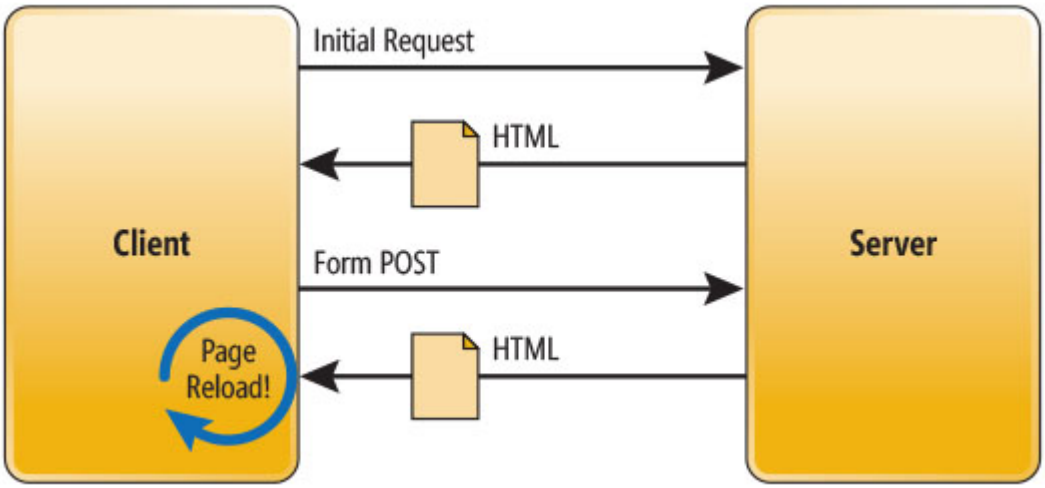
Основы. Компоненты. Props. JSX

Multiple-page application (MPA)

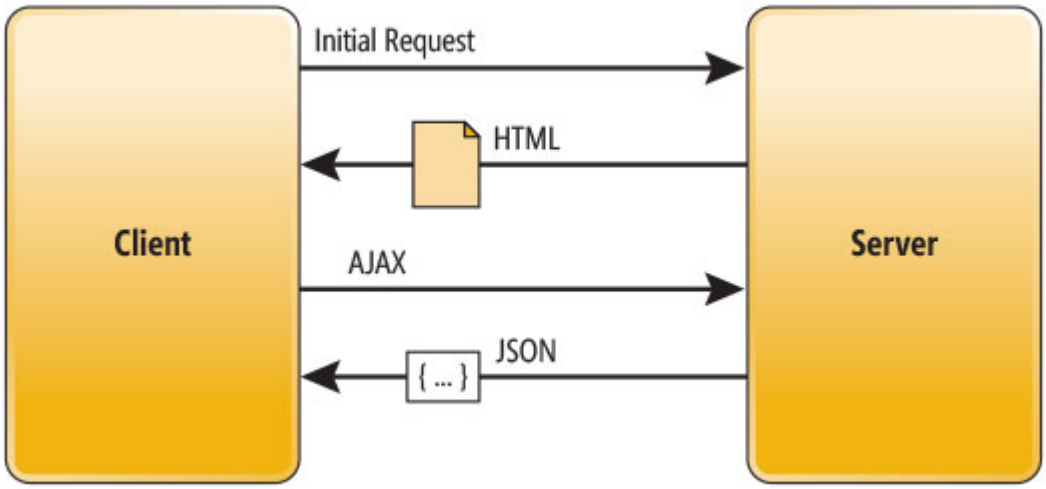
Классический веб-сайт, интернет-магазины, сайты-визитки и т. п.



- Архитектура клиент <-> сервер
- На каждый запрос сервер отсылает готовый HTML-документ
- Перезагрузка страницы при каждом запросе, раздражает
- Относительно медленно
- Отличное SEO

Single-page Application (SPA)

Одностраничное приложение это веб-сайт, интерфейс которого перерисовывается на клиенте без перезагрузки страницы, вместо запроса страниц с сервера.



- Архитектура клиент <-> сервер
- При загрузке сайта сервер отдает стартовую HTML-страницу
- Каждый последующий запрос на сервер получает только данные в JSON-формате
- Обновление интерфейса происходит динамически на клиенте

- Загрузка первой страницы может быть довольно медленной (лечится)
- Сложность кода и его поддержки масштабируется с кол-вом функционала
- Слабое SEO (лечится)

Single-page application vs. multiple-page application

Кратко о React

- **View** - простая библиотека для построения интефрейса. Не имеет встроенной маршрутизации, HTTP-модуля и т. п.
- **Экосистема** - огромная экосистема которая превращает React в полноценный фреймворк
- **Компонентный подход** - основная концепция это компоненты, маленькие строительные блоки из которых собирается интерфейс.
- **Декларативный** - позволяет декларативно описывать интерфейс как функцию от данных. React, при каждом изменении данных, обновит интерфейс.
- **Мультиплатформенный** - можно рендерить на сервере (`NodeJS`), писать нативные (`React Native`) или десктопные (`Electron`) приложения.
- **Нет манипуляции DOM** - никакой (почти) манипуляции с DOM-деревом напрямую. Вся работа с DOM выполняется внутренними механизмами React. Задача разработчика, описать интерфейс с помощью компонентов и управлять изменением данных.

Thinking in React

The DOM

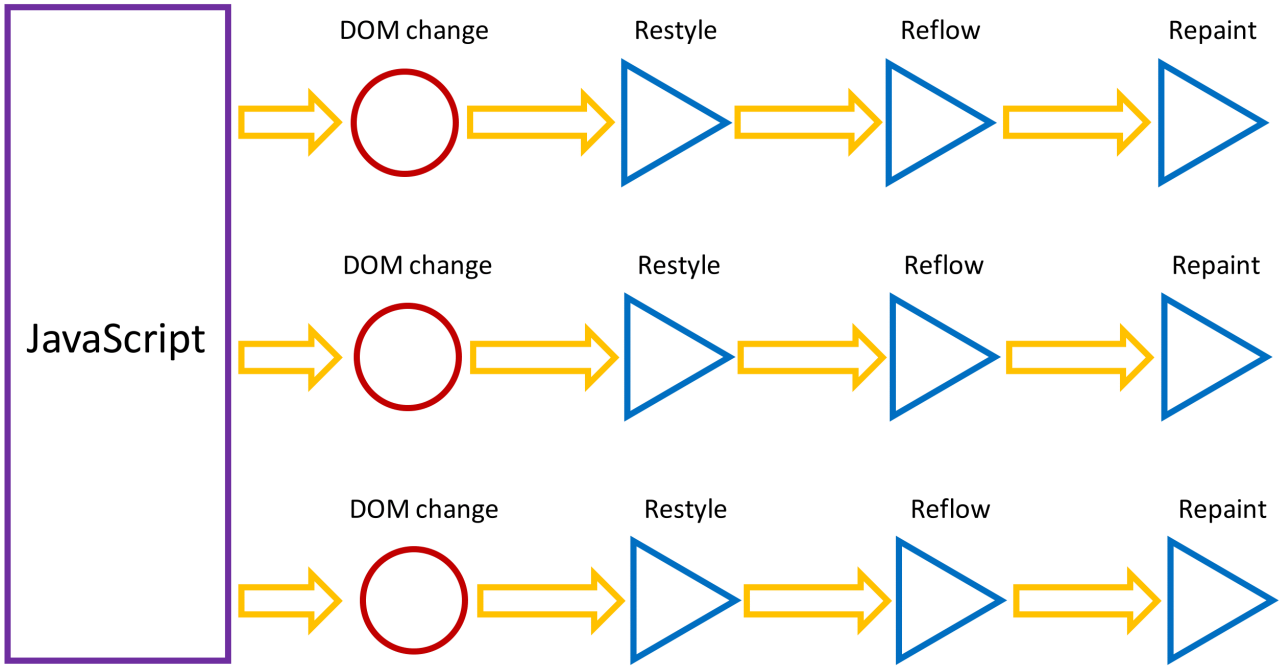
Browser DOM

DOM - древовидное представление HTML-документа. Хранится в памяти браузера и напрямую связан с тем что мы видим на странице.

- DOM всегда представлен в виде дерева. Делать поиск по нему легко, но довольно медленно.
- Обновление DOM вручную - грязная работа, так как необходимо следить за предыдущим состоянием DOM-дерева.
- DOM не предназначен для постоянного изменения, поэтому он медленный и обновлять его необходимо эффективно.

При каждом изменении DOM, браузер выполняет несколько трудоемких операций:

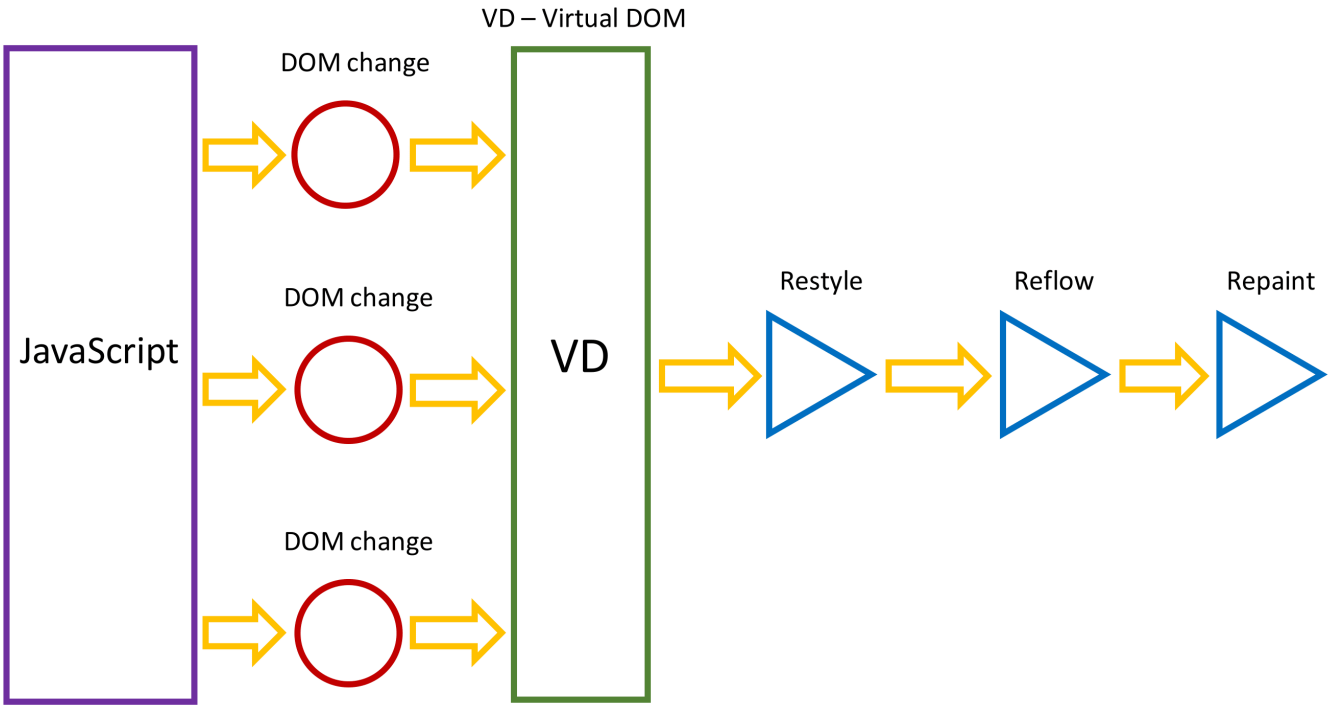
- **restyle** - изменения которые не затронули геометрию
- **reflow** - изменения которые затронули геометрию
- **repaint** - отрисовка изменений



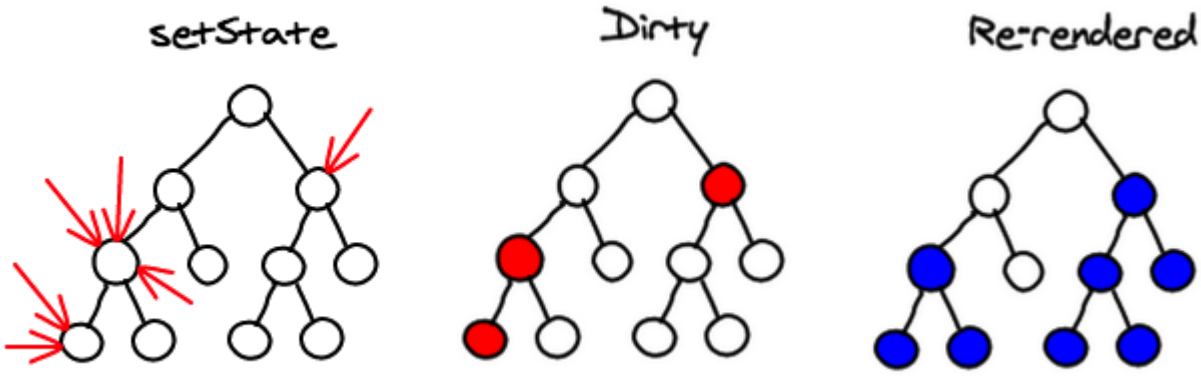
Virtual DOM

React хранит копию DOM в виде Virtual DOM - абстракции, легковесной копии реального DOM представленная JS-объектами.

- Не зависит от внутренней имплементации браузера
- Использует лучшие практики при обновлении реального DOM
- Позволяет собирать обновления в группы для оптимизации рендера



Алгоритм обновления DOM



- Взаимодействие пользователя с интерфейсом изменяет состояние приложения
- Каждый раз когда часть дерева изменяется, React помечает ее как `грязную (dirty)`
- Создается новая копия Virtual DOM
- Запускается алгоритм сравнения предыдущей и следующей версии Virtual DOM используя breadth-first алгоритм
- Происходит обновление Virtual DOM в тех местах, где дерево помечено (грязные элементы), при этом производятся внутренние оптимизации
- Базируясь на предыдущем шаге, вычисляется наименьшее необходимое количество изменний реального DOM
- Обновляется реальный DOM

Материалы

- Reconciliation
- React Reconciliation
- How Virtual-DOM and diffing works in React

Инструменты

Необходимы NodeJS, Webpack, Babel и React. Все это включает в себя `Create React App`.

- Абстрагирует всю конфигурацию, позволяя сосредоточиться на написании кода
- Включает самые необходимые и частоиспользующиеся базовые пакеты
- Имеет функцию извлечения, которая удаляет абстракцию и открывает конфигурацию
- Легко расширяется дополнительными инструментами

```
npx create-react-app my-app
```

или

```
npm init react-app my-app
```

или

```
yarn create react-app my-app
```

- Create a New React App
- Блог пост о Create React App 2.0
- Документация create-react-app
- Статья о NPX

React-элементы

- Элементы это самые маленькие строительные блоки React
- Элемент описывает то, что вы хотите увидеть на странице
- В отличии от DOM, React-элементы это обычные объекты, поэтому создавать их очень быстро
- Не путайте элементы с компонентами, элементы это то, из чего состоят компоненты

React.createElement()

Функция `React.createElement()` это самый главыный метод React API. Подобно `document.createElement()` для DOM, `React.createElement()` это функция используемая для создания React-элементов.

```
React.createElement(type, props, child, child, ...);
```

- `type` - это не имя HTML-тега, это имя встроенного React-компонента который соответсвует HTML-тегу в Virtual DOM
- `props` - объект содержащий HTML-атрибуты и кастомные свойства
- `child` - принимает произвольное количество аргументов после второго, все они описывают детей создаваемого элемента. Таким образом, фактически, создается дерево элементов

```
const element = React.createElement(
  'a',
  { href: 'https://www.google.com' },
  'Google.com',
);
```

Создадим элемент с детьми, карточку товара.

```
const product = React.createElement(
  'div',
  { className: 'product' },
  React.createElement('img', {
    className: 'image',
    src: 'https://placeimg.com/320/240/arch',
    alt: 'yummi',
  }),
```

```
  React.createElement('h2', { className: 'name' }, 'Raging waffles'),
  React.createElement('p', { className: 'price' }, '20$'),
  React.createElement(
    'button',
    { className: 'btn', type: 'button' },
    'Add to cart',
  ),
);
```

- Вызовы `React.createElement()` можно вкладывать потому что это просто JavaScript
- Второй аргумент может быть `null` или пустой объект, если нет необходимости передавать атрибуты и свойства
- React пытается максимально близко повторять нативный DOM API, поэтому для описания класса используется `className`
- Результат вызова `React.createElement()` всегда объект, элемент Virtual DOM

Рендер элемента в DOM-дерево

Для того чтобы отрендерить созданный элемент, необходимо вызвать метод `ReactDOM.render()`, который, первым аргументом принимает ссылку на React-элемент или компонент (что рендерить), а вторым, ссылку на уже существующий DOM-элемент (куда рендерить).

```
ReactDOM.render(product, document.getElementById('root'));
```

React использует модель отношений `предок - потомок`, поэтому достаточно использовать только один вызов `ReactDOM.render()`, рендер родительского элемента отобразит все вложенные в него дочерние элементы.

React Components, Elements, and Instances

JavaScript Syntax Extension

Код в предыдущем разделе - это то, что понимает браузер. Однако, для человека, описывать разметку интерфейса таким образом неудобно, нам привычен HTML. Для этого был придуман **JSX**.

- Описывает объекты, элементы Virtual DOM
- Позволяет использовать XML-образный синтаксис прямо в JavaScript
- Упрощает код, делает его декларативным и читабельным
- Это не HTML, Babel трансформирует JSX в вызовы `React.createElement()`
- Напоминает язык шаблонов, но в отличии от них, в нем можно использовать весь потенциал JavaScript

JSX не обязателен, но давайте сравним следующий код.

```
// Plain JavaScript
const element = React.createElement(
  'a',
```

```
  { href: 'https://google.com' },
  'Google.com',
);

// JSX
const element = <a href="https://google.com">Google.com</div>;
```

Как видите, JSX значительно чище и приятнее для восприятия. Используя JSX, наши компоненты становятся похожи на HTML-шаблоны.

Перепишем предыдущий пример используя JSX.

```
const product = (
  <div className="product">
    <img
      className="image"
      src="https://placeimg.com/320/240/arch"
      alt="yummi"
    />
    <h2 className="name">Raging waffles</h2>
    <p className="text">20$</p>
    <button className="btn" type="button">
      Add to cart
    </button>
  </div>
);

ReactDOM.render(product, document.getElementById('root'));
```

- Так как JSX преобразовывается в вызовы `React.createElement()`, пакет React должен быть в области видимости модуля
- В JSX можно использовать практически любое валидное JavaScript-выражение, оборачивая его в фигурные скобки
- Используя JSX, можно указывать атрибуты и их значения через двойные кавычки, если это обычная строка, или через фигурные скобки, если значение вычисляется
- Все атрибуты React-элементов именуются с помощью `camelCase`
- JSX-теги могут быть родителями других JSX-тегов. Если тег пустой или самозакрывающийся, его `обязательно` необходимо закрыть используя `/>`

Правило общего родителя

Разберем следующий код:

```
const post = (
  <header>Post Header</header>
  <p>Post text</p>
);
```

```
ReactDOM.render(post, document.getElementById('root'));
```

Здесь не валидная JSX-разметка, почему? Перепишем код используя `createElement`.

```
const post = (
  React.createElement('header', null, 'Post Header')
  React.createElement('p', null, 'Post text')
);

ReactDOM.render(post, document.getElementById('root'));
```

Ну вот, не валидное JavaScript-выражение справа от оператора присваивания. Все логично, потому что само по себе присваиваемое выражение не имеет смысла. Выражение это одно значение, результат неких вычислений, отсюда и правило общего родителя.

```
const post = React.createElement(
  'div',
  null,
  React.createElement('header', null, 'Post Header'),
  React.createElement('p', null, 'Post text'),
);

ReactDOM.render(post, document.getElementById('root'));
```

В JSX это выглядит так:

```
const post = (
  <div>
    <header>App Header</header>
    <main>Main Page Content</main>
  </div>
);

ReactDOM.render(post, document.getElementById('root'));
```

Если лишний тег-обертка не нужен в разметке, существуют фрагменты, очень похожие на `DocumentFragment`, которые при рендере растворяются, подставляя содержимое.

```
import React, { Fragment } from 'react';

const post = (
  <Fragment>
    <header>App Header</header>
    <main>Main Page Content</main>
```

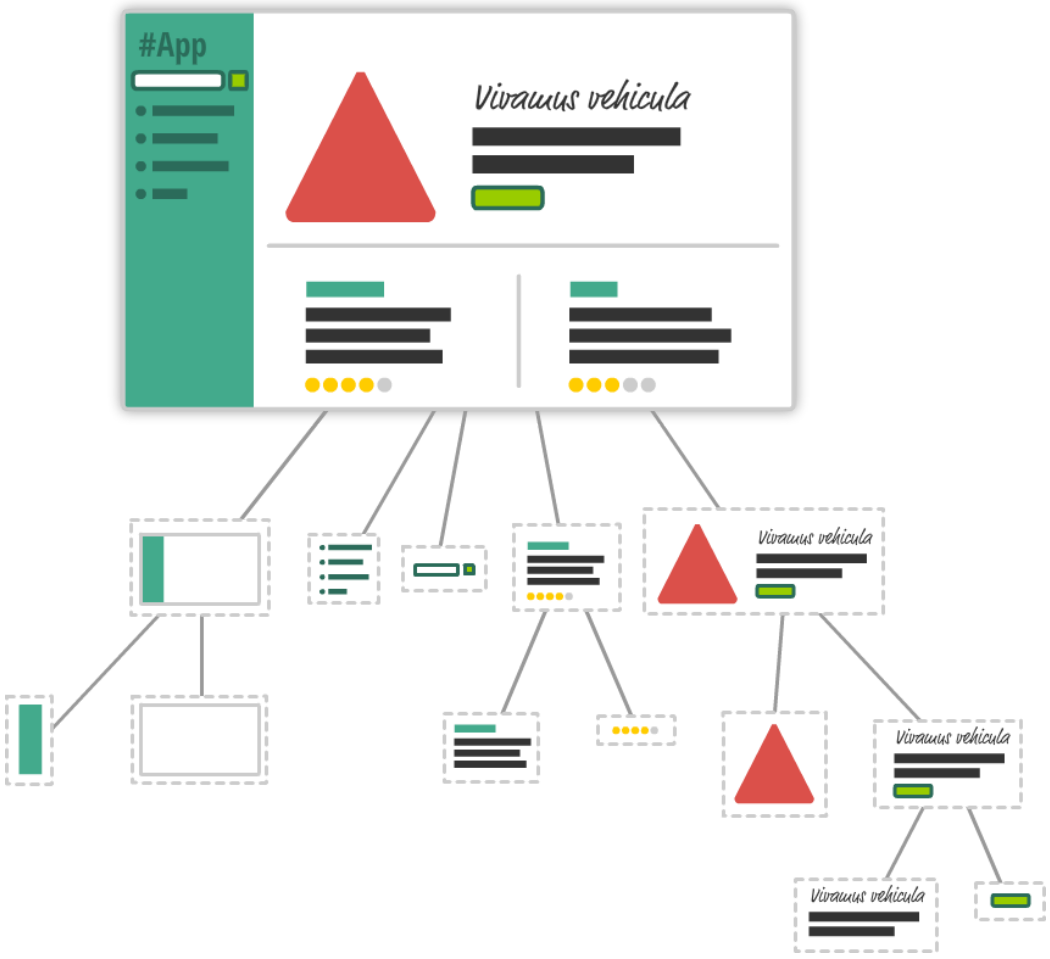
```
  </Fragment>
);

ReactDOM.render(post, document.getElementById('root'));
```

- Introducing JSX
- Differences In Attributes
- Rendering Elements

Компоненты

Компоненты - основные строительные блоки React-приложений, каждый из которых описывает часть интерфейса.

Разработчик создает небольшие компоненты, которые можно объединять, чтобы сформировать более крупные или использовать их как самостоятельные элементы интерфейса. Самое главное в этой концепции то, что и большие, и маленькие компоненты можно использовать повторно и в текущем и в новом проекте.



Таким образом React-приложение можно представить как дерево компонентов, где на верхнем уровне стоит корневой компонент `App`, в котором вложено произвольное количество компонентов.

Каждый компонент должен вернуть разметку (результат JSX, вызов createElement), тем самым указывая React, какой HTML мы хотим отрендерить в DOM.

Функциональные компоненты

В простейшей форме компонент это JavaScript-функция с очень простым контрактом: функция получает объект свойств который называется `props` и возвращает дерево Virtual DOM.

Имя компонента обязательно должно начинаться с заглавной буквы. Названия компонентов с маленькой буквы зарезервированы для HTML-элементов. Если вы попробуете назвать компонент `button`, а не `Button`, при рендере, React проигнорирует его и отрисует обычную HTML-кнопку.

```
const MyComponent = () => <div>Functional Component</div>;
```

Функциональные компоненты составляют большую часть React-приложения. Они чисты и понятны, используйте их как можно чаще.

- Меньше boilerplate-кода
- Легче воспринимать
- Нет внутреннего состояния (stateless)
- Легче тестировать
- Нет контекста (this)

Сделаем карточку продукта функциональным компонентом.

```
const Product = () => (
  <div className="product">
    <img
      className="image"
      src="https://placeimg.com/320/240/arch"
      alt="yummy"
    />
    <h2 className="name">Raging waffles</h2>
    <p className="price">20$</p>
    <button className="btn" type="button">
      Add to cart
    </button>
  </div>
);
```

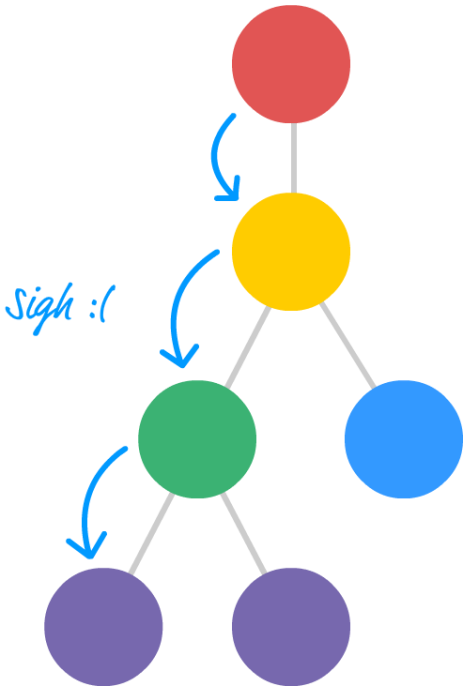
```
// Вызов компонента записывается как JSX-тег
ReactDOM.render(<Product />, document.getElementById('root'));

// Это аналогично
ReactDOM.render(React.createElement(Product),
document.getElementById('root'));
```

React Stateless Functional Components: Nine Wins You Might Have Overlooked

Props

Свойства (props, пропы) это одна из основных концепций React. Компоненты принимают произвольные свойства и возвращают React-элементы, описывающие что должно отрендерится в DOM.

- Пропы используются для передачи данных от родителя к ребенку
- Пропы всегда передаются только вниз по дереву от родительского компонента
- При изменении `props` React ре-рендерит компонент и, возможно, обновляет DOM
- Пропы доступны только для чтения, изменить их в ребенке нельзя
- Пропы могут быть строками или JavaScript-выражениями
- Если передано только имя пропа то это буль, по умолчанию `true`



Свойством может быть текст кнопки, картинка, url, любые данные для компонента.

```
const App = () => (
  <div>
    <h1>Top Products</h1>
    <Product name="Raging waffles" />
  </div>
);
```

Компонент `Product` получает параметр `props`, это всегда будет объект содержащий все переданные свойства. `name` это свойство компонента `Product`, которое было добавлено в объект-свойств.

```
const Product = props => (
  <div>
    <h2>{props.name}</h2>
  </div>
);
```

Добавим компоненту `Products` несколько других свойств.

```
const Product = props => (
  <div className="product">
    <img className="image" src={props.imgUrl} alt={props.alt} />
    <h2 className="name">{props.name}</h2>
    <p className="price">{props.price}$</p>
    <button className="btn" type="button">
      Add to cart
    </button>
  </div>
);
```

Сразу будем использовать простой паттерн при работе с `props`. Так как `props` это объект, мы можем деструктуризировать его в подписи функции.

```
const Product = ({ imgUrl, alt, name, price }) => (
  <div className="product">
    <img className="image" src={imgUrl} alt={alt} />
    <h2 className="name">{name}</h2>
    <p className="price">Price: {price}$</p>
    <button className="btn" type="button">
      Add to cart
    </button>
  </div>
);

const App = () => (
  <div>
    <h1>Top Products</h1>
    <Product
      imgUrl="https://placeimg.com/320/240/arch"
      alt="cool alt"
      name="Raging waffles"
      price={20}
    />
    <Product
      imgUrl="https://placeimg.com/320/240/tech"
      alt="very cool alt"
```

```
      name="Next level tech"
      price={100}
    />
  </div>
);

ReactDOM.render(<App />, document.getElementById('root'));
```

Это уже похоже не реиспользуемый, настраиваемый компонент. Мы передаем ему данные как свойства, а в ответ получаем HTML-разметку с подставленными данными.

- Components and Props
- JSX In Depth
- Spread Attributes

Children

- Свойство `children` автоматически доступно в каждом компоненте, его содержимым является то, что стоит между открывающим и закрывающим JSX-тегом
- В функциональных компонентах обращаемся как `props.children`
- В компонентах объявленых через `class`, обращаемся как `this.props.children`
- Значением `props.children` может быть практически что угодно

В виде детей можно передавать компоненты, как встроенные так и кастомные. Это очень удобно при работе со сложными составными компонентами.

К примеру у нас есть оформительный компонент `Panel`, в который мы можем помещать произвольный контент.

```
const Panel = ({ title, children }) => (
  <section>
    <h2>{title}</h2>
    {children}
  </section>
);

const ProfileDetails = ({ name, age }) => (
  <section>
    <p>{name}</p>
    <p>{age}</p>
  </section>
);

const App = () => (
  <div>
    <Panel title="User profile">
      <ProfileDetails name="Mango" age={2} />
    </Panel>
  </div>
);
```

В противном случае нам бы пришлось пробросить пропы для `ProfileDetails` сквозь `Panel`, что более тесно связывает компоненты и усложняет переиспользование.

Инструменты разработчика

Да, они нужны, часто. Посмотреть состояние, пропы, как и что рендерится и т. д.

- Use React DevTools
- React Developer Tools
- Репозиторий react-devtools

Условный рендеринг

Условный рендеринг в React работает точно так же, как и в JavaScript. Можно использовать `if`, `&&`, `||`, `?` и любые другие операторы условий. Условия можно выполнять перед возвратом разметки, или прямо в JSX.

if с помощью логического оператора &&

```
const Mailbox = ({ unreadMessages }) => (
  <div>
    <h1>Hello!</h1>
    {unreadMessages.length > 0 && (
      <p>You have {unreadMessages.length} unread messages.</p>
    )}
  </div>
);
```

if...else с помощью условного оператора ?

```
const AuthManager = ({ isLoggedIn }) => (
  <div>
    <p>{isLoggedIn ? 'Logout' : 'Login'}<p>
    {isLoggedIn ? <LogoutButton /> : <LoginButton />}
  </div>
);
```

Если по условию ничего не должно быть отрендерено, можно вернуть `null`, `undefined` или `false`, они не рендерятся.

- Conditional Rendering
- All the Conditional Renderings in React

Коллекции

Для того чтобы отрендерить коллекцию однотипных элементов, используется метод `map`, callback-функция которого, для каждого элемента коллекции, возвращает JSX-разметку. Таким образом получаем массив React-элементов который можно рендерить.

```
const tech = [
  { id: 'id-1', name: 'JS' },
  { id: 'id-2', name: 'React' },
  { id: 'id-3', name: 'React Router' },
  { id: 'id-4', name: 'Redux' },
];

const TechList = ({ items }) => (
  <ul>
    {items.map(item => (
      <li>{item.name}</li>
    ))}
  </ul>
);

ReactDOM.render(<TechList items={tech} />,
document.getElementById('root'));
```

Ключи

При выполнении кода из примера выше, всплывет предупреждение о том, что для элементов списка требуется ключ.

key — это специальный строковый атрибут, который нужно задать при создании элементов списка.

- React не может отличить элементы в коллекции, таким образом, перерисовывая всю коллекцию целиком при любых изменениях.
- Элементы внутри массива должны быть обеспечены ключами, чтобы иметь стабильную идентичность.
- Важно чтобы среди одной коллекции элементов ключи не повторялись.

Лучший способ задать ключ — использовать статическую строку, которая однозначно идентифицирует элемент списка среди остальных. В качестве ключей используются идентификаторы объектов.

```
const tech = [
  { id: 'id-1', name: 'JS' },
  { id: 'id-2', name: 'React' },
  { id: 'id-3', name: 'React Router' },
  { id: 'id-4', name: 'Redux' },
];

const TechList = ({ items }) => (
  <ul>
    {items.map(item => (
      <li key={item.id}>{item.name}</li>
```

```
    ))}
  </ul>
);

ReactDOM.render(<TechList items={tech} />,
document.getElementById('root'));
```

- Не используйте индексы массива для ключей, в случае если элементы изменят порядок, изменятся ключи, и React придется заново запомнить элементы
- Никогда не создавайте ключи в рантайме, ключ должен быть постоянным, неизменным значением

Lists and Keys

Дополнительные материалы

- React blog
- Полное руководство по ReactJS (перевод документации)
- Lin Clark - A Cartoon Intro to Fiber - React Conf 2017
- 9 things every React.js beginner should know

Licence

Owner - GoIT

Author - Alexander Repeta