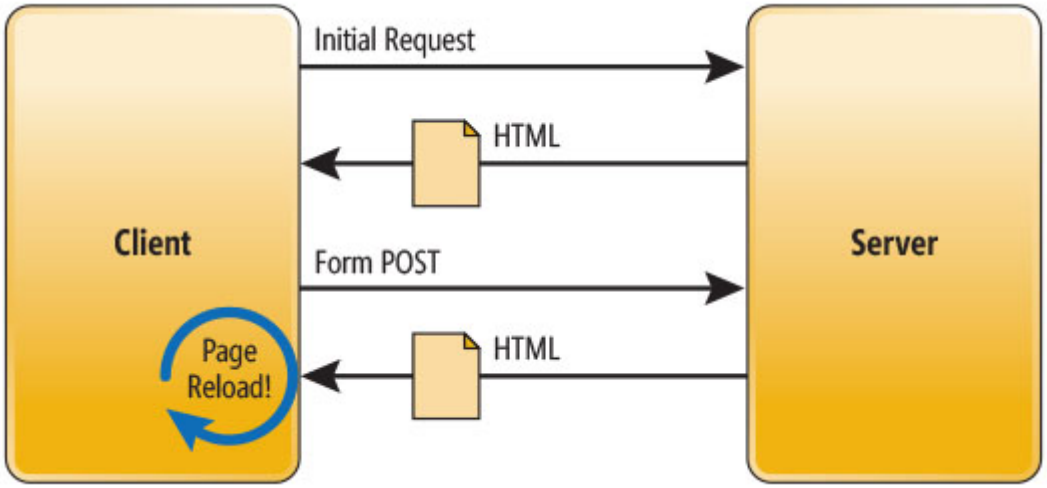
Основы. Компоненты. Props. JSX

Multiple-page application (MPA)

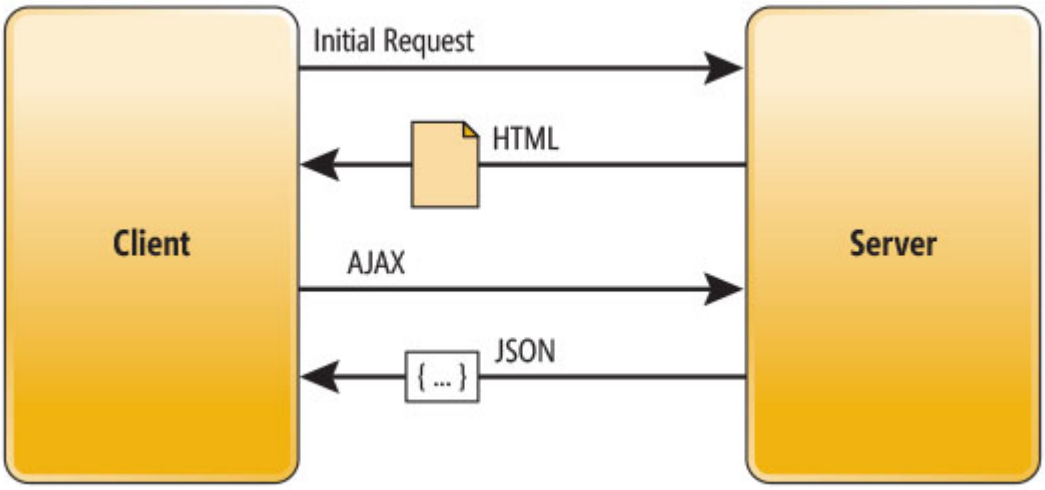
Классический веб-сайт, интернет магазины, сайты визитки и т.п.



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

Single-page Application (SPA)

Одностраничное приложение это веб-сайт интерфейс которого постоянно перерисовывается прямо на клиенте, вместо запроса страниц с сервера, без перезагрузки страницы. Такой подход позволяет создать веб-сайт который по ощущениям очень похож на десктоп-приложение.



- Архитектура клиент <-> сервер

- При загрузке страницы сервер отдает бандл с js-кодом
- Каждый последующий запрос на сервер получает только JSON-данные
- Обновление интерфейса происходит динамически прямо на клиенте
- Нету перезагрузки страницы при каждом запросе на сервер
- Отзывчивый интерфейс
- Загрузка первой страницы может быть довольно медленной (лечится)
- Сложность кода и его поддержки масштабируется с кол-вом функционала
- Слабое SEO (лечится рендером на сервере)

Материалы

- Single-page application vs. multiple-page application
- Everything You Need to Know About Single Page Applications

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.
- **Декларативный** - позволяет декларативно описывать интерфейс как функцию от данных. React, при каждом изменении данных, эффективно обновит интерфейс.
- **Компонентный подход** - основная концепция это компоненты, маленькие строительные блоки из которых собирается 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

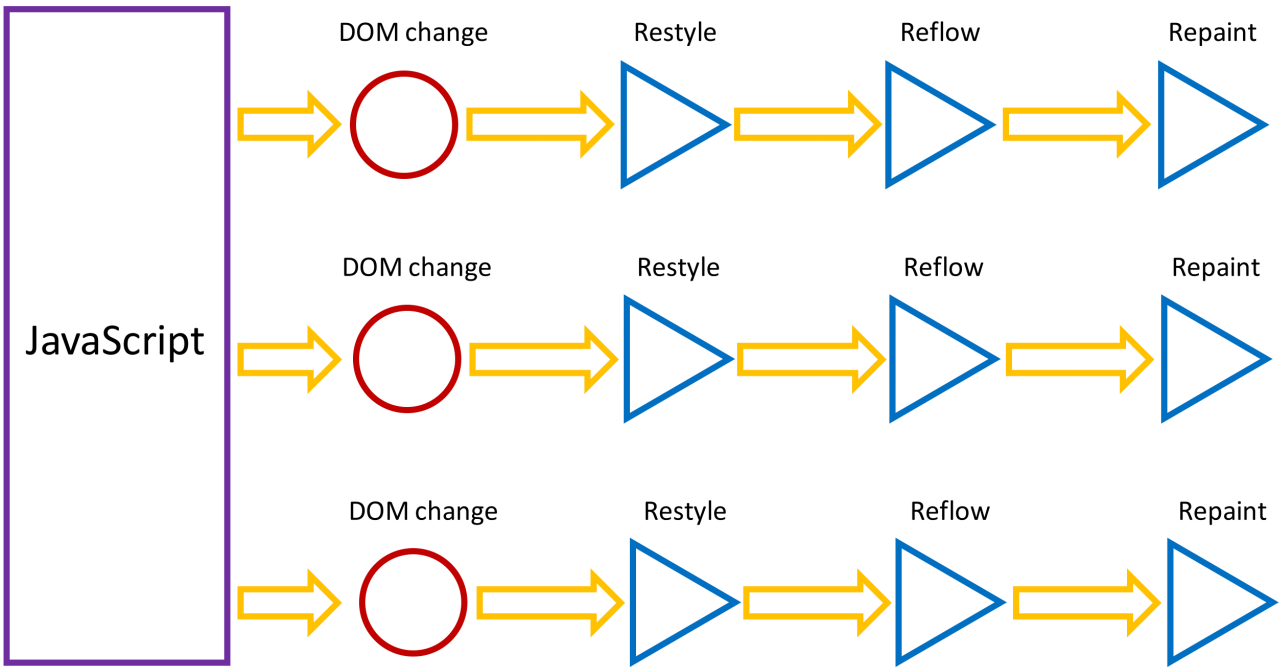
DOM - древовидное представление HTML-документа. Хранится в памяти браузера и напрямую связан с тем что мы видим на странице.

- 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

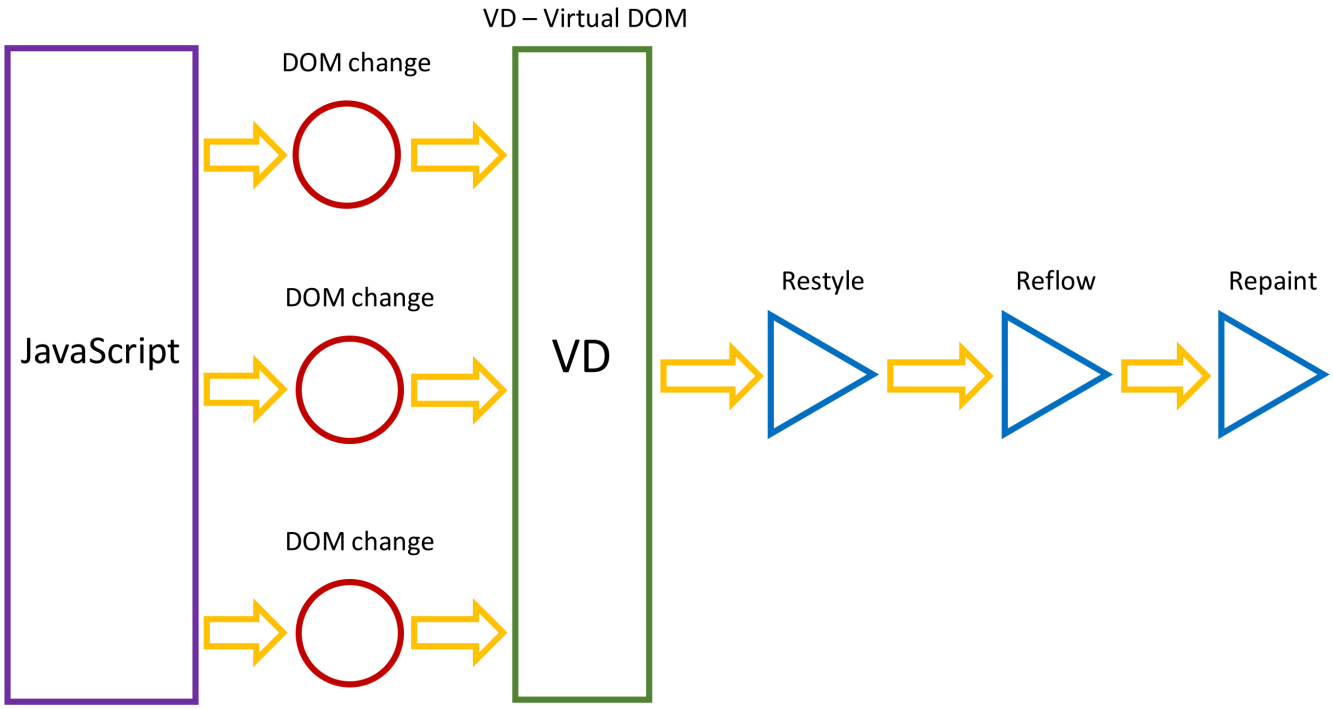
Rendering: repaint, reflow/relayout, restyle



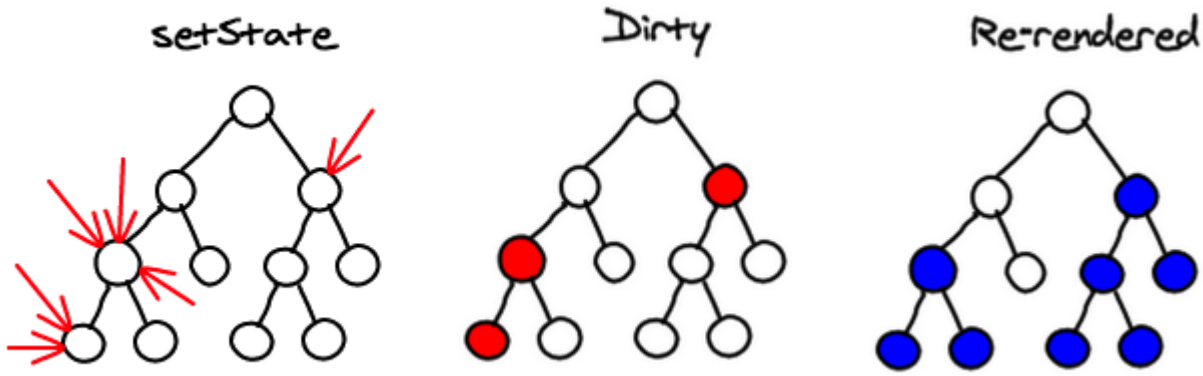
Virtual DOM

React хранит копию DOM в виде Virtual DOM.

- Легковесная копия реального DOM представленная JavaScript-объектами.
- Не зависит от внутренней имплементации браузера.
- Использует лучшие практики при обновлении реального DOM.
- Позволяет собирать обновления в группы для оптимизации рендера. Таким образом repaint выполняется один раз для группы.



- Некоторые действия пользователя изменяют 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.
- 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.



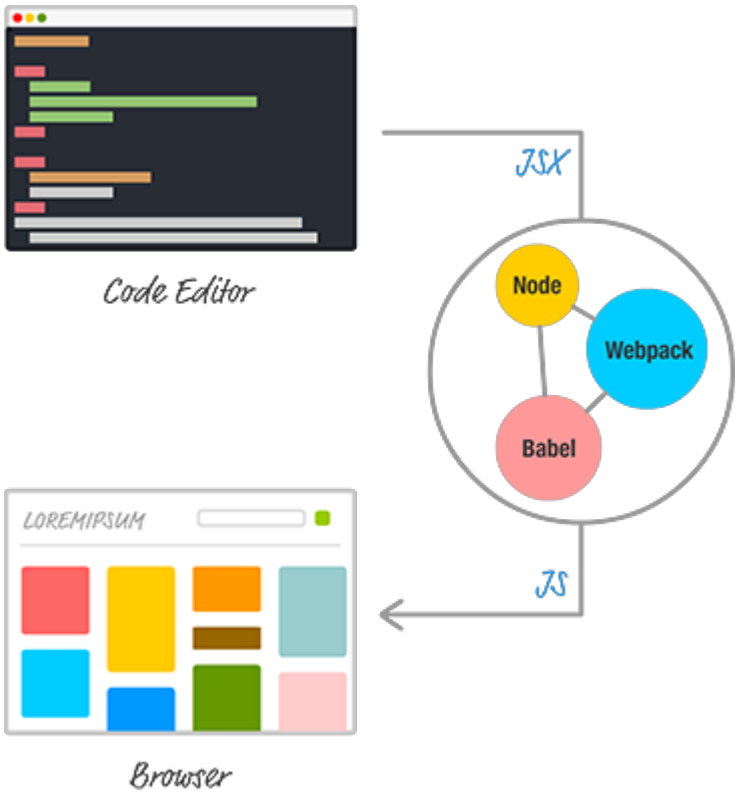
Материалы

- What is Virtual Dom
- Reconciliation
- Using Virtual DOM in React.js: Top 5 Benefits
- The difference between Virtual DOM and DOM

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

Инструменты

Для начала необходим NodeJS, Webpack, Babel и сам React.



- Абстрагирует всю конфигурацию, позволяя сосредоточиться на написании кода.
- Включает самые необходимые и частоиспользующиеся пакеты
- Имеет функцию извлечения, которая удаляет абстракцию и предоставляет базовую конфигурацию.
- Легко настраивается "под себя"
- Легко расширяется дополнительными инструментами

```
npm i -g create-react-app
```

или

```
yarn global add create-react-app
```

После чего команда `create-react-app` будет доступна в консоли.

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

- Create a New React App

- create-react-app docs

React-элементы

- Элементы это самые маленькие строительные блоки React.
- Элемент описывает то, что вы хотите увидеть на странице.
- В отличии от DOM-элементов, элементы в React это обычные объекты, поэтому создавать их очень быстро.
- `ReactDOM` отвечает за обновление реального DOM в соответсвии с указанными React-элементами.
- Не путайте элементы с компонентами. Элементы это то, из чего компоненты состоят.

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 и React-компонентов.

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

- `type` - это не имя HTML-тега, это имя встроенного React-компонента который соответсвует HTML-тегу ссылки 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: 'product-image',
    src: 'https://placeimg.com/320/240/arch',
    alt: 'yummi',
  }),
  React.createElement('h2', { className: 'product-name' }, 'Raging waffles'),
  React.createElement('p', { className: 'product-price' }, 20),
  React.createElement('button', { className: 'product-btn' }, '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 (что рендерить), а вторым, ссылку на уже существующий DOM-элемент (куда рендерить).

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

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

Материалы

React Components, Elements, and Instances

JavaScript Syntax Extension

Код в предыдущем разделе - это то, что понимает браузер. Однако, для человека, описывать интерфейс таким образом неудобно, нам привычен HTML.

JSX - позволяет писать XML-образный синтаксис, который впоследствии трансформируется в вызовы `React.createElement()`.

Возможно, поначалу вам будет несколько необычно мешать шаблоны и логику. Но как только вы ближе познакомитесь с React, любые сложности восприятия кода отойдут на второй план.

- JSX описывает объекты, элементы Virtual DOM
- JSX позволяет использовать XML-образный синтаксис прямо в JavaScript
- Упрощает код, делает код декларативным и читабельным
- Это не HTML, Babel трансформирует JSX в вызовы `React.createElement()`
- JSX 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="product-image"
      src="https://placeimg.com/320/240/arch"
      alt="yummi"
    />
    <h2 className="product-name">Raging waffles</h2>
    <p classproduct-name="product-text">20</p>
    <button classproduct-name="product-btn">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.
- После преобразования, JSX это обычные функции возвращающие объект.
- 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, 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, хм, почему? Давайте перепишем код используя `createElement` и посмотрим.

```
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'));
```

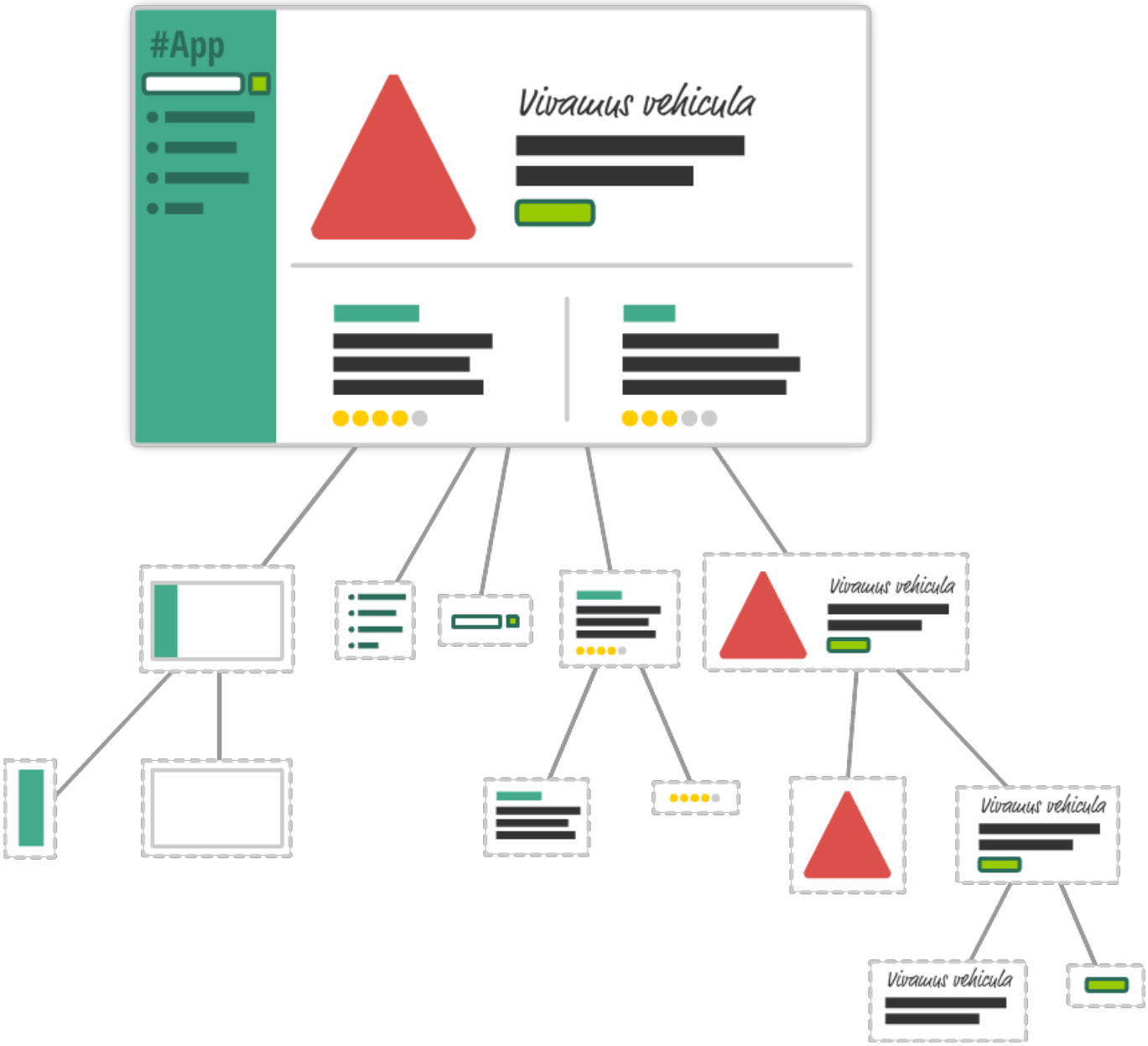
Материалы

- Introducing JSX
- Differences In Attributes
- Rendering Elements
- Pete Hunt: React - rethinking best practices

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 вложенных 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 обычно называется `props` и возвращает элемент Virtual DOM.

У этих компонентов нету состояния - `stateless`. Они чисты и понятны, используйте их как можно чаще.

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

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

Функциональные компоненты составляют большую часть React-приложения

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

Сделаем карточку продукта функциональным компонентом.

```
const Product = () => (
  <div className="product">
    <img
      className="product-image"
      src="https://placeimg.com/320/240/arch"
      alt="yummy"
    />
    <h2 className="product-name">Raging waffles</h2>
```

```
    <p className="product-price">20$</p>
    <button className="product-btn">Add to cart</button>
  </div>
);

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

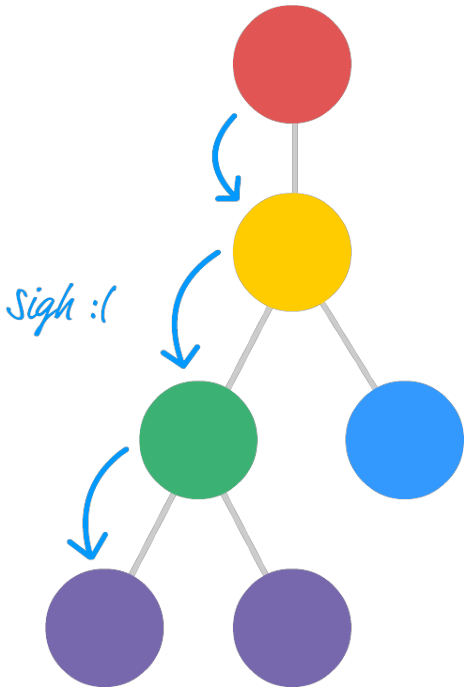
Отлично, у нас есть компонент. К сожалению он не гибкий и его нельзя отрендерить с другой информацией. Давайте решать эту задачу :)

Материалы

React Stateless Functional Components: Nine Wins You Might Have Overlooked

Props

Свойства (`props`) это одна из основных концепций React. Компоненты принимают произвольные свойства (настройки, называемые `props`) и возвращают React-элементы, описывающие что должно появиться на экране.



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

```
// App.jsx
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.

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

Добавим компоненту `Products` несколько других свойств.

```
const Product = props => (
  <div className="product">
    <img className="product-image" src={props.imgUrl} alt={props.alt} />
    <h2 className="product-name">{props.name}</h2>
    <p className="product-price">{props.price}$</p>
    <button className="product-btn">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="product-image" src={imgUrl} alt={alt} />
    <h2 className="product-name">{name}</h2>
    <p className="product-price">Price: {price}$</p>
    <button className="product-btn">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={999}
    />
  </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.

- 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`

Материалы

- 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 компонента
- 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.

Текст

Обычный текст между открывающим и закрывающим JSX-тегом передается как `children`.

```
const Button = ({ children }) => <button>{children}</button>;

// Строка 'Login' будет доступна в props.children компонента Button
<Button>Login</Button>;
```

JSX-выражения

Любое валидное JSX-выражение может быть передано как `children`. Это чаще всего используется для рендера динамических данных от `props`.

```
const items = ['wake up', 'drink coffee', 'code till sunset'];

const List = () => <ul>{items.map(item => <li>{text}</li>)}</ul>;
```

Компоненты

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 }) => (
  <div>
    <h2>{title}</h2>
    {children}
  </div>
);

<Panel title="User profile">
  <ProfileDetails details={details} />
</Panel>;
```

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

PropTypes

Поскольку JavaScript является динамически типизированным языком, нет способа жестко указать ожидаемый тип переменной. Если компоненту придут пропы не того типа, будут ошибки в рантайме.

PropTypes - один из вариантов проверки типа свойств во время разработки.

```
import PropTypes from 'prop-types';

const Button = ({ type, text }) => <button type={type}>{text}</button>;

Button.propTypes = {
  type: PropTypes.string,
  text: PropTypes.string.isRequired,
};

Button.defaultProps = {
  type: 'button',
};
```

```
ReactDOM.render(
  <div>
    <Button text={5} />
    <Button type={[1, 2, 3]} text="Click me" />
  </div>,
  document.getElementById('root'),
);
```

- Свойтсво `propTypes` содержит объект с полями по имени свойств и значениями указывающими ожидаемый тип.
- Свойство `defaultProps` содержит объект тех свойств, которые не указаны как обязательные.
- Проверка происходит только в разработке.

Материалы

- Typechecking With PropTypes
- prop-types

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 возвращаение 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`, он не рендерится, также как `undefined` и `false`.

- Conditional Rendering

- All the Conditional Renderings in React

Коллекции

Для того чтобы отрендерить коллекцию однотипных элементов (посты, карточки продуктов и т.п), используется метод `map`, callback-функция которого, для каждого элемента коллекции, возвращает JSX-разметку. Таким образом мы 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.

React не может отличить элементы в коллекции, таким образом, перерисовывая всю коллекцию целиком при любых изменениях. Добавляя уникальные идентификаторы для каждого элемента, мы помогаем React оптимизировать работу с коллекцией.

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

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. Чаще всего, в качестве ключей вы будете использовать идентификаторы объектов данных.

```
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, что относительно медленно.
- Никогда не создавайте ключи в рантайме, ключ должен быть постоянным, неизменным значением.

Материалы

Lists and Keys

Дополнительные материалы

- 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