоторых пояснений: "*semi*" - название правила(rule), а [2,"*never*"] - его значение.

В ESLint каждому правилу можно задать один из 3 уровней строгости:

- **off (0): Правило выключено**
- **warn (1): Правило предупреждения**
- **error (2): Правило ошибки**

Таким образом, указав в нашем примере значение 2, мы говорим, что ESLint, при срабатывании этого правила, должен ругаться о наличии ошибки в коде.

Второй параметр ("*never*") говорит о том, что точка с запятой никогда не должна использоваться внутри проекта.

ESLint и его плагины хорошо документированы и можно всегда посмотреть, как именно работают правила, и что значат их аргументы.

Теперь создадим файл index.js со следующим содержимым:

```
var foo = 'bar';
```

(Мы используем *var* так как ESLint еще не знает о том, что мы собираемся использовать ES2015).

Если мы запустим $eslintindex.js$, то получим следующее сообщение:

Extra semicolon (semi)

Заработало, теперь мы можем добавлять правила и следовать (или не следовать) им внутри проекта.

Мы можем добавлять все правила вручную или включить рекомендованный набор правил одной строкой в .eslintrc:

```
{
  "extends": "eslint:recommended"
}
```

После этого каждое отдельное правило может быть изменено по необходимости вручную.

После применения рекомендованных правил мы должны перестать получать ошибку о наличии точки с запятой (т.к. это правило не входит в рекомендованные), но должны получить ошибку, что переменная foo объявлена, но никогда не используется.

Правило $no-unused-vars$ очень полезно для сохранения чистоты кодовой базы.

Вспомним, что мы хотим писать код на ES2015, но если мы изменим исходный код на следующий:

```
const foo = 'bar'
```

То получим следующую ошибку:

Parsing error: The keyword 'const' is reserved

Для того, чтобы включить поддержку ES2015, мы должны добавить информацию об этом в .eslintrc:

```
"parserOptions": {
  "ecmaVersion": 6,
}
```

Также для нас будет полезно указать, что мы используем JSX:

```
"parserOptions": {
  "ecmaVersion": 6,
  "ecmaFeatures": {
    "jsx": true
  }
},
```

Если вы уже писали приложения на React, но не использовали линтер, то хорошим упражнением будет добавить ESLint в уже существующий проект и исправить все ошибки, которые будут найдены.

Использование ESLint из консоли конечно позволяет нам проверять код, но еще одним плюсом этого инструмента является то, что он поддерживается большинством современных редакторов и IDE (SublimeText, Atom и множество других).

Очень часто есть сильный соблазн просто проигнорировать все, что говорит нам ESLint и залить код в общий репозиторий. Чтобы избежать этого, имеет смысл добавить линтер как один из шагов в процесс сборки приложения, в этом случае код с ошибками просто не попадет в продакшн.

Еще вариант, добавить линтер на этап создания пулл реквета. В этом случае код с ошибками не попадет даже на ревью кода.

React плагин

Как было сказано раньше, одна из сильнейших сторон ESLint - его расширяемость плагинами. Самый важный плагин для нас eslint-plugin-react.

В целом ESLint может работать с JSX без дополнительных плагинов. Но мы хотим больше, например хранить наши компоненты в одном определенном стиле.

Прежде всего нам нужно установить плагин:

```
npm install --global eslint-plugin-react
```

После этого мы можем добавить его в наш файл с настройками:

```
"plugins": [
  "react"
]
```

По аналогии с самим ESLint мы можем добавить набор рекомендованных настроек для react плагина:

```
"extends": [
  "eslint:recommended",
  "plugin:react/recommended"
],
```

С этого момента, если что-то будет не так с нашими компонентами, например мы попробуем передать один и тот же параметр дважды, ESLint будет предупреждать об ошибке:

```
<Foo bar bar />
```

И соответствующее сообщение об ошибке:

No duplicate props allowed (react/jsx-no-duplicate-props)

Есть множество правил, которые мы можем использовать в нашем проекте. Давайте посмотрим, как некоторые из них могут улучшить нашу жизнь.

Одна из проблем, которую может помочь решить для нас ESLint, - это одинаковый размер отступов в JSX верстке. Это поможет нам сохранить единый стиль внутри всего приложения и не держать постоянно в голове точный размер отступов.

Для того, чтобы включить эту проверку, достаточно добавить следующее правило:

```
"rules": {
  "react/jsx-indent": [2, 2]
}
```

Первая 2 означает, что ESLint будет говорить об ошибке, если сработает это правило, вторая 2 говорит от том, что размер отступа для каждого компонента должен быть из двух пробелов. Также можно сказать, что отступов не должно быть вовсе, заменив вторую 2 на 0.

Создайте файл с содержимым вида:

```
<div>
<div />
</div>
```

И вы получите ошибку:

Expected indentation of 2 space characters but found 0 (react/jsx-indent)

Аналогичным образом мы можем заставить всех использовать одинаковый отступ для всех параметров элементов:

```
"react/jsx-indent-props": [2, 2]
```

Также часто возникают вопросы, на которые у каждого разработчика может найтись свое мнение отличное от других. Например, какой максимальной длины должны быть строки кода? Или когда считать, что у элемента слишком много параметров? С ESLint и правилом jsx-max-props-per-line можно выбрать единое значение для всех, чтобы не возвращаться к этому вопросу на каждом ревью кода.

React плагин для ESLint позволяет проверять не только JSX, но и сами компоненты.

Например, мы можем договориться определять PropTypes в алфавитном порядке. Есть правило, чтобы проверить, что используются только объявленные в PropTypes параметры. Или правило для проверки того, что все компоненты без состояния объявлены в функциональном стиле. А также множество других правил.

Airbnb configuration

Мы уже посмотрели, как ESLint помогает искать ошибки и следовать единому стилю кода. Также мы увидели, насколько ESLint открыт для настройки и расширения.

Можно сделать шаг дальше.

Через атрибут extends в файле настроек ESLint можно подключить конфигурации сторонних организаций, а затем уже добавлять свои правила поверх них.

Одна из самых известных конфигураций для ESLint в мире React была создана в стенах компании Airbnb. Разработчики этой компании создали набор правил, чтобы следовать единому стилю кода среди всех разработчиков, и каждый желающий может подключить этот набор правил в свой проект.

Для того, чтобы сделать это необходимо добавить несколько зависимостей:

```
npm install --global eslint-config-airbnbeslint@^2.9.0 eslint
    -plugin-jsx-a11y@^1.2.0 eslint-plugin-import@^1.7.0 eslint
    -plugin-react@^5.0.1
```

А затем добавить новые настройки в .eslintrc:

```
{
  "extends": "airbnb"
}
```

Это один из самых простых и распространенных способов начать работать с ESLint.

2.3 Основы функционального программирования

Кроме исправления JSX и использования линтера есть еще один способ сделать код чище: следовать **Функциональному (Functional Programming, FP)** стилю.

Функциональное программирование - парадигма декларативного программирования, сосредоточенная на минимизации побочных эффектов (side effects) и сохранении данных неизменяемыми (immutable).

Следующая часть не ставит своей целью полностью раскрыть обширную тему функционального программирования, но мы можем посмотреть на некоторые концепции, которые часто используются в React.

Объект первого класса

В JavaScript функции являются *объектами первого класса (first-class objects)*, т.е. они могут быть присвоены переменным как значение и переданы как аргументы другим функциям.

Это позволяет нам ввести концепцию **Функций высшего порядка (Higher-order Functions, HoF)**. Такой функцией мы будем назвать функцию, которая принимает аргументом функцию (и возможно другие аргументы) и возвращает новую функцию. Возвращаемая функция чаще всего расширяет функционал изначальной функции.

Посмотрим на простой пример, функцию сложения двух чисел, которую мы обернем функцией, логирующей все аргументы функции и вызывающей изначальную функцию сложения:

```
const add = (x, y) => x + y

const log = func => (...args) => {
  console.log(...args)
  return func(...args)
}
const logAdd = log(add)
```

Функции высшего порядка часто используется в React разработке. Один из самых распространенных паттернов использования этого приема - **Компоненты высшего порядка (Higher-order Components, HoC)**. Подробнее на компоненты высшего порядка мы посмотрим в Главе 4.

Purity

Важный аспект функционального программирования - чистые функции. С ними вы будете встречаться очень часто, особенно в таких библиотеках как Redux.

Главное отличие чистой функции - отсутствие побочных эффектов.

К примеру, если функция меняет состояние приложения, меняет значение переменных во внешней области видимости переменных или меняет значение переменных переданных по ссылке, то функция не является чистой.

Чистые функции значительно проще тестировать так как при вызовах с одинаковыми аргументами функция возвращает одинаковый результат.

Простой пример чистой функции - функция сложения:

```
const add = (x, y) => x + y
```

Она может быть запущена множество раз с одними и те ме же аргументами, но в каждый раз она вернет одинаковый результат.

В следующем примере функция перестает быть чистой:

```
let x = 0
const add = y => (x = x + y)
```

Если мы вызовем $add(1)$ дважды, то сначала мы получим 1, а затем 2. Причина в том, что работа функции зависит от переменной во внешней области видимости, которая еще и редактируется внутри этой функции.

Immutability

Мы разобрались с чистыми функциями, которые не изменяют состояние приложения, но что если в функцию приходит сложный объект, в котором мы хотим что-либо изменить?

FP говорит нам о том, что в этом случае следует создать новый объект с измененным значением и вернуть его, не трогая исходный объект.

Рассмотрим пример:

```
const add3 = arr => arr.push(3)
const myArr = [1, 2]
add3(myArr) // [1, 2, 3]
add3(myArr) // [1, 2, 3, 3]
```

Эта функция изменяет состояние исходного массива, что противоречит парадигме неизменяемости. Если вызвать эту функцию несколько раз с одним и тем же исходными массивом, то мы будем получать разные значения на выходе.

Мы можем исправить эту функцию, чтобы она стала immutable с помощью метода concat, который возвращает новый массив, оставляя исходный без изменений:

```
const add3 = arr => arr.concat(3)
const myArr = [1, 2]
const result1 = add3(myArr) // [1, 2, 3]
const result2 = add3(myArr) // [1, 2, 3]
```

Или используя спред оператор:

```
const add3 = arr => [...arr, 3]
```

Сколько бы мы ни вызывали этот метод, исходный массив остается нетронутым.

Каррирование

Каррирование (Curring) - еще одна распространенная техника функционального программирования, которая заключается в конвертации функции от многих переменных в функцию одной переменной, которая возвращает другую функцию.

Посмотрим на примере функции *add* как это работает на практике. Вместо исходной функции от двух аргументов:

```
const add = (x, y) => x + y
```

Мы можем определить каррированную функцию:

```
const add = x => y => x + y
```

Такую функцию мы можем использовать следующим образом:

```
const add1 = add(1)
add1(2) // 3
add1(3) // 4
```

Таким способом мы можем заключить первый аргумент внутри переменной $add1$ и использовать ее множество раз без явной передачи.

Композиция

Композиция (Composition) - еще один прием, который позволяет сохранять функции небольшими и тестируемыми.

Рассмотрим следующий пример:

```
const add = (x, y) => x + y
const square = x => x * x
```

Эти функции могут быть скомбинированы вместе, чтобы создать новую функцию, которая складывает два числа и возвращает их квадрат:

```
const addAndSquare = (x, y) => square(add(x, y))
```

Следуя этому простому приему можно сохранять функции небольшими и читаемыми, а также упростить их тестирование.

FP и пользовательские интерфейсы

Последний шаг - понять, как функциональное программирование помогает нам строить пользовательский интерфейс.

Мы можем думать о UI как о функции, которая принимает аргументом состояние приложения, а возвращает интерфейс приложения:

$$UI = f(state)$$

Мы хотим ожидать, что эта функция будет чистой, т.е. для одного и того же состояния приложения она будет возвращать всегда одинаковое представление для пользователя.

В React мы будем будем рассматривать компоненты как функции и комбинировать из этих функций пользовательский интерфейс.

Есть много схожего между функциональным программирование и созданием пользовательского интерфейса с React. И чем больше мы будем смотреть на разработку с React как на функциональное программирование, тем лучше будет наш код.

2.4 Заключение

В этой главе мы детально разобрали строение JSX и как с помощью него создавать компоненты.

Также мы разобрались со статической проверкой кода средствами ESLint и его плагинов. Это позволит нам быстрее находить потенциальные проблемы в коде.

И в конце мы рассмотрели базовые концепции функционального программирования, которые часто встречаются в React.

Используя все выше сказанное, можно сделать код значительно чище и более тестируемым. Настало время сделать еще один шаг вперед и разобраться, как сделать действительно переиспользуемые компоненты.

Глава 3

Создаем реально переиспользуемые компоненты