

Глава 1

Собираем все в кучу

В предыдущей главе мы разобрались, как создавать переиспользуемые компоненты. Теперь мы можем поговорить о том, как заставить эти 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 посредством 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 компонента, 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 значении).

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

В этом случае мы получим следующий HTML код:

```
<button className="btn">
  
  <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 происходит 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 (callback)** 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 отображает 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

1.4 Higher-order Components

1.5 Recompose

1.5.1 Context

1.6 Function as Child

1.7 Summury