Глава 1

Собираем все в кучу

В предыдущей главе мы разобрались, как создавать переиспользуемые компоненты. Теперь мы можем поговорить о том, как заставить это компоненты эффективно взаимодействовать друг с другом.

Сильной стороной React является то, что он позволяет создавать сложные интерфейсы комбинирование маленьких, тестируемых компонент. Этот подход позволяет контролировать каждый аспект приложения.

В этой главе мы рассмотрим самые распространенные паттерны и инструменты для комбинирования компонент.

Мы обсудим следующие вопросы:

- Как компоненты коммуницируют друг с другом посредство передачи *props* дочерним элементам
- Как паттерн Контейнер и Представление помогает писать более поддерживаемый код
- Проблему, которую пытались решить миксины (mixins), но не смогли
- Улучшение структуры приложения с Компонентами Высшего порядка
- Библиотеку recompose и ее встроенные функции
- Как мы можем взаимодействовать с контекстом и как избежать сильной связности компонентов с ним

- Паттерн Function as a Child и какую пользу он может принести

1.1 Взаимодействие компонентов

Переиспользуемые компоненты могут использоваться внутри множества других компонент в процессе разработки вашего приложения.

Небольшие компоненты с простым интерфейсом могут составлять более сложные компоненты, которые в свою очередь являются частью еще более сложных компонент и приложения в целом.

Мы уже неоднократно видели, как в React объединяются компоненты. Для этого достаточно описать структуру из вложенных компонент внутри метода *render*:

```
const Profile = ({ user }) => (
  <div>
    <Picture profileImageUrl={user.profileImageUrl} />
    <UserName name={user.name} screenName={user.screenName}
       />
  </div>
)
Profile.propTypes = {
  user: React.PropTypes.object,
}
```

Например вы можете создать компонент $Profile$ путем комбинирования компонентов $Picture$ для отображения изображения профиля и $UserName$ для имени пользователя.

Таким образом вам требуется всего нескольких строчек кода для добавления новых блоков интерфейса.

После того как вы объединили компоненты как на примере выше, вы можете передавать между ними данные, используя *props*.

Props - основной способ передачи данных от родительских компонент дочерним в React.

Когда компонент передает данные другому компоненту, он является **Владельцем (Owner)** этого компоненты, не зависимо от иерархической принадлежности каждого из них.

Например, в последнем примере $Profile$ не является непосредственным родителем $Picture$ (между ними еще тег *div*), но $Profile$ является владельцем $Picture$, так как передает ему данные через параметры

(прим.пер. далее я все равно буду называть такие компоненты родительскими, просто в чуть более обобщенном значение).

Children

Есть специальный параметр **children**, который передается от родительского компонента дочерним и доступен в методе render.

В документации React говорится, что это *непрозрачный (opaque)* параметр, так как он не несет никакой информации о том, что именно внутри него содержится.

Вложенные компоненты, определенные в методе *render*, обычно получают параметры через атрибуты в JSX (или через второй аргумент метода *createElement*).

Также компонент можно определить с вложенными компонентами, в этом случае они будут доступны для него через параметр *children*.

Представим, что у нас есть компонент *Button*, у которого есть параметр *text*, отвечающий за текст на кнопке:

```
const Button = ({ text }) => (
  <button className="btn">{text}</button>
)
Button.propTypes = {
  text: React.PropTypes.string,
}
```

Этот компонент можно использовать следующим образом:

```
<Button text="Click me!" />
```

Теперь предположим, что мы хотим использовать ту же самую кнопку с тем же className, но отображать внутри нее что-то более сложное чем просто текст.

Что если мы хотим, чтобы у нас были кнопки с текстом, кнопки с изображением и кнопки с текстом и заголовком?

В множестве случаем достаточным решением будет добавить множество параметров в компонент *Button* или создать специализированные компоненты, например *IconButton*.

Однако, если мы понимаем, что *Button* всего лишь обертка, которая должна отображать любое содержимое, то мы можем использовать параметр *children*.

Мы можем легко поправить предыдущий вариант *Button*, чтобы иметь возможность отображать любое содержимое:

```
const Button = ({ children }) => (
  <button className="btn">{children}</button>
)
Button.propTypes = {
  children: React.PropTypes.array,
}
```

Теперь мы можем использовать любые компоненты внутри *Button*, они будут подставлены вместо *children* в JSX.

Например мы можем создать кнопку с изображением и текстом внутри:

```
<Button>
  <img src="..." alt="..." />
  <span>Click me!</span>
</Button>
```

В этом случае мы получим следующий HTML код:

```
<button className="btn">
  <img src="..." alt="..." />
  <span>Click me!</span>
</button>
```

Это очень удобный способ, чтобы позволить компонентам принимать любые дочерние элементы и оборачивать их предопределенным образом.

Как вы могли заметить в предыдущем примере, мы определили параметр *children* как массив, что значит, что можно передать любое количество элементов.

Но если мы передадим только один элемент, например:

```
<Button>
  <span>Click me!</span>
</Button>
```

то получим следующую ошибку:

> **Failed prop type: Invalid prop ‘children‘ of type ‘object‘ supplied to ‘Button‘, expected ‘array‘.** (Неверный тип параметра: неверный тип параметра ’children’ с типом ’объект’, переданный компоненту *Button*; ожидается ’массив’)

Это происходим из-за того, что в случае с передачей одиночного элемента React оптимизирует выделение памяти и используем сам элемент вместо создания массива с одним элементом.

Мы можем легко это поправить, указав в propTypes не только массив, но и одиночный элемент:

```
Button.propTypes = {
  children: React.PropTypes.oneOfType([
    React.PropTypes.array,
    React.PropTypes.element,
  ]),
}
```

1.2 Паттерн Контейнер и Представление

В этой главе мы рассмотрим паттерн, который поможет сделать наш код еще чище и более поддерживаемым.

Как правило React компоненты представляют из себя сочетание **логики** и **отображения**.

Под логикой мы понимаем все, что не относится к UI, т.е. такие вещи как обращения к API сервера, преобразование данных и обработку событий.

А под представлением наоборот, ту часть, которая отвечает за создание элементов для UI. Прежде всего это содержимое метода *render*.

В React есть простой и мощный паттерн, **Контейнер и Представление (Container and Presentational)**, который помогает разделить по отдельным компонентам две эти составляющие.

Заодно посмотрим в этой главе, какая еще польза, помимо переиспользуемости компонент, может быть от разделения логики и представления.

Как всегда начнем изучения паттерна с примера, в котором он используется.

Предположим, у нас есть компонент, который получает из API геолокации долготу и широту, а затем отображает их на экране.

Для начала создадим файл *geolocation.js* и определим в нем компонент *Geolocation*:

```
class Geolocation extends React.Component
```

В этом компоненте создадим конструктор для определения начального состояния и привязки обработчиков событий:

```
constructor(props) {
  super(props)
```

```
  this.state = {
    latitude: null,
    longitude: null,
  }
  this.handleSuccess = this.handleSuccess.bind(this)
}
```

Теперь в *componentDidMount* мы можем осуществить вызов API:

```
componentDidMount() {
  if (navigator.geolocation){
    navigator.geolocation.getCurrentPosition(this.
        handleSuccess)
  }
}
```

После того, как компонент получит данные от сервера, их можно сохранить во внутреннее состояние компонента:

```
handleSuccess({ coords }) {
  this.setState({
    latitude: coords.latitude,
    longitude: coords.longitude,
  })
}
```

И в конце концов мы можем отобразить высоту и широту на экране через метод *render*:

```
render() {
  return (
    <div>
      <div>Latitude: {this.state.latitude}</div>
      <div>Longitude: {this.state.longitude}</div>
    </div>
  )
}
```

Важно заметить, что после первой отрисовки компонента значение долготы и широты равно *null*, так как запрос на данные асинхронен и лишь инициализируется в *componentDidMount*. В реальном проекте вы скорее всего захотите в этот момент показывать какой-то индикатор загрузки, что можно сделать с помощью условных операторов, подробно разобранных в Главе 2.

Но в целом в этом компоненте нет никаких проблем и он прекрасно работает.

Предположим, что мы работаем с дизайнером над UI составляющей компонента. Не было бы это хорошей идеей, создать компонент, состоящий только из UI части, чтобы иметь возможность быстрее обсудить ее с дизайнером.

Если мы отделим представление от логики, то мы сможем без проблем добавить его в документацию на основе **Storybook**, как мы делали в одной из предыдущих глав.

Как вы уже можете догадаться, использование паттерна Контейнер и Представление предполагает разделение компонента на два, с более четкой зоной ответственности у каждого.

Если быть более точным, то в Контейнере находится вся логика и работа с данными, а в Представлении создание элементов и минимум логики. Чаще всего компонент представления может быть выражен компонент-функцией.

Но это не значит, что в Представлении не может быть состояния вообще. В некоторых случаях, например при создании полей ввода, может быть уместнее хранить состояние в компоненте представления.

В случае нашего примера мы отображаем на экране лишь широту и долготу, поэтому мы воспользуемся компонент-функцией для создания Представления.

Для начала переименуем наш компонент *Geolocation* в *GeolocationContainer*:

```
class GeolocationContainer extends React.Component
```

А также переименуем файл, содержащий этот компонент, из *geolocation.js* в *geolocation − container.js*.

Такой вариант наименования не высечен в камне, но является наиболее распространенным в сообществе React. К компоненты Контейнера мы добавляем в конце *Container*, а компоненте Представления оставляем оригинальное имя.

Также нам нужно изменить реализацию метода *render*, заменив все содержимое отрисовкой одного компонента:

```
render() {
  return (
    <Geolocation {...this.state} />
  )
}
```

Таким образом, вместо отрисовки HTML элементов мы просто отображаем компонент Представления и передаем в него свое состояние.

В **состоянии** нашего компонента высота и широта, которые по умолчанию имеют значение *null* и меняются на координаты пользователя через **обратный вызов (calback)** после вызова API.

Чтобы передать состояние целиком, мы используем спред оператор (spread operator), который избавляет нас от необходимости указывать параметры один за другим вручную.

Теперь создадим файл *geolocation.js*, в котором создадим компонент Отображения:

```
const Geolocation = ({ latitude, longitude }) => (
  <div>
    <div>Latitude: {latitude}</div>
    <div>Longitude: {longitude}</div>
  </div>
)
```

Компонент-функции - очень лаконичный способ описания интерфейса. Чистые функции однозначно отображают состояние в набор элементов.

В нашем случае компонент принимает через *state* долготу и широту и отображет их внутри *div* элементов.

Мы хотим следовать лучшим практикам, поэтому определим необходимый и достаточный интерфейс для этого компонента:

```
Geolocation.propTypes = {
  latitude: React.PropTypes.number,
  longitude: React.PropTypes.number,
}
```

Следуя паттерну Контейнер и Представление, мы сорздаем глупые компоненты, которые потом можно использовать в Style guide с искусственными данными.

Если в нашем приложении в другом месте будет предполагаться такой же визуальный компонент, то нам не придется создавать компонент с нуля. Нам будет достаточно создать новый Контейнер для существующего представления, например если нам нужно будет загрузить координаты из другого сервиса.

Также другим членам команды будет проще расширить логику в контейнере, например добавить обработку ошибок, не затрагивая представления.

Также можно создать временный компонент с отображением отладочной информации для скорейшей реализации логики.

Также такой подход позволяет разделить создание компонента между разными людьми, что особенно полезно в случае больших команд и итеративных процессов разработки.

Это очень простой в использовании и полезный на практике паттерн. В большой команде он способен значительно увеличить скорость разработки и поддерживаемость написанного кода.

Но с другой стороны, использование этого паттерна без явной необходимости может значительно увеличить количество файлов и размер кодовой базы.

Не стоит начинать делить все компоненты на два сломя голову. Чаще всего стоит начинать рефакторить компонент посредством разделения логики и представления, когда они начинают быть сильно связанными.

Например в нашем примере, мы предположили, что у нас может появиться другой источник данных, для которого мы и будем создавать отдельный компонент.

Не всегда можно однозначно понять, что должно быть в Контейнере, а что в Представлении. Следующий список утверждений должен помочь вам в сложной ситуации:

Компонент Контейнера:

- Сосредоточен больше на поведении
- Отображает компонент Представления
- Выполняет асинхронные запросы к серверу и преобразует данные
- Определяет обработчики событий
- Создаются как наследуемые от React.Component классы

Компонент Представления:

- Сконцентрированы на визуальной составляющей
- Отображают HTML разметку (и другие компоненты)
- Получают данные от родительского компонента через *props*
- Часто определяются через компонент-функции без состояния

1.3 Mixins

Компоненты отлично служат цели достижения переиспользуемости кода, но что если у нас появляется множество различных компонент, которые должны обладать общими чертами?

Очевидно, мы не хотим дублировать код, к счастью React предоставляет специальный инструмент для решения этой проблемы: **примеси (mixins)**.

В общем и целом примеси не рекомендуются к использованию, но все равно стоит знать, какие проблемы они решают и какие есть альтернативы.

Также есть не нулевая вероятность, что вас может занести на проект с кучей старого кода, где могут во всю применяться примеси, поэтому быть готовым к такому повороту лишним не будет.

Начать стоит с того, что примеси работают только с *createClass*, что является одной из причин предания их забвению.

Предположим, что вы используете *createClass* и понимаете, что вам нужно написать один и тот же код в разные компоненты.

Например, вам нужно подписаться на событие изменения размера экрана и выполнять по нему какой-то код.

Собственно его можно написать один раз и передавать через примеси в любые компоненты. Посмотрим на примере кода.

Точкой соприкосновения компонента и примеси обычно выбирается *state*. Мы можем выделить в *state* конкретное поле и использовать его и из компонента и из примеси. В остальном примесь описывается как обычный самостоятельный компонент.

Определим в нашей примеси начальное состояние с помощью метода *getInitialState*, в котором будет одно поле *innerWidth*:

```
getInitialState() {
  return {
    innerWidth: window.innerWidth,
  }
},
```

Теперь мы можем начать отслеживать изменения размера экрана, для чего подпишемся на соответсвующее событие:

```
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
},
```

Также мы хотим удалить этот обработчик события перед удалением компонента, чтобы избежать накопления неиспользуемых обработчиков в объекте *window*:

```
componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
},
```

И осталось только создать функцию, которая будет вызваться на каждом изменении размера экрана.

В этой функции мы будем обновлять значение поля $innerWidth$ в *state* актуальным значением, так что любой компонент, который использует эту примесь, будет перерисован как после собственного $setState$:

```
handleResize() {
  this.setState({
    innerWidth: window.innerWidth,
  })
},
```

Как видно из примера, создание примеси почти не отличается от создания обычного компонента.

Чтобы использовать эту примесь вместо с компонентом, достаточно добавить ее в массив *mixins* внутри компонета:

```
const MyComponent = React.createClass({
  mixins: [WindowResize],
    render() {
      console.log('window.innerWidth', this.state.innerWidth)
      ...
  },
})
```

С этого момента значение $innerWidth$ будет доступно не только в примеси, но и в компоненте, который будет перерисовывать при каждом обновлении состояния из примеси.

Само собой мы можем использовать одну и ту же примесь в множестве компонент, также и внутри одного компонента может использоваться сразу множество примесей.

Очень полезной особенностью примесей является то, что они обладают одинаковым с компонентами жизненным циклом, а также возможностью задать состояние по умолчанию.

Например, если мы используем $WindowResize$ в компоненте, в котором уже есть $componentDidMount$, то никаких коллизий не произойдет,

и оба метода выполнятся.

Теперь посмотрим, в чем проблемы примесей и почему от них отказались. А в следующей части разберемся, как достигнуть такого же результата другими средствами.

Во первых примеси часто используют внутренние функции для взаимодействия с компонентом.

Например, наша примесь $WindowResize$ может ожидать, что функция обратного вызова $handleResize$ будет реализована внутри компонента, что даст возможность разработчикам большую свободу в обработке изменения размера экрана.

Или наоборот, примесь хочет получать данные из компонента и дергает специальный метод, что-то вроде $getInnerWidth$. Само собой этот метод тоже должен быть реализован внутри компонента.

К сожалению, нет никакой возможности получить точный список методов, которые должны быть реализованы внутри компонента при добавлении примеси.

Такой подход очень сильно ухудшает поддерживаемость кода. Если компонент использует множество примесей, то при их удалении или изменении очень сложно выделить код, которые также может быть удален или требует модификации.

Также частая проблема - конфликты имен. Очень часто примеси могут начать требовать функции или атрибуты с одинаковыми названиями. React без проблем разделяет вызовы методов жизненного цикла компонент, но совсем ничего не может сделать с вызовами пользовательских функций.

Таким образом примесям остается использовать внутреннее состояние компонент, что не очень хорошо, так как мы пытаемся наоборот сократить его использование с целью повышения переиспользуемости.

Помимо этого, может начать складываться ситуация, когда одни примеси начинают зависеть от других. Например, мы можем создать еще одну примесь **ResponsiveMixin**, которая будет скрывать некоторые элементы с экрана в зависимости от текущего размера экрана, который мы получаем из примеси $WindowResize$.

Такая тесная связь примесей значительно усложняет отладку приложения и его масштабируемость.

1.4 Компоненты высшего порядка

В прошлой части мы посмотрели, как примеси помогают избежать дублирования кода при создании общего для компонент функционала, и какие проблемы это приносит.

Когда мы говорили о функциональном программировании в Главе 2, мы упоминали концепцию **Функций высшего порядка (Higher-order Functions, HoFs)**. Такая функция принимает аргументом другую функцию и возвращает ее с измененным поведением.

Посмотрим, можем ли мы применить этот подход к React компонентам и достигнуть цели переиспользования функционала множеством компонент.

В случае применения данной концепции к компонентам React они станут называться **Компонентами высшего порядка (Higher-order Components, HoCs)**

Структура любого HoC выглядит следущим образом:

```
const HoC = Component => EnhancedComponent
```

Компонент высшего порядка - это функция, которая принимает аргументом React компонент и возвращает его с расширенным функционалом.

Давайте начнем с простого примера, чтобы как это все выглядит на практике.

Предположим, что вам по какой-то причине необходимо добавить к множеству компонент один и тот же *className*. Никто не запрещает обойти все компоненты и в каждом поправить метод *render*, а можно создать один HoC, который решит нашу проблему:

```
const withClassName = Component => props => (
  <Component {...props} className="my-class" />
)
```

Если вы впервые встречаете эту концепцию, может быть не очевидно, как работает этот код, поэтому давайте детально разбираться, что тут происходит.

Мы определили функцию *withClassName*, которая принимает аргументом компонент *Component* и возвращает другую функцию.

Эта созданная функция есть обыкновенная компонент-функция, которая принимает аргументом параметры *props* и возвращает компонент

Component, передавая ему с помощью спред оператора все параметры и в дополнение к ним параметр *className* со значением "*my − class*".

Чаще всего HoC передают параметры дальше через спред оператор. Это делается для того, чтобы HoC меньше зависел от изменения API компонента, а также чтобы только добавлять поведение и минимально затрагивать поведение самого компонента.

Это очень простой пример, который скорее всего никогда не пригодился бы в реальном проекте, но на нем мы посмотрели как выглядит HoC и как его можно создать.

Теперь посмотрим, как *withClassName* можно использовать с другими компонентами.

Прежде всего создадим компонент, который принимает в параметрах *className* и добавляет его к *div* элементу:

```
const MyComponent = ({ className }) => (
  <div className={className} />
)
MyComponent.propTypes = {
  className: React.PropTypes.string,
}
```

Но вместо того, чтобы использовать этот компонент напрямую, мы передадим его созданному ранее HoC'у, и по сути получим новый компонент:

```
const MyComponentWithClassName = withClassName(MyComponent)
```

Оборачивая наш компонент в *withClassName*, мы гарантируем получение компонентом параметра *className*.

Давайте теперь попробуем сделать что-то более впечатляющее и переделаем примесь *WindowResize* из предыдущей части в HoC, чтобы снова иметь возможность переиспользоввать ее в сферическом проекте в вакууме.

Напомним, что эта примесь создавала обработчик для отслеживания изменения размера экрана и сохраняла актуальное значение в поле *innerWidth* внутри состояния компонента.

Основная проблема была в том, что примесь использовала *state* компонента, чтобы передавать ему актуальные данные.

Это не очень хорошее поведение, так как могут возникнуть конфликты имен внутри состояния компонента.

Прежде всего создадим функцию, которая принимает аргументом компонент:

```
const withInnerWidth = Component => (
  class extends React.Component { ... }
)
```

Возможно вы обратили наименование HoC. Это распространенная практика начинать название с *with*, если HoC расширяет параметры, которые передаются компоненту.

Помимо этого, *withInnerWidth* будет возвращать компонент-класс, а не компонент-функцию, так как нам потребуется использовать внутреннее состояние и методы жизненного цикла.

Посмотрим, как будет выглядеть возвращенный класс.

В конструкторе мы определим начальное состояние и привяжем функцию обработчика событий к создаваемому экземпляру класса:

```
constructor(props) {
  super(props)
  this.state = {
    innerWidth: window.innerWidth,
  }
  this.handleResize = this.handleResize.bind(this)
}
```

Добавление и удаление обработчиков события изменения размера экрана и обновление внутреннего состояния аналогично уже реализованному в примеси:

```
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
}
componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
}
handleResize() {
  this.setState({
    innerWidth: window.innerWidth,
  })
}
```

И в конце нам нужно реализовать метод *render*, в котором мы должны отобразить изначальный компонент, передавая ему новые данные:

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

Можно обратить внимание, что мы через спред оператор передаем не только параметры, но также и внутреннее состояние.

По сути мы аналогично примеси храним *innerWidth* внутри состояния, но передаем его не в *state* изначального параметра, а в его *props*.

Как мы уже говорили в Главе 3, использование параметров чаще всего предпочтительнее состояния в разрезе повышения переиспользуемости компонент.

Теперь мы можем без проблем обернуть любой компонент, который ожидает параметр *innerWidth* (или не ожидает, но зачем тогда все это..) в *withInnerWidth* HoC.

Создадим для примера компонент, который получает параметр *innerWidth* и просто выводит его значение на экран:

```
const MyComponent = ({ innerWidth }) => {
  console.log('window.innerWidth', innerWidth)
  ...
}
MyComponent.propTypes = {
  innerWidth: React.PropTypes.number,
}
```

Который мы можем теперь обернуть функцией *withInnerWidth* следующим образом:

```
const MyComponentWithInnerWidth = withInnerWidth(MyComponent)
```

Есть несколько преимуществ использования этого подхода перед примесями: прежде всего мы не затрагиваем внутреннее состояние исходного компонента, а также не требуем (и не ожидаем) от него реализации каких-либо специфичных методов.

Это значит, что и исходный компонент и компонент высшего порядка не связаны, что позволяет переиспользовать их независимо друг от друга в дальнейшем.

Также передача данных через параметры позволяет уменьшить количество логики внутри исходного компонента, что упрощает его использование внутри Style Guide.

В этом случае нам достаточно создать компонент с разными размера экрана, которые мы поддерживаем внутри приложения.

Т.е. мы без проблем можем передать конкретное число в через параметры так:

```
<MyComponent innerWidth={320} />
```

Или так:

```
<MyComponent innerWidth={960} />
```

1.5 Recompose

1.5.1 Context

1.6 Function as Child

1.7 Summury