**Инфраструктура приложения: CRA**

Приступим к созданию нового приложения на React. Инфраструктуру приложения можно настраивать самостоятельно или пользоваться готовым решением. Оба способа имеют и плюсы, и минусы. Опытные разработчики иногда делают собственные инфраструктурные шаблоны, которые сочетают в себе всё необходимое. Но пока наша задача — освоить React, а не мериться с коллегами инфраструктурными файлами. Так что выберем готовую инфраструктуру.

В этом уроке рассмотрим утилиту Create React App (CRA), которая предоставляет простое окружение для начала работы с React-приложением. Чтобы установить пакет create-react-app утилиты CRA, потребуется Node.js и NPM.

**Подготовительный этап**

Установите Node.js на свой компьютер. Для этого скачайте дистрибутив с [официального сайта](https://nodejs.org/en/) в соответствии с вашей операционной системой. Мы рекомендуем использовать LTS сборку: она более стабильна и содержит всё необходимое.

После установки откройте терминал на компьютере. Это может быть **Terminal** на macOS или **cmd.exe** на Windows. Введите в терминал команду:

Скопировать код

node -v

Если всё прошло успешно, то вы увидите версию установленной Node.js, например так:

Скопировать код

v14.12.0

Теперь введите эту команду и убедитесь, что пакетный менеджер npm установлен корректно. Он устанавливается вместе с Node.js.

Скопировать код

npm -v

В консоли должна появиться версия пакетного менеджера, например:

Скопировать код

6.14.8

Версии, которые вы установили, могут отличаться от описанных в этом материале — это не критично.

После этой операции вам стали доступны более миллиона библиотек и пакетов из <https://www.npmjs.com/>. Чтобы ознакомиться с возможностями npm, перейдите [по ссылке на официальную страницу документации](https://docs.npmjs.com/about-npm).

**Зачем использовать Create React App**

Автомобильные производители обычно используют комплектующие, которые собирают заранее. Это экономит им много времени и денег. Так и в программировании: утилита Create React App помогает разработчикам писать код быстрее и эффективнее.

Как и React, она разработана командой Facebook. CRA позволяет быстро собрать и запустить проект: можно создать его на основе стандартного шаблона и не беспокоиться о детальной настройке.

Основные причины применять CRA:

* Можно развернуть полноценный рабочий проект с помощью одной команды.
* Проект уже содержит удобную файловую структуру, стандартную для большинства приложений на React.
* В проект легко подключить статические ресурсы: изображения, шрифты и так далее.
* Есть готовый механизм для тестирования компонентов. С ним вы познакомитесь позже.
* Проект будет автоматически обновляться в браузере, если вносить изменения в режиме разработки.
* Настроена оптимизация продакшн-сборки.
* Можно использовать NPM, легко устанавливать пакеты.

В теории настроить всё это можно и вручную, но CRA существенно ускоряет этот процесс и, что немаловажно, учитывает при этом общепринятые подходы. У вас будет возможность самостоятельно настроить всю инфраструктуру, но об этом мы расскажем в дополнительной программе.

**Установка Create React App**

Это несложно и похоже на установку пакетов NPM.

Установите CRA глобально с помощью npm install -g create-react-app или yarn global install create-react-app.

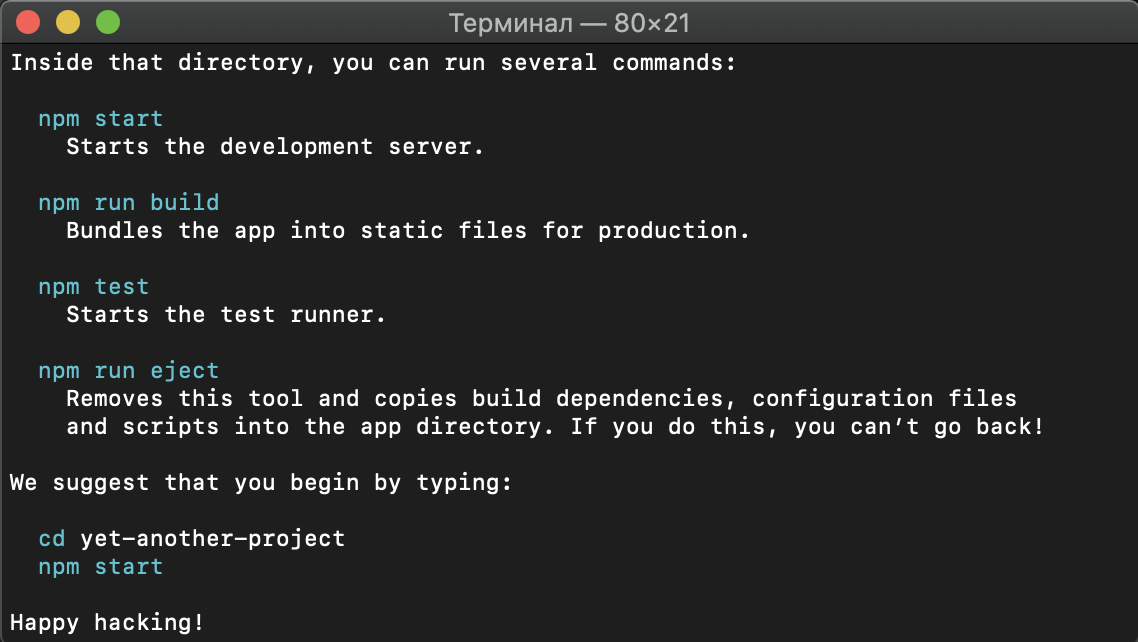
Вот и всё! Можно создавать первое React-приложение.

**Создание приложения**

В командной строке перейдите в директорию, где планируете создать проект. Для нового CRA-проекта используют команду npx create-react-app название\_проекта. Название проектов обычно пишут латиницей. Наш проект назовём yet-another-project, поэтому введём команду npx create-react-app yet-another-project.

После этого начинается магия CRA. Этот процесс может занять несколько минут — отличный повод сделать перерыв.

После успешного выполнения скрипта появится такое сообщение:



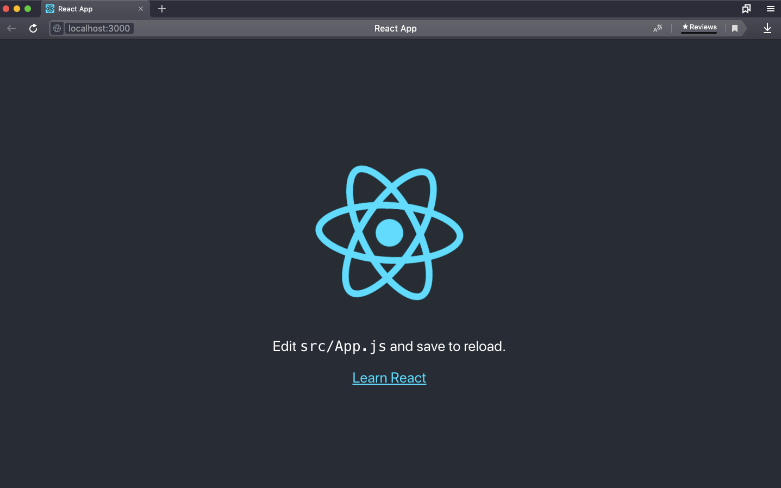
Теперь в вашем распоряжении проект, полностью готовый к работе. Запустим его.

**Запуск проекта**

Перейдите в директорию проекта командой cd yet-another-project. Находясь в ней, введите команду npm start.

После этого проект автоматически откроется в браузере. Если этого не произошло, введите адрес <http://localhost:3000/>.

Сейчас проект выглядит так:



*Уже неплохо, но будет ещё лучше*

Это шаблон проекта, который Create React App создаёт по умолчанию: простой компонент из вращающегося логотипа и текста. Настоящему приложению потребуется совсем другое наполнение. Но прежде чем редактировать App.js, нужно ознакомиться со структурой созданного проекта.

**Файлы инфраструктуры**

Посмотрим на созданные файлы в новом приложении:

Скопировать код

└── yet-another-project/

├── src/

├── public/

├── .gitignore

├── package.json

├── package-lock.json

└── README.md

В директории проекта нет знакомых вам файлов babel.config.js и webpack.config.js, в которых указываются настройки для сборки приложения и транспиляции кода. Но это не магия вне Хогвартса. CRA использует Babel и Webpack «под капотом» и всю настройку этих пакетов выполняет за вас.

Тем не менее CRA предлагает возможность редактировать настройки ESLint и Babel. Это можно сделать внутри файла package.json. Например, настройка расширений для ESLint выглядит так:

Скопировать кодJSON

"eslintConfig": {

"extends": [

"react-app",

"react-app/jest"

]

},

Пока рекомендуем не редактировать эти настройки. На более поздних спринтах конфигурация настроек ESLint не будет подходить под наши задачи, и мы её расширим.

В следующем уроке подробнее рассмотрим структуру проекта и компоненты в нём.

**Структура приложения**

В предыдущем уроке вы познакомились с CRA и файловой структурой приложения. Но в нашем приложении есть две директории, про которые мы ещё не говорили, — src/ и public/:

Скопировать код

└── yet-another-project/

├── src/

│ ├── App.css

│ ├── App.js

│ ├── App.test.js

│ ├── index.css

│ ├── index.js

│ ├── logo.svg

│ ├── reportWebVitals.js

│ └── setupTests.js

├── public/

│ ├── index.html

│ └── ...Другие файлы со странными названиями

├── .gitignore

├── package.json

├── package-lock.json

└── README.md

Назначение директорий:

* Директория public/ содержит скомпилированное приложение. Именно эти файлы попадают к пользователю на сайте. В директории обязательно должен быть файл с именем index.html.
* Директория src/ содержит исходный код приложения. В ней обязательно должен быть файл с именем index.js.

Все файлы, которые относятся к функциональности и работе приложения, должны находиться внутри src/. Файлы вне этой директории CRA игнорирует при сборке.

Подробнее рассмотрим директорию src/: все файлы в ней лежат без отдельных директорий. Сейчас приложение очень маленькое и состоит из одного логотипа и нескольких текстовых элементов. Маленькие приложения часто не требуют особого порядка и жёсткого контроля файловой структуры. Но ваше приложение будет масштабироваться и расти. Тут на помощь и приходят лучшие практики структурирования React приложений.

Конечно, мы могли бы написать всё приложение в одном файле, и оно работало бы ничуть не хуже, но изменения и поиск неисправностей стали бы настоящим испытанием для вас и колёсика вашей мыши. Ещё больший спектр эмоций испытали бы ваши коллеги, когда столкнулись бы с таким кодом. Именно поэтому важно с самого начала озадачиться удобным структурированием будущего приложения.

Мы расскажем про два основных подхода в структурировании:

* по принадлежности к определённому структурному типу;
* по выполняемым функциональным особенностям.

Например, так выглядит структурирование по принадлежности к структурному типу. Иначе говоря, структурирование по типу файла:

Скопировать код

└── src/

├── components/

│ ├── homepage.js

│ ├── cart.js

│ └── search.js

├── tests/

│ ├── homepage.test.js

│ ├── cart.test.js

│ └── search.test.js

└── styles/

├── homepage.css

├── cart.css

└── search.css

В такой структуре компоненты, тесты и стили лежат отдельно. Эта структура удобна для небольших приложений и зачастую встречается в монолитных приложениях.

Следующий пример, наоборот, хорошо подходит для крупных приложений. В нём файлы сгруппированы по продуктовым функциям:

Скопировать код

└── src/

└── components/

├── homepage/

│ ├── homepage.js

│ ├── homepage.test.js

│ └── homepage.css

├── cart/

│ ├── cart.js

│ ├── cart.test.js

│ └── cart.css

└── search/

├── search.js

├── search.test.js

└── search.css

В такой структуре директория с компонентом содержит все файлы, которые относятся к его работе: тесты, стили и сам компонент. Такие блоки удобно отделять друг от друга, поддерживать распределёнными командами и превращать в подключаемые модули.

Способы структурирования выше — не единственные. Можно разделить компоненты Homepage и Cart на разные папки, а можно выделить из обоих функциональностей похожие. Например, элементы интерфейса расположить в одной папке, а функции, которые отвечают за взаимодействие с сервером, — в другой.

Любой вариант структурирования жизнеспособен и активно применяется многими компаниями. В проектной работе мы рекомендуем использовать группировку по функциям в приложении. А на курсе мы ещё неоднократно будем обращаться к теме структуры приложения.

В этом уроке вы узнали, как улучшить опыт от разработки — DX. Вокруг этого построено множество вспомогательных библиотек и инструментов, но немаловажна и структура папок и файлов в приложении. В следующем уроке перейдём к деталям работы с приложением.

# Импорт модулей

В предыдущем уроке вы узнали, почему так важно структурировать проекты. Теперь расскажем о способах импортирования модулей в проект.

Использование CRA даёт нам ещё одно преимущество. До использования CRA весь новый код на React был написан внутри одного тега script:

Скопировать кодJSX

<script type="text/jsx">

class DogImage extends React.Component {

render() {

return (

<img alt='Грустная собачка' />

);

}

}

ReactDOM.render((

<>

<DogImage />

</>

), document.querySelector('#root'));

</script>

Если продолжать писать код подобным образом, файлы станут большими, а их содержимое — трудным для восприятия. Команде тяжелее поддерживать большой файл в актуальном состоянии. А ещё в него намного сложнее внедрять новую функциональность. Это лишнее время, а время для бизнеса — деньги. Но есть современное решение этой проблемы, с которым вы уже знакомы. Можно разделить код на модули и использовать конструкции export и import.

Модули — синтаксис ES6, они требуют дополнительной подготовки для правильной работы кода. Create React App уже включает в себя Webpack и Babel со всеми необходимыми настройками, что экономит время на их конфигурации.

Благодаря CRA мы можем переписать код в App.js с использованием import и знаний о компонентах в React:

Скопировать кодJSX

*// App.js*

import React from 'react' *// импорт библиотеки*

import ReactDOM from 'react-dom'

import DogImage from './dog-image/dog-image'

class App extends React.Component {

render() {

return (

<div className='App'>

<DogImage/>

</div>

)

}

}

export default App;

Импортируем пакеты react, react-dom и компонент DogImage с указанием перед ними ключевого слова import. После пропишем путь до этих файлов, используя относительный путь до файла.

Компонент DogImage вынесем в отдельный файл и экспортируем с помощью export default:

Скопировать кодJSX

*// ./dog-image/dog-image.js*

class DogImage extends React.Component {

render() {

return (

<img alt='Грустная собачка' />

);

}

}

export default DogImage

Таким же образом можно разделять, организовывать, импортировать и экспортировать компоненты внутри проекта.

Такое разделение может показаться лишним, ведь компонент один и содержит в себе только изображение. Но в предыдущем уроке мы выяснили, что приложения становятся больше, а количество компонентов в них растёт. И вот уже добавляется новый компонент:

Скопировать кодJSX

*// App.js*

import React from 'react' *// импорт библиотеки*

import ReactDOM from 'react-dom'

import DogImage from './dog-image/dog-image'

import Header from './header/header'

class App extends React.Component {

render() {

return (

<div className='App'>

<Header/>

<DogImage/>

</div>

)

}

}

export default App;

*// ./header/header.js*

class Header extends React.Component {

render() {

return (

<h1>

Мемы и искусство

</h1>

);

}

}

export default Header

Конструкции import и export помогают соединить всю структуру из компонентов. Пока проект небольшой, это незаметно. Но лучше сразу позаботиться о файловой структуре и взаимодействии между компонентами. В дальнейшем, когда вы уже будете понимать размер написанного кода, не придётся заниматься монотонным рефакторингом.

В следующих уроках расскажем про работу с CSS-кодом и изображениями.

# Использование изображений

В предыдущем уроке мы добавили компонент DogImage и экспортировали его:

Скопировать кодJSX

*// ./dog-image/dog-image.js*

import React from 'react';

class DogImage extends React.Component {

render() {

return (

<img alt='Грустная собачка' />

);

}

}

export default DogImage

Кажется, что без атрибута src браузер не отрисует картинку. Если попробовать установить значение атрибуту src в виде относительного пути до изображения — оно также не отобразится. Дело в том, что Webpack хеширует файлы при сборке проекта. Имя файла при хешировании изменяется, поэтому статически указанный путь в src не сможет найти изображение, имя файла которого изменилось.

Это не очередная шутка от разработчиков, а оптимизация процесса сборки проекта и работы с кешем браузера. Если файл изменился — Webpack изменит его хеш, а браузер обновит этот файл в кеше. Конечные пользователи приложения будут получать его актуальную версию, а проблема устаревших файлов в кеше браузера их не затронет.

Научиться работать с файлами изображений по правилам Webpack поможет import. Создадим директорию images/ в проекте и разместим в ней изображение dog-image.jpg:

Скопировать код

└── src/

├── components/

│ ├── dog-image/

│ │ └── dog-image.js

│ └── header/

│ └── header.js

│ <!-- остальные компоненты, файлы для тестирования -->

├── images/

│ └── dog-image.jpg

└── index.js

Теперь напишем import к файлу изображения: для этого присвоим переменной dogImagePath импортированное значение. Воспользуемся этой переменной в атрибуте src:

Скопировать кодJSX

*// ./dog-image/dog-image.js*

import React from 'react';

import dogImagePath from '../../images/dog-image.jpg'

class DogImage extends React.Component {

render() {

return (

<img src={dogImagePath} alt='Грустная собачка' />

);

}

}

export default DogImage

Суперспособность Webpack, не иначе. Такой директивой import мы говорим Webpack включить изображение в итоговую сборку, а в переменную dogImagePath сохраняется итоговый путь до изображения.

После сборки изображение корректно отобразится в проекте:



Доге в виде арта? Почему бы и нет

А вот и Доге! Теперь вы умеете добавлять картинки в компоненты. В следующем уроке по-новому взглянем на CSS-стили в React-проектах.

**CSS-Модули. Composes**

Проект с мемами из предыдущего урока всё ещё выглядит невзрачно, поэтому нужно подумать, как его стилизовать. В этом уроке расскажем, как решить главную проблему CSS-кода: глобальное пространство имён. А ещё научимся подключать стили в приложении на React.

Сначала разберёмся, зачем вообще нужно решать проблему глобальной видимости. Если вы хоть раз писали на CSS, то эти сложности вам наверняка известны. Возможно, вы уже давно  свыклись с ними и перестали замечать. Например, методология БЭМ старается решить проблемы коллизий имён классов и создать что-то похожее на область видимости — вы создаёте блоки, элементы и модификаторы. Всё это — сложное пространство имён, которое защищает большие кодовые базы CSS от конфликтов. Иначе в какой-то момент в проекте появится 10 одинаковых классов button, стили которых будут переопределять друг друга.

Разберёмся на примере. Возьмём простое поле ввода и опишем два его состояния: обычное и неактивное. Сделаем это в методологии БЭМ:

Скопировать кодCSS

.input {

background: #f4f4f4;

border-radius: 6px;

border-color: transparent;

padding: 15px 5px;

}

.input\_state\_disabled {

color: #bbb;

background: #f9f9f9;

}

Мы использовали модификатор \_state блока input со значением disabled. И проделали эту работу лишь затем, чтобы не стало очень больно, если написать CSS несколько иначе. CSS заставляет писать код в сложном пространстве имён, чтобы скрывать проблему глобальной области видимости. Попробуем переписать код так, будто не боимся CSS:

Скопировать кодCSS

.input {

background: #f4f4f4;

border-radius: 6px;

border-color: transparent;

padding: 15px 5px;

}

.input.disabled {

*/\* Вот если бы мы совсем не боялись CSS, то написали бы просто .disabled \*/*

color: #bbb;

background: #f9f9f9;

}

Похоже бояться CSS всё-таки стоит, а проблема глобального пространства имён существует. Приложение будет разрастаться, и разработчик создаст селектор .disabled где-нибудь ещё в CSS. Тогда поле ввода получит, например, красивую красную рамку и ещё десяток CSS-свойств. А если разработчик решит переопределить CSS-правила неактивному полю ввода другими селекторами — придётся решить проблему специфичности селектора .input.disabled.

Чтобы поставить CSS «на место», разработчики собрали все лучшие практики из CSS и JS для создания npm-пакета css-loader, который образует локальные области видимости — CSS-модули.

CSS-модули хороши по нескольким причинам:

* Не требуют дополнительной настройки и содержатся в CRA.
* Позволяют не задумываться о пространстве имён — каждый компонент имеет отдельный CSS-файл и свою область видимости.
* Повышают возможность повторно использовать компоненты без возникновения коллизий, если у разных компонентов появятся одинаковые селекторы классов.

Теперь вернёмся к проекту с мемами и установим новые правила для CSS. Сейчас проект выглядит так:

Скопировать кодMARKDOWN

└── src/

├── components/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ ├── dog-image/

│ │ └── dog-image.js

│ └── header/

│ └── header.js

├── images/

│ └── dog-image.jpg

└── index.js

Сначала создадим для компонента App свою комнату в проекте. Для этого выделим ему отдельную директорию:

Скопировать кодMARKDOWN

└── src/

├── components/

│ ├── app/

│ │ ├── app.js

│ │ ├── app.css

│ │ └── app.test.js

│ ├── dog-image/

│ │ └── dog-image.js

│ └── header/

│ └── header.js

├── images/

│ └── dog-image.jpg

└── index.js

В проекте содержатся компоненты без CSS-файлов. Займёмся ими чуть позже, а пока рассмотрим app.css. По умолчанию этот файл содержит много стилей, которые CRA применяет для стилизации шаблонного проекта. Удалим их и оставим только один селектор:

Скопировать кодCSS

.app {

text-align: center;

}

Теперь в игру вступают CSS-модули. Для начала переименуем файл app.css в app.module.css — так Webpack поймёт, как правильно работать с этим файлом. Затем импортируем его в App.js и присвоим значение переменной styles:

Скопировать кодJSX

import React from 'react';

import styles from './app.module.css';

*// Импортируем компоненты*

import Header from '../header/header';

import DogImage from '../dog-image/dog-image';

*// Далее идёт содержимое компонента App, которое нам пока не очень интересно*

На этом моменте можно сделать паузу и приготовить вкусный напиток. Дальше разберёмся, что именно произошло.

Импортировать CSS-файл можно двумя способами: в зависимости от того, указано ли ключевое слово module перед расширением файла.

Если CSS-файл — не модуль, подключить его можно так:

Скопировать кодJSX

import './awesome-css-file.css';

Эта директива сообщает Webpack, что содержимое CSS-файла нужно взять в сборку. Если посмотреть на исходный код страницы во время работы приложения, то содержимое этого файла можно увидеть в первоначальном виде внутри элемента style. CSS-файлы удобно использовать для хранения общих стилей, которые не относятся к какому-либо компоненту.

Если же CSS-файл — модуль, подключить его можно уже знакомым способом:

Скопировать кодJSX

import styles from './app.module.css';

Теперь директива работает иначе. Webpack преобразует и хеширует селекторы, которые содержатся внутри файлов модулей. В этом случае хеш схож с отпечатками пальцев. Как бы ни были похожи два человека, их отпечатки различаются, это врождённая особенность. Так и хеш селекторов определяются модулем, в котором описан селектор.

Если заглянуть в исходный код страницы во время работы приложения, то содержимое таких файлов можно увидеть с хешированными селекторами внутри элемента style.

Посмотрим, что содержит переменная styles:

Скопировать кодJSON

{

app: "app\_app\_\_llV42"

}

Переменная styles — объект, в котором располагаются:

* ключ — название селектора .app в CSS-модуле;
* значение — хешированная запись селектора .app, в которой app\_app\_\_llV42 — название импортированного модуля, app\_app\_\_IIV42 — название селектора, а app\_app\_\_llV42 — хеш.

Такая запись значения селектора — редактируемый параметр. Но мы рекомендуем оставить его без изменений.

Теперь попробуем воспользоваться стилями, которые предоставляет модуль app. Для этого элементу в className запишем путь до нужного селектора из объекта styles:

Скопировать кодJSX

import React from 'react';

import styles from './app.module.css';

*// Подключили модуль app;*

import Header from '../header/header';

import DogImage from '../dog-image/dog-image';

class App extends React.Component {

render() {

return (

<div className={ styles.app }>

{/\* Указали в значении путь до селектора; \*/}

<Header />

<DogImage />

</div>

)

}

}

export default App;

Подключить селектор из модуля достаточно просто. Для этого укажите в className ключ хешированного значения селектора. В нашем случае это свойство app.

Продолжим стилизовать проект и займёмся компонентом DogImage. Перед этим немного его усложним: добавим название мема и год, когда мем набрал популярность.

Скопировать кодJSX

class DogImage extends React.Component {

render() {

return (

<div>

<img

src={ this.props.image }

alt='Грустная собачка'

/>

<h2>{ this.props.name }</h2>

<span>{ this.props.description }</span>

</div>

);

}

}

Теперь создадим новый CSS-модуль в директории dog-image и добавим в него немного CSS-правил для элементов:

Скопировать кодCSS

*/\**

*Стили самой карточки*

*\*/*

.card {

display: flex;

flex-direction: column;

align-items: center;

}

*/\**

*Стили изображения*

*\*/*

.image {

max-width: 400px;

}

*/\**

*Название*

*\*/*

.title {

font-size: 25px;

color: #393a34;

font-family: 'Helvetica', Arial, sans-serif;

margin: 5px 0;

}

*/\**

*Дата появления*

*\*/*

.description {

font-size: 14px;

font-style: italic;

color: #393a34;

font-family: 'Helvetica', Arial, sans-serif;

margin: 5px 0;

}

Просто насладитесь таким неймингом для классов после БЭМ. Вам больше не нужно думать о том, как назвать элемент. Не нужно думать и о выделении элемента в отдельный блок и миксах со схожими БЭМ-сущностями. Полная свобода от методологии говорит о том, что теперь CSS — ваш друг. И не ведёт себя как школьник с задней парты, который только и ждёт, когда вы ошибётесь на проекте с десятком тысяч строк CSS.

Вернёмся к работе и подключим новый модуль к нашему компоненту:

Скопировать кодJSX

import React from "react";

import dogImage from "../dog-image/dog-image.module.css";

class DogImage extends React.Component {

render() {

return (

<div className={dogImage.card}>

<img

className={dogImage.image}

src={this.props.image}

alt="Грустная собачка"

/>

<h2 className={dogImage.title}>{this.props.name}</h2>

<span className={dogImage.description}>{this.props.description}</span>

</div>

);

}

}

export default DogImage;



*Какой же он классный!*

У селекторов title и description очень похожие CSS-свойства. Здорово было бы не дублировать лишний раз этот CSS-код. Но не спешите создавать отдельный БЭМ-блок и выносить туда все общие стили, ведь на помощь спешит ключевое слово composes. Оно предлагает npm-пакет css-modules и позволяет селектору использовать CSS-свойства другого селектора в качестве зависимости. Да так, что мы можем переопределять необходимые CSS-правила.

Попробуем установить зависимость description от title и посмотрим на изменения в итоговом коде:

Скопировать кодCSS

.card {

display: flex;

flex-direction: column;

align-items: center;

}

.image {

max-width: 400px;

}

.title {

font-size: 25px;

color: #393a34;

font-family: 'Helvetica', Arial, sans-serif;

margin: 5px 0;

}

.description {

composes: title;

*/\* В значении composes указывается название селектора. \*/*

font-size: 15px;

font-style: italic;

}

Если обратиться к переменной dog-image, в которой находятся хешированные селекторы, можно увидеть следующее:

Скопировать кодJSON

{

card: "dog-image\_card\_\_11D0F"

description: "dog-image\_description\_\_1IUew dog-image\_title\_\_3mTse"

image: "dog-image\_image\_\_2fGQ8"

title: "dog-image\_title\_\_3mTse"

}

Получается, теперь к HTML-элементу добавляется сразу два класса: dog-image\_description\_\_1IUew и dog-image\_title\_\_3mTse. Другими словами, composes работает на основе каскадности CSS-правил. А значит, зависимое CSS-правило .description должно быть объявлено после .title.

На этом возможности композиции не заканчиваются. Мы можем обращаться не только к селекторам внутри одного модуля, но и к селекторам других модулей с помощью схожей с import директивы:

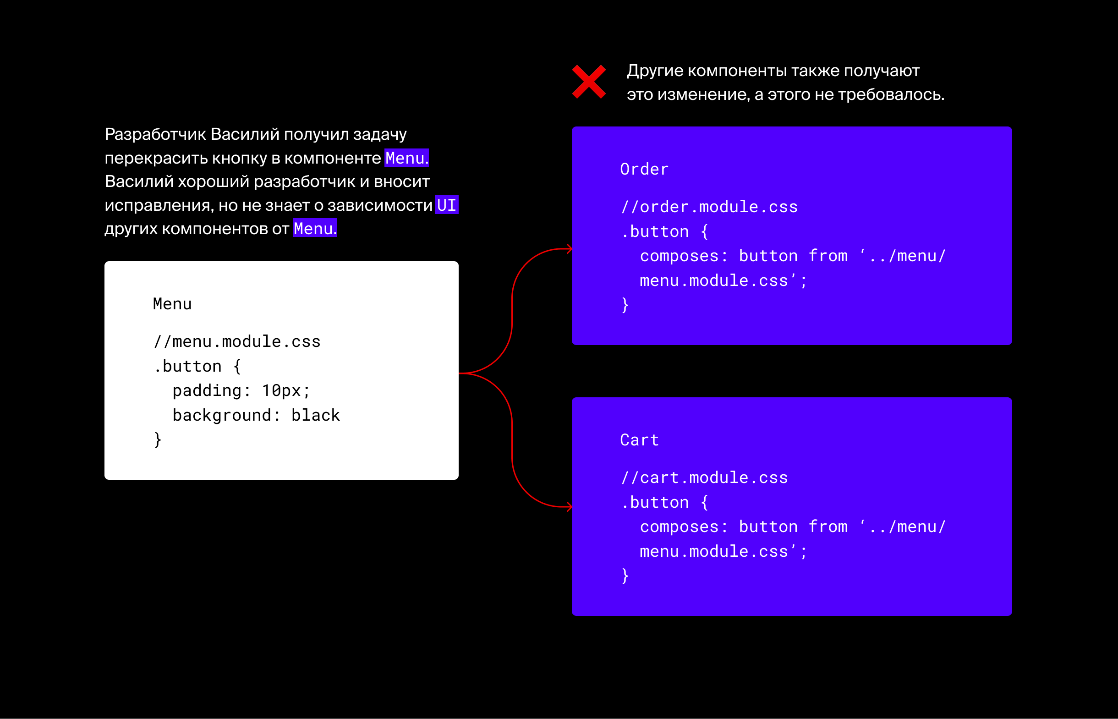
Скопировать кодCSS

.selector {

composes: anotherSelectorName from "./yet-awesome-file.module.css"

}

Но рекомендуем не злоупотреблять этой директивой. CSS-модули, из которых таким образом можно импортировать селекторы, должны находиться отдельно, а не входить в состав функциональных компонентов. Будет не совсем корректно, если селектор .button из компонента Menu экспортируется в несколько других компонентов. И если потребуется изменить кнопку в компоненте Menu— все другие кнопки получат эти изменения.



На этом мы заканчиваем изучение CSS-модулей в основной программе. А если вы настолько с ними сроднились, что не переживёте разлуки, — зайдите в дополнительную программу. В ней расскажем про css-in-js, который перевернёт представление о написании стилей. Техника css-in-js предоставляет возможность не наследовать свойства от родительских элементов и использовать единое пространство для написания CSS- и JS-кода.

А пока вам предстоит закрепить знания на практике — перейдём к задачам.

# Использование шрифтов

В предыдущих уроках вы подключили стили и добавили изображения на лендинг. Пора подумать о шрифтах. Мемный лендинг «не взлетит» с классическим шрифтом Helvetica. Почти любой проект требует подключения шрифтов, которые подобрал дизайнер. Их много в открытом доступе. Чтобы найти шрифты, удобно использовать библиотеки, к примеру [Google Fonts](https://fonts.google.com/).

Потенциальные пользователи приложения проголосовали за шрифт OpenSansCondensed, поэтому попробуем подключить его к проекту. Скачаем шрифт и разместим файл с ним в новой директории fonts/:

Скопировать кодMARKDOWN

└── src/

├── components

├── images

├── fonts/

│ ├── OpenSansCondensed-Bold.woff

│ ├── OpenSansCondensed-Light.woff

│ └── OpenSansCondensed-LightItalic.woff

└── <!-- остальные файлы в директории src -->

Подключать шрифты можно как в каждом компоненте, так и сразу в index.css. Первый вариант подходит для очень крупных приложений, в которых используется много разных шрифтов и важно максимально изолировать компоненты друг от друга. Нам больше подойдёт второй вариант: приложение небольшое, а шрифт подключаем только один.

Для подключения воспользуемся классической директивой @font-face. Поскольку мы используем CRA со всеми преимуществами Webpack, внутри директивы @font-face укажем относительный путь к файлу шрифта. Это нужно, чтобы при сборке все файлы оказались в правильном месте. Добавим директиву @font-face в файл index.css:

Скопировать кодCSS

*/\* index.css \*/*

*/\* Другие стили в index.css \*/*

@font-face {

font-family: 'OpenSansCondensed';

font-style: normal;

font-weight: 700;

src: url("./fonts/OpenSansCondensed-Bold.woff") format("woff");

}

@font-face {

font-family: 'OpenSansCondensed';

font-style: normal;

font-weight: 300;

src: url("./fonts/OpenSansCondensed-Light.woff") format("woff");

}

@font-face {

font-family: 'OpenSansCondensed';

font-style: italic;

font-weight: 300;

src: url("./fonts/OpenSansCondensed-LightItalic.woff") format("woff");

}

Применим этот шрифт к селектору .title в модуле dog-image.module.css:

Скопировать кодCSS

.card {

display: flex;

flex-direction: column;

align-items: center;

}

.image {

max-width: 400px;

}

.title {

font-size: 25px;

color: #393a34;

font-family: 'OpenSansCondensed', Arial, sans-serif;

margin: 5px 0;

}

.date {

composes: title;

font-size: 15px;

font-style: italic;

font-weight: 300;

}

Обновим страницу:



Проект изменился не сильно, но пользователи будут довольны

Работать со шрифтами можно по-разному. Например, мы могли бы создать отдельный CSS-файл в директории fonts/, подключить там шрифт и импортировать этот файл в index.js или конкретный компонент. А если в проекте используется всего один шрифт, то можно определить подключённый шрифт в свойстве font-family в index.css. Но такой вариант не очень подходит для возможного масштабирования.

В проектной работе можно применять любой из методов, которые мы рассмотрели.

В этом уроке вы разобрались с инфраструктурой проекта — подключили изображения, шрифты и CSS. Дальше познакомитесь с инструментами отладки React-приложения.

# Отладка приложения. Установка расширения React DevTools

Наверняка вы уже пробовали отлаживать код в веб-браузере. Всё это применимо и к проектам на React. Можно использовать console.log и debugger для отладки приложения внутри методов компонентов, в том числе и метода render(). Это ничем не отличается от отладки JavaScript-кода. Но React имеет ряд концепций, для отладки которых требуются дополнительные инструменты. Один из них — React DevTools.

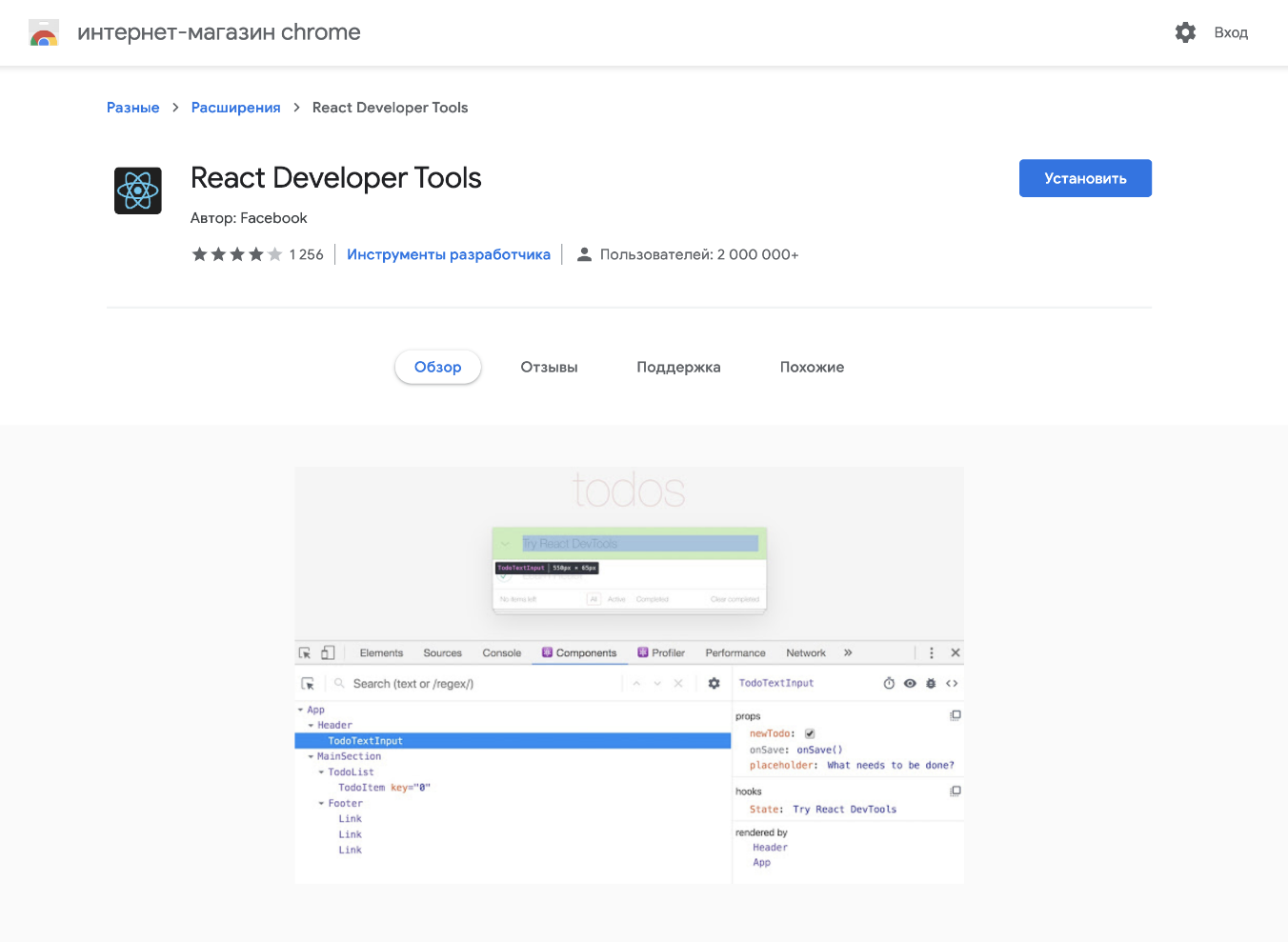
[React DevTools](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=ru) — популярное расширение для браузера, которое предоставляет больше возможностей в работе с React. В этом уроке мы установим расширение, а в следующем научимся им пользоваться. Установить расширение достаточно просто, поэтому урок станет для вас передышкой.

На данный момент расширение React DevTools доступно для браузеров Firefox, Яндекс.Браузер и Chrome.

## Установка в Яндекс.Браузере и Chrome

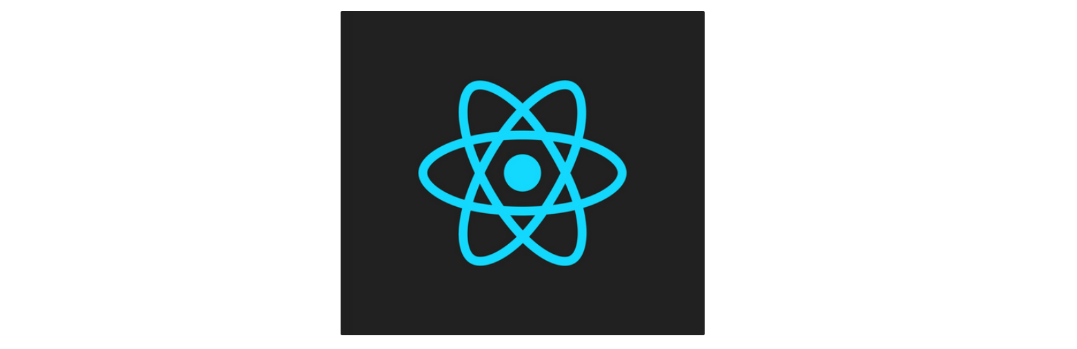
[Перейдите по ссылке в браузере](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=ru).

Вы увидите такую страницу:



Нажмите кнопку «Установить».

Появится всплывающее окно. Нажмите «Добавить расширение». Через несколько секунд появится ещё одно окно, подтверждающее, что расширение React DevTools успешно установлено. В правом верхнем углу (справа от адресной строки) вы увидите такую иконку:



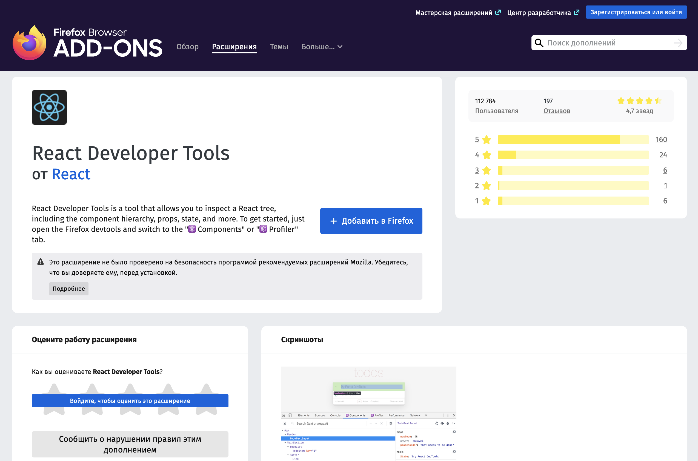
Иконка становится цветной на сайтах c React

Если иконка появилась, поздравляем, дело сделано! Переходите к следующему уроку.

## Установка в Firefox

[Перейдите по ссылке в браузере](https://addons.mozilla.org/ru/firefox/addon/react-devtools/).

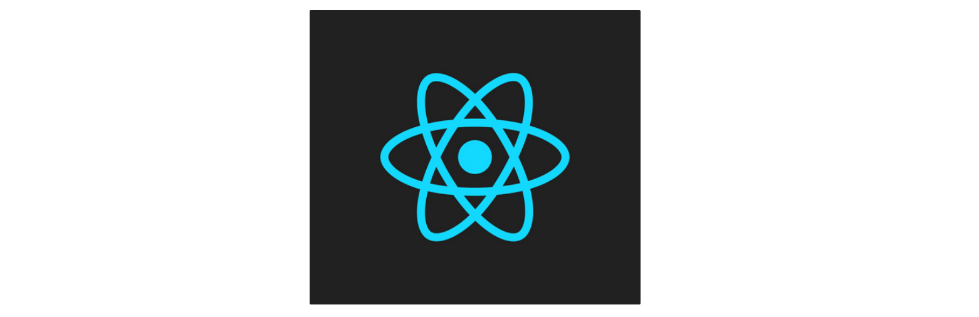
Вы увидите такую страницу:



Нажмите кнопку «Добавить в Firefox».

Появится всплывающее окно, которое запрашивает разрешение добавить расширение в браузер. Нажмите «Добавить».

Через несколько секунд появится ещё одно окно, подтверждающее, что расширение успешно установлено. Нажмите «Ок, понятно». После этого вы увидите, что в верхнем правом углу браузера появилась такая иконка:



Иконка становится цветной на сайтах c React

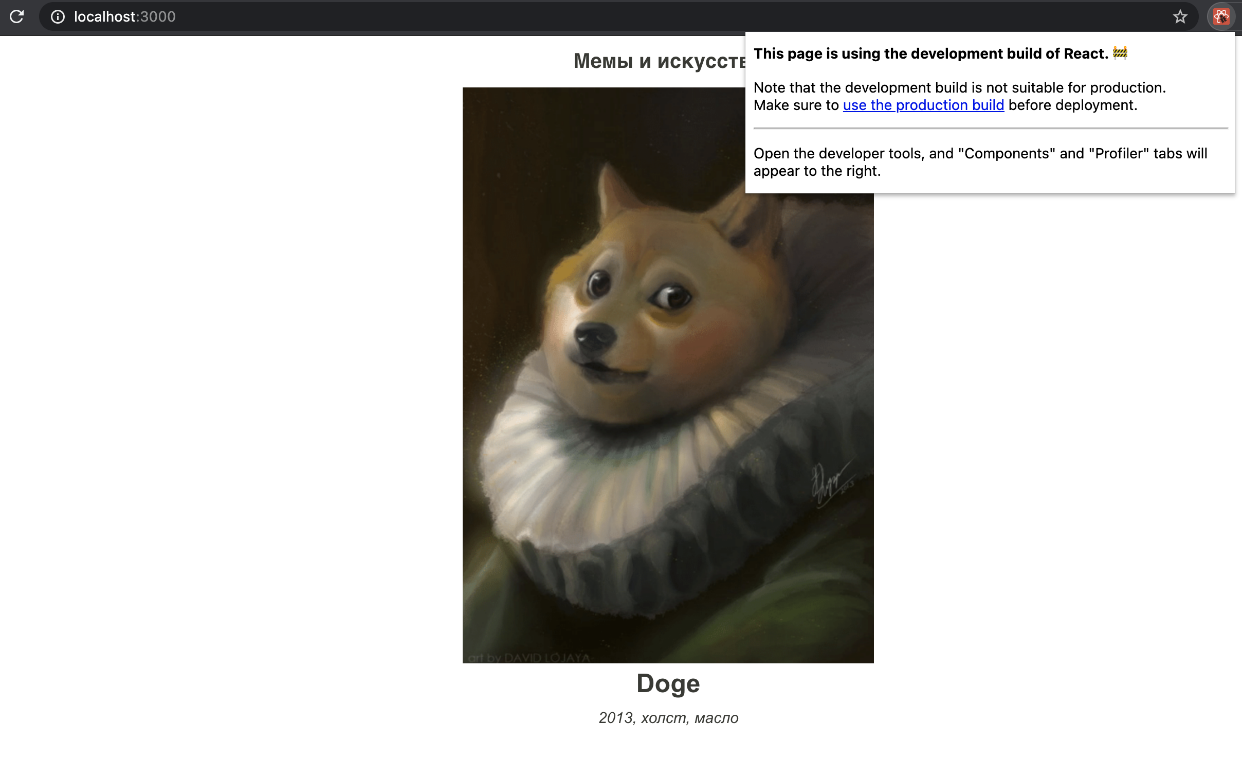
Если видите эту иконку, значит, вы справились!

В следующем уроке научимся работать с этим расширением.

# Отладка приложения. Использование расширения React DevTools

Теперь, когда расширение React DevTools установлено, вернёмся к мемам. Запустим приложение с помощью npm run start и перейдём по адресу <http://localhost:3000/>. Иконка React DevTools стала красной. Чтобы узнать почему, кликните по ней.

Появится модальное окно React DevTools с сообщением: «На этой странице используется сборка React для разработки». Значит, расширение обнаружило проект:



Это нормально

Внизу модального окна появится текст: «Откройте инструменты разработчика. Вкладки “Components” и “Profiler” появятся справа».

Перед тем как рассмотреть инструменты разработчика подробнее, добавим немного интерактива в проект — возможность лайкнуть карточку с мемом. Это поможет оценить все возможности React DevTools. Так выглядит компонент DogImage с новой функциональностью:

Скопировать кодJSX

import dogImage from "../dog-image/dog-image.module.css";

import React from "react";

class DogImage extends React.Component {

*// Опишем внутреннее состояние isLiked*

state = { isLiked: false }

*// Опишем метод изменения состояния isLiked*

onIconClick = () => {

this.setState({ isLiked: !this.state.isLiked })

};

render() {

return (

<div className={ dogImage.card }>

<img className={ dogImage.image } src={ this.props.image } alt='Грустная собачка' />

{/\* Добавим стили и разметку для кнопки лайка \*/}

<button onClick={ this.onIconClick } className={ dogImage.button } type='button' >

<svg className={ dogImage.icon } width="22" height="19" viewBox="0 0 22 19">

<path

d="M20.2991 1.68186C22.567 3.90213 22.567 7.54338 20.2991 9.78586L10.9804 19L1.6841

9.80806C0.606277 8.72013 0 7.27695 0 5.74496C0 4.21297 0.583823 2.76979 1.6841

1.68186C3.92957 -0.560619 7.61215 -0.560619 9.88007 1.70406L10.9804 2.792L12.0806

1.68186C14.3486 -0.560619 18.0311 -0.560619 20.2991 1.68186Z"

fill={ this.state.isLiked ? "red" : "white" } />

</svg>

</button>

<h2 className={ dogImage.title }>{ this.props.name }</h2>

<span className={ dogImage.date }>{ this.props.date }</span>

</div>

);

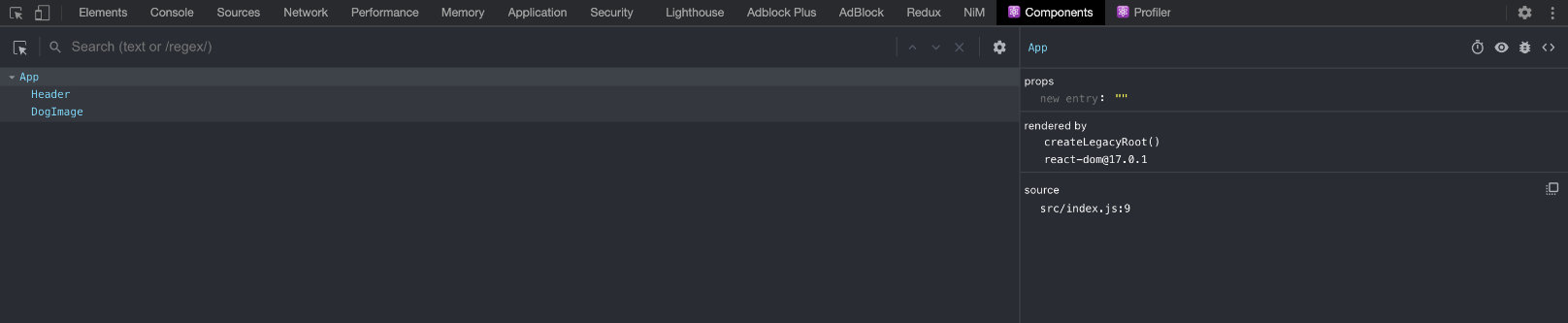
}

}

## Использование React DevTools в браузере

В инструментах разработчика находится много вкладок. С некоторыми из них вы уже знакомы, например Console и Elements или Inspector в Chrome. Рассмотрим остальные вкладки.

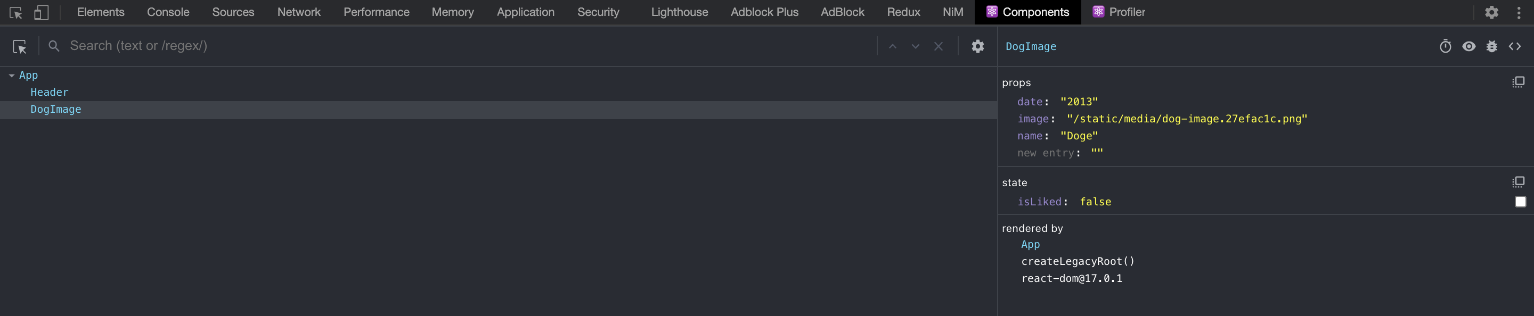
После установки расширения React DevTools появятся две вкладки: Components и Profiler. Сначала разберём вкладку Components. Вот что покажет инструмент, когда будет загружено приложение:



Вкладка Components

Отобразится дерево компонентов проекта, в котором можно получить подробную информацию о каждом компоненте. Рассмотрим на примере компонента DogImage.

Подробная информация о компоненте находится внизу вкладки:



И никакой *console.log(props)* не нужен

Здесь можно посмотреть расположение файла, изучить пропсы компонента, его текущее состояние state и какой компонент отвечает за его отрисовку. В нашем случае компонент App отрисовывает DogImage.

Особого внимания заслуживают значения стейта компонента, которые обновляются в режиме реального времени:

Во вкладке Components есть ещё несколько кнопок: например, чтобы приостановить работу компонента там, где это возможно, или взглянуть на ту часть DOM-дерева, в которой находится компонент. Ещё есть кнопки для вывода данных о компоненте в консоль и для просмотра его исходного кода.

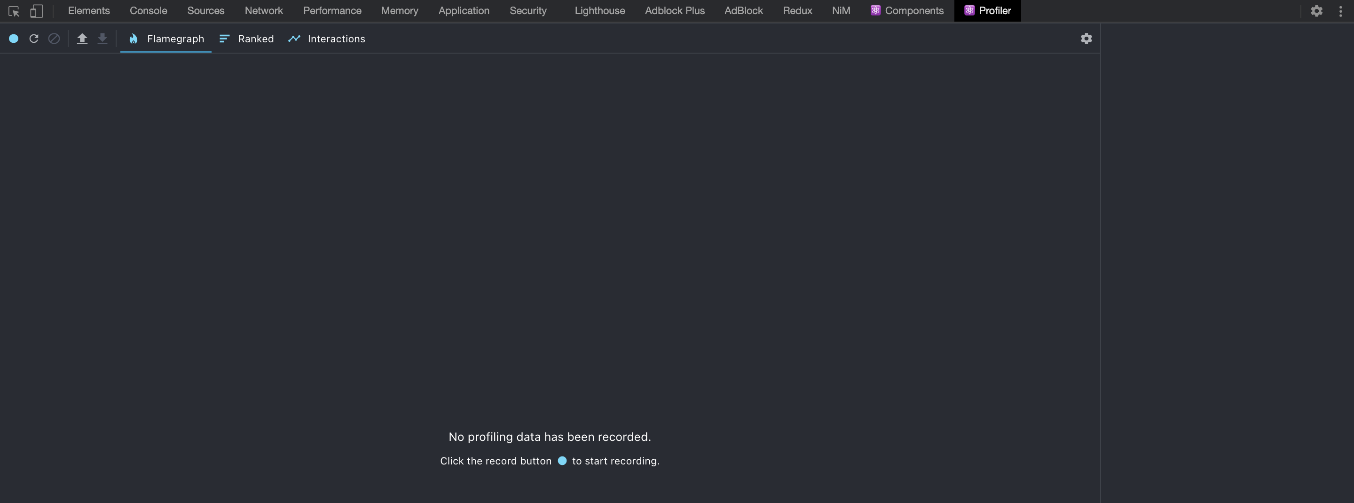
Если использовать эти расширенные возможности вместе с console.log и debugger, можно точнее анализировать изменения состояния и оперативно отслеживать и устранять баги в проектах на React.

## Вкладка Profiler в React DevTools

Поговорим о функции Profiler. Она позволяет детально изучить приложение: отслеживает, как часто оно отрисовывается, указывает, какие части приложения могут тормозить его работу.

Поскольку приложение небольшое, мы не сможем в полной мере ощутить преимущества этой функции. Но в больших проектах без неё не обойтись, ведь она помогает оптимизировать работу приложения.

Откроем вкладку Profiler:



Выглядит пустовато. Но за этой пустотой скрывается множество возможностей

Чтобы начать собирать данные, нужно нажать на кнопку записи. Сейчас есть только одна кнопка лайка. Когда пользователь нажимает на неё, карточка отрисовывается заново, а пользователь видит лайкнутую кнопку.

Чтобы перестать записывать данные, нужно ещё раз нажать на кнопку записи.

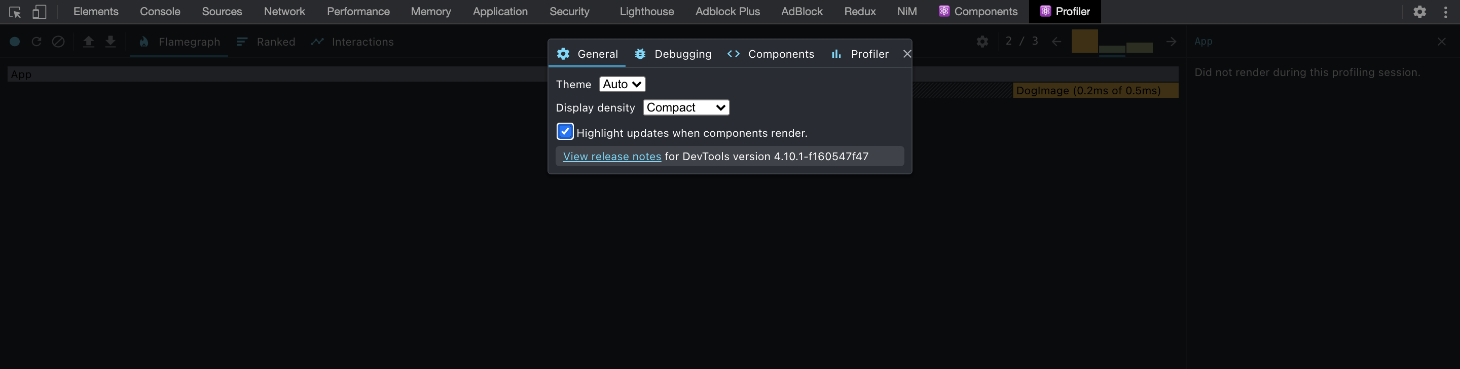
Попробуем собрать данные о проекте:

Выглядит круто. Но можно добавить ещё больше возможностей.

Разберёмся, что произошло. В момент начала сбора данных компоненты App, Header, DogImage уже присутствовали на странице. После остановки записи и при наведении на компоненты App и Header видно, что эти компоненты не монтировались повторно.

С DogImage немного другая история. Мы несколько раз нажали на иконку лайка, что привело к повторной отрисовке компонента DogImgage. Если нажать на прогресс-бар компонента, в правом блоке отобразится информация о каждом повторном монтировании: когда оно случилось и сколько времени заняло. Сейчас это кажется незначительным, но производительность проекта — важный показатель больших приложений. Вы также будете разрабатывать большое приложение в рамках проектных работ, поэтому этот навык поможет найти «дорогие» по производительности компоненты.

На этом базовые возможности React DevTools не заканчиваются. Часть функциональности доступна и без запуска Profiler. Для этого нужно перейти в настройки на одной из вкладок Components или Profiler, а после — отметить чекбокс со скриншота:



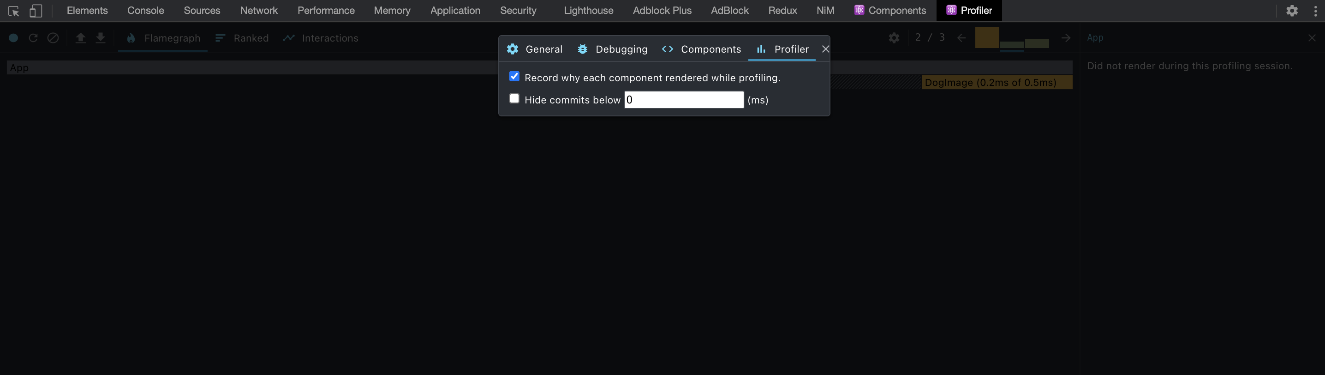
“Highlight updates when components render” — нужный нам чекбокс

Чекбокс “Highlight updates when components render” будет подсвечивать все изменяющиеся компоненты в процессе взаимодействия с проектом:

Теперь можно отслеживать простые изменения рендера компонента без Profiler.

Такой подход очень удобен и не требует постоянной работы с Profiler. Информация о производительности будет недоступна, зато можно быстро найти некоторые баги. Например, когда половина приложения внезапно начнёт повторно рендериться без особой необходимости. React быстрый, но это не значит, что все баги можно отдавать ему на откуп.

Последнее, чем мы займёмся, — вернёмся в настройки и отметим другой чекбокс:



“Record why each component rendered while profiling” — расскажет о причине рендера компонентов

Чекбокс “Record why each component rendered while profiling” отображает в меню компонента после сбора данных причину его рендера:

Теперь мы видим и количество отрисовок компонента DogImage, и причину, по которой они произошли. Полезно ли это сейчас? Не совсем. Будет ли это полезно в дальнейшем? Однозначно.

В этом уроке вы изучили инструментарий для отладки проектов на React. Пока мы работали только со сборкой для разработки. В следующем уроке расскажем, как подготовить проект к запуску.

**Инфраструктурные скрипты. Сборка проекта**

Во всех предыдущих уроках мы разрабатывали проект. Теперь пора его запускать. Для начала нужно собрать проект. В этом уроке расскажем, что для этого нужно.

Внутри файла package.json, который создан с помощью CRA, хранятся четыре готовых скрипта:

Скопировать кодJSX

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test",

"eject": "react-scripts eject"

},

Мы уже использовали скрипт "start", когда запускали проект на локальном сервере. А вот про скрипты "test" и "eject" не сказали ни слова. Разберёмся, что они делают:

* npm run test — исполняет все написанные файлы тестирования. Мы ещё не занимались вопросом тестирования, поэтому вернёмся к этой команде немного позже.
* npm run eject — после запуска команды CRA необратимо копирует все инфраструктурные файлы конфигурации из пакета react-scripts в сам проект и редактирует package.json файл. Назовём её «красной кнопкой». Мы сообщаем CRA о том, что перед ней крутой разработчик, которой не прочь сам изменять конфигурацию Babel, Webpack и ESLint. Но мы только начали изучать React, поэтому на красную кнопку нажимать не будем.

Остался последний скрипт, который отвечает за сборку проекта, — "build". При запуске команды npm run build через консоль скрипт генерирует оптимизированную сборку проекта. На практике это означает, что внутри проекта появится новая папка build/. Внутри build/static можно найти оптимизированные версии всего кода, наряду с другими ресурсами: JS, CSS и шрифтами.

Затем файлы из директории build можно разместить на любом доступном сервере. Иными словами, эти файлы можно использовать для деплоя веб-приложения в интернете.

Чтобы посмотреть, что получится, запустим проект из папки build/ на локальном сервере. Для этого понадобится установить ещё один пакет. Из командной строки запустим команду npm i -g serve. Пакет установлен в вашей системе глобально — так же, как вы установили ранее Create React App.

Если вы используете macOS, установите пакет глобально командой sudo npm i -g serve и введите пароль.

Далее остаётся только запустить serve -s build. В консоли отобразится сообщение, что проект выполняется на локальном сервере — по умолчанию адресом будет localhost:3000. Введите этот адрес в браузере и увидите свой проект!