**Инструментарий «Реакта». Введение**

Из предыдущей темы вы узнали много нового о «Реакте». Научились создавать и отрисовывать компоненты, а также изучили основы нового языка JSX. Пока «Реакт» может казаться чем-то внеземным, и это нормально. Главное — помнить, что в основе всех механизмов движка лежат те же принципы JavaScript, что вы усердно изучали ранее.

В этой теме вы изучите инструменты и приёмы, которые оптимизируют и упрощают процесс разработки, объединения проектов, помогают находить и исправлять ошибки. В конце узнаете, как подготовить и настроить сборку проекта, написанного на «Реакте».

Некоторые из уроков помогут пересмотреть уже знакомые концепции, другие же погрузят глубже в экосистему «Реакта».

Уже в следующем уроке вы познакомитесь с полезной утилитой Create React App, которая помогает быстро создавать и запускать проекты на «Реакте».

# Знакомство с Create React App

Немного пофантазируем.

Представьте, что вы задумались о новом увлечении и решили построить миниатюрный городок с железной дорогой. Начинание серьёзное: вы тщательно выбираете подходящие краски и материалы, аккуратно раскрашиваете детали и бережно склеиваете их вместе. Когда отдельные конструкции готовы, можно приступать к прокладке рельсов и сборке. Уже не терпится запустить готовый поезд и показать городок друзьям.

Новое занятие приносит массу веселья, и однажды вы задумываетесь: это может понравиться людям. Из этого можно сделать бизнес! Клиент бы описывал город своей мечты, показывал эскизы или фотореференсы, а вы бы воплощали всё это в реальность.

Но вот незадача: первый проект потребовал целых две недели. Собирать всё вручную с нуля просто нерентабельно. Чтобы сэкономить время, вы решаете использовать заранее собранные компоненты: дома, деревья, лавочки и так далее. Вам останется лишь покрасить их и разместить в соответствии с требованиями заказчика. Вы отказались от более тонкого уровня контроля в пользу эффективности.

# Зачем использовать Create React App

Использование заранее собранных моделей сэкономило бы много времени бизнесу по производству миниатюрных городов. Так и в программировании, утилита Create React App (CRA) помогает разработчикам писать код быстрее и эффективнее.

Как и «Реакт», она разработана командой Facebook. CRA позволяет быстро собрать и запустить проект, создав его на основе стандартного шаблона, не беспокоясь о детальной настройке.

Основные причины применять CRA:

* Одной командой можно развернуть полноценный рабочий проект.
* Проект уже содержит удобную файловую структуру, стандартную для большинства проектов на «Реакте».
* В проект легко подключить статические ресурсы: изображения, шрифты и так далее.
* Есть готовый механизм для тестирования компонентов.
* Проект будет автоматически обновляться в браузере при внесении изменений.
* Настроена оптимизация продакшн-сборки.
* Можно использовать NPM и легко устанавливать пакеты.

Есть и особое преимущество, о котором поговорим отдельно. Вспомним пример из предыдущей темы. Весь наш новый код на «Реакте» был написан внутри одного тега script:

Скопировать кодJSX

<script type="text/jsx">

class Switch extends React.Component {

*// Конструктор компонента Switch, обработчик и функция render*

}

ReactDOM.render((

<>

<Switch title="Счастье" color="blue" isActive={true} />

<Switch title="Любовь" color="orange" isActive={false} />

<Switch title="Шаурма" color="green" isActive={false} />

</>

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

</script>

Если продолжать писать код подобным образом, файлы станут большими, а их содержимое — трудным для восприятия. Но есть современное решение этой проблемы, с которым вы уже знакомы. Можно разделить код на модули и использовать конструкции export и import.

Модули — новый синтаксис ES6, они требуют дополнительной подготовки для правильной работы кода. Create React App уже включает в себя «Вебпак» и «Бабель» со всеми необходимыми настройками, что сэкономит уйму времени на их конфигурации.

Благодаря CRA мы можем переписать наш код с использованием import:

Скопировать кодJSX

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

import ReactDOM from 'react-dom'

class Switch extends React.Component {

*// Конструктор компонента Switch, обработчик и функция render*

}

ReactDOM.render((

<>

<Switch title="Счастье" color="blue" isActive={true} />

<Switch title="Любовь" color="orange" isActive={false} />

<Switch title="Шаурма" color="green" isActive={false} />

</>

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

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

В теории настроить всё это можно было и вручную, но CRA позволяет существенно ускорить этот процесс и, что немаловажно, учитывать при этом общепринятые подходы.

# Установка Create React App

Это несложно и похоже на то, как вы уже устанавливали зависимости пакетов NPM.

Если вы до этого установили глобально CRA через npm install -g create-react-app, переустановите его. Для этого используйте npm uninstall -g create-react-app или yarn global remove create-react-app. В дальнейшем используйте npx для установки и работы с проектом.

Вот и всё! В следующем уроке создадим новый проект с использованием Create React App.

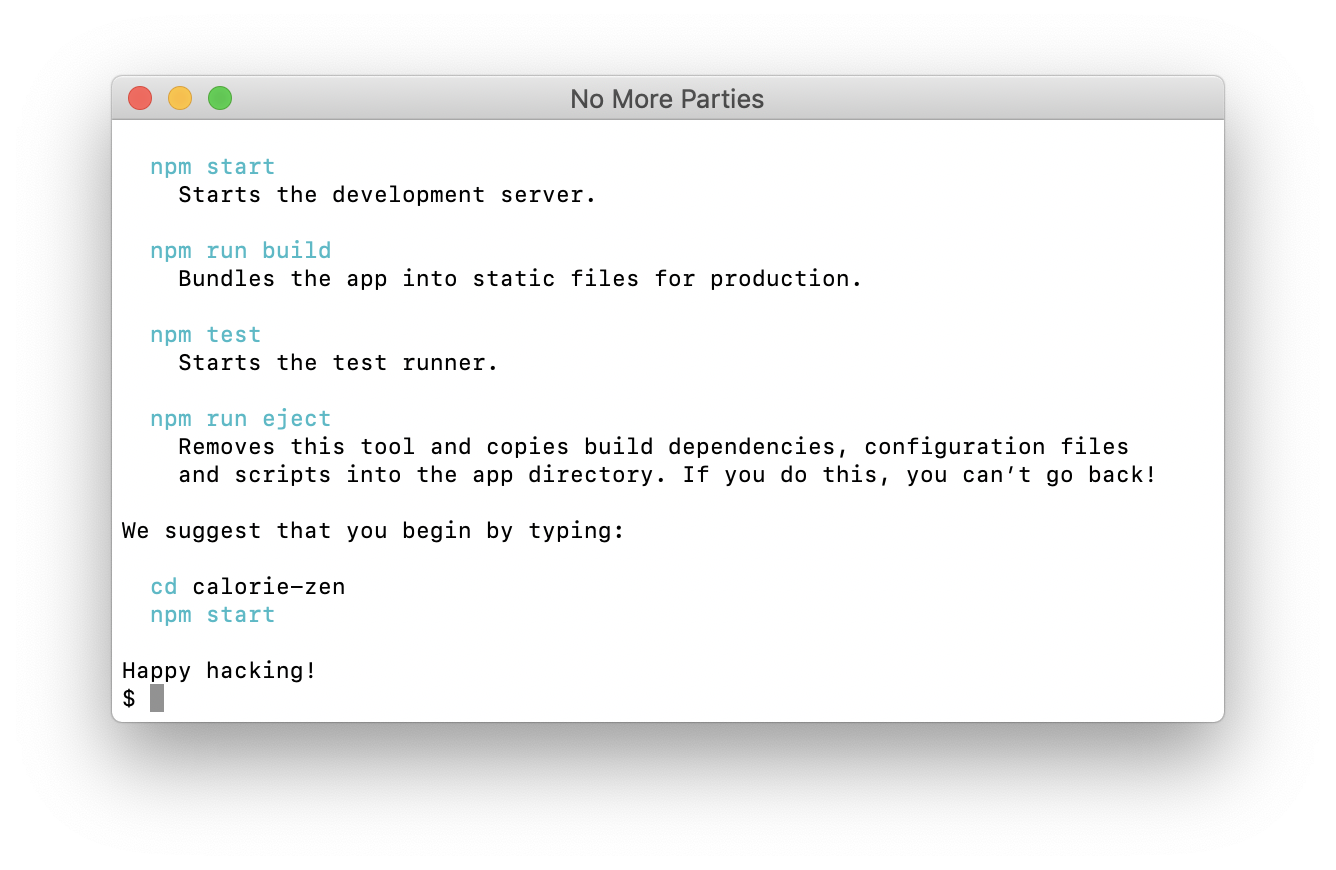
# Новый проект

В этом уроке начнём разработку приложения Calorie-Zen. Оно призвано помочь пользователям соблюдать рацион и дневную норму калорий без лишнего стресса. Для начала создадим прототип. Чтобы быстро запустить проект, воспользуемся утилитой, которую установили в предыдущем уроке.

Откроем командную строку и перейдём в директорию, в которой хотим создать проект. Для создания нового CRA-проекта используют команду npx create-react-app название\_проекта. Название проектов обычно пишут латиницей. Наш проект называется calorie-zen, поэтому введём команду npx create-react-app calorie-zen.

После этого начинается магия CRA. Этот процесс может занять несколько минут — отличный повод сделать перерыв. А раз уж мы разрабатываем приложение для поддержания здорового образа жизни, можно выпить стакан воды.

После успешного выполнения скрипта появится такое сообщение:



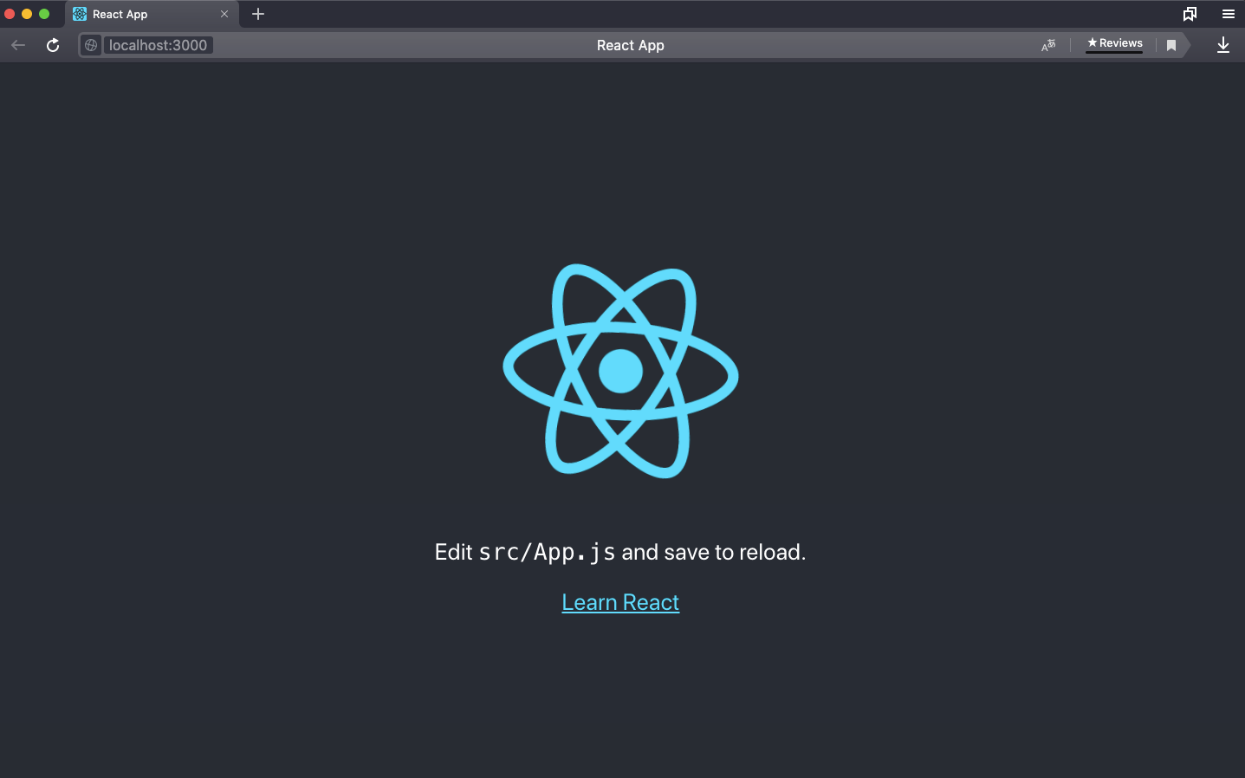
Теперь в вашем распоряжении проект, полностью готовый к работе. Запустим его.

# Запуск проекта

Перейдите в директорию проекта командой cd calorie-zen. Находясь в ней, введите ещё одну команду — npm start.

После этого проект автоматически откроется в браузере. Если этого не произошло, откройте проект в браузере, введя адрес http://localhost:3000/.

Сейчас проект выглядит так:



Уже неплохо, но у вас получится ещё лучше

Это шаблон проекта, который создаёт Create React App по умолчанию: простой компонент, состоящий из вращающегося логотипа и текста. Настоящему приложению потребуется совсем другое наполнение.

В следующем уроке разберём файловую структуру CRA-проекта.

# Распаковка проекта

В предыдущем уроке мы создали проект при помощи Create React App. В этом уроке разберёмся, что утилита Create React App ещё умеет. Поговорим об основных файлах и директориях, с которыми предстоит работать при разработке проекта.

Для начала посмотрим на стандартную файловую структуру проекта calorie-zen:

Скопировать код

└── calorie-zen/

├── node\_modules/

├── public/

├── src/

├── .gitignore

├── package.json

├── README.md

└── yarn.lock

Разберём файлы и директории, из которых она состоит. Обратите внимание на знакомые наименования: node\_modules/ и package.json. Значит, мы можем легко подключить новые пакеты к проекту с применением NPM. Так же, как мы делали это в предыдущих спринтах.

# Директория src/

Посмотрим, что находится в директории src/. Именно в ней вы будете совершать бóльшую часть работы над проектами:

Скопировать кодMARKDOWN

└── calorie-zen/

├── node\_modules/

├── public/

├── src/ <!-- откроем эту директорию! -->

│ ├── App.css

│ ├── App.js

│ ├── App.test.js

│ ├── index.css

│ ├── index.js

│ ├── logo.svg

│ ├── serviceWorker.js

│ └── setupTests.js

├── .gitignore

├── package.json

├── README.md

└── yarn.lock

Рассмотрим три важных файла:

1. Файл index.js — точка входа JavaScript. Внутри этого файла подключают корневой компонент App.
2. Файл App.js — корневой компонент приложения. О нём поговорим чуть позже.
3. CSS-файл App.css содержит стили корневого компонента App.js. В проекте на «Реакте» может быть много разных CSS-файлов: например, по файлу для каждого компонента. Мы легко подключим все эти стили к компонентам, используя возможности Create React App и ключевое слово import.

# Директория public/

Теперь изучим директорию public/:

Скопировать код

└── calorie-zen/

├── node\_modules/

├── public/ <!-- а теперь откроем эту директорию -->

│ ├── favicon.ico

│ ├── index.html

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── robots.txt

├── src/

├── package.json

├── README.md

└── yarn.lock

Разработка проекта ведётся в директории src/, после чего выполняется сборка проекта в результирующий код, который используют внутри HTML-документа. Обратите внимание на файл index.html в директории public/. Вам не придётся вносить много изменений в этот файл, однако иногда это будет нужно. Например, когда потребуется изменить заголовок приложения или метатег.

Директория public/ также содержит файл favicon.ico — иконку сайта, которая отображается во вкладке либо в закладках браузера.

Изображения лучше импортировать через JavaScript. Так мы с помощью хэширования избежим проблем с отображением изображений, когда придётся загрузить новые файлы с теми же названиями. Подробнее вы разбирали это [в теме про сборку проекта](https://praktikum.yandex.ru/trainer/web/lesson/ef8a1ed3-560d-4bf5-80f0-326a4f0b83e7). Это ещё одно преимущество Create React App: вся конфигурация настроена за нас.

# Файл App.js

App.js выполняет функции корневого компонента в нашем проекте. Каждый раз, когда мы создаём проект с помощью Create React App, автоматически появляется вот такой файл App.js:

Скопировать кодJSX

import React from 'react';

import logo from './logo.svg';

import './App.css';

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Edit <code>src/App.js</code> and save to reload.

{/\* Отредактируйте App.js и сохраните файл, чтобы

страница перезагрузилась. \*/}

</p>

<a

className="App-link"

href="https://reactjs.org"

target="\_blank"

rel="noopener noreferrer"

>

Learn React

{/\* Узнать больше о «Реакте» \*/}

</a>

</header>

</div>

);

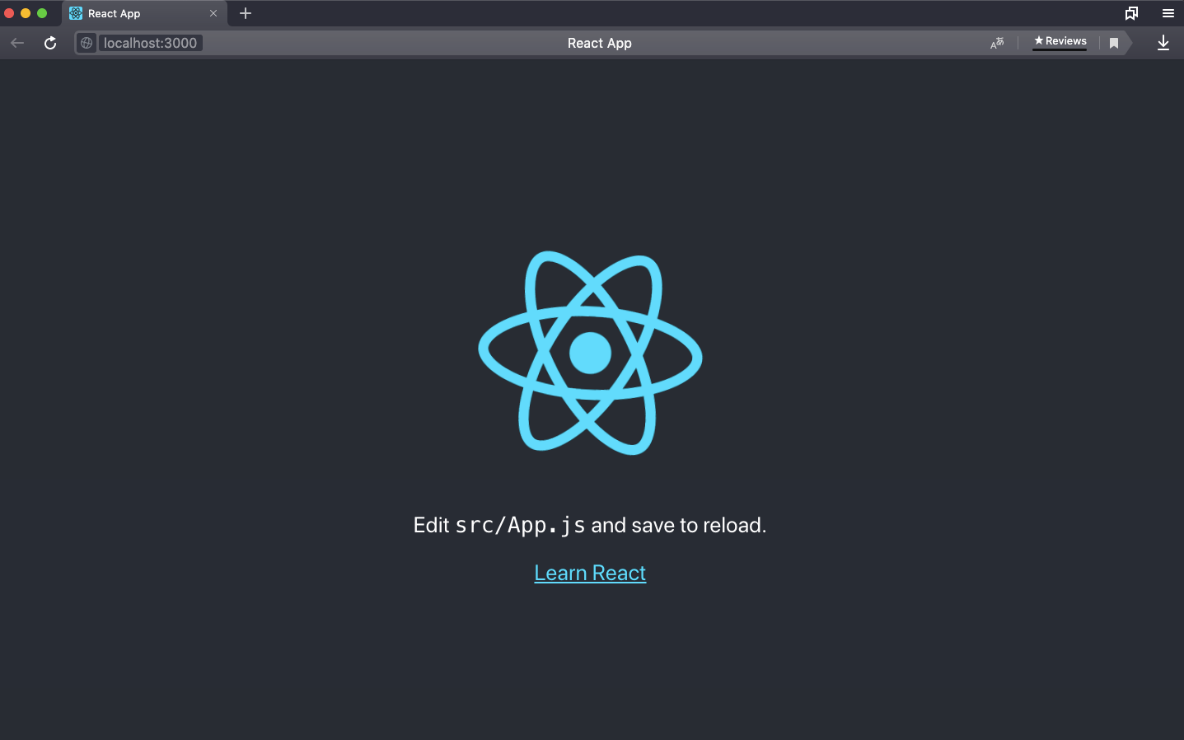
}

export default App;

Благодаря новым знаниям о JSX и компонентах, а также о том, как использовать ключевые слова import и export, такой код вам теперь не страшен.

И чтобы точно ни в чём не запутаться, стоить упомянуть, что мы импортируем библиотеку «Реакта» в начале файла инструкцией import React from 'react'. Вы сможете найти «Реакт» в разделе dependencies файла package.json. Если бы мы создавали проект с нуля без CRA, нам бы пришлось отдельно вводить в командную строку npm install react --save для установки React. Вы уже знаете, как импортировать модули и устанавливать пакеты из курса по JavaScript; здесь же всё это уже настроено и доступно в готовом шаблоне проекта CRA.

Код в файле App.js отвечает за основную часть финального отображения проекта, который мы видим при запуске скрипта npm start:



При разработке полноценного приложения нужно удалить всё стандартное содержимое. Вместо удалённых файлов вы напишите собственные. Научимся это делать на практике.

# Структура проекта

В предыдущем уроке мы разобрали базовую структуру проекта, созданного с помощью Create React App. Проекты со временем растут, поэтому важно подумать, как лучше их структурировать.

Скопировать код

└── easy-project/

├── index.html

├── scripts.js

└── styles.css

В этом случае всё понятно: HTML, JavaScript и CSS находятся в отдельных файлах. Если с проектом будут работать другие разработчики, они легко сориентируются в чётко выстроенной структуре и поймут, за что отвечает каждый файл.

Но если взглянуть на проект calorie-zen, даже в состоянии по умолчанию он намного больше проекта easy-project из примера. Вот в чём дело: наш проект будет разрастаться, а его файловая структура станет больше и сложнее. Если сейчас не позаботиться об организации работы, очень скоро ситуация выйдет из-под контроля.

Хорошо структурированный проект облегчит работу не только вам, но и другим разработчикам, которые в будущем будут работать с вашим кодом. Структура нашего проекта во многом базируется на продолжении тех практик, что мы уже используем. Мы внимательно подбираем названия функциям и переменным, оставляем комментарии в коде, чтобы сделать код понятным и упростить работу с проектом. В этом уроке разберёмся, что ещё нужно сделать, чтобы организовывать хорошую структуру в проекте.

## Экскурсия по проекту

Откроем директорию src/:

Скопировать код

└── src/

├── App.css

├── App.js

├── App.test.js

├── index.css

├── index.js

├── logo.svg

├── serviceWorker.js

└── setupTests.js

Рассмотрим файлы компонента приложения: App.js, App.css и App.test.js. Первые два мы уже затрагивали в прошлом уроке, но не изучали их подробно.

Мы называем файлы по правилу: они должны иметь то же имя, что и компонент. Название должно начинаться с заглавной буквы.

Файл App.test.js предназначен для автоматического тестирования компонента App.

По мере разработки приложения, вероятно, понадобится добавить в него новые компоненты. В нашем случае понадобятся такие:

1. ZenForm.js — простой компонент для формы, который позволит пользователям вносить информацию о приёмах пищи.
2. ZenImage.js — компонент для логотипа.
3. ZenTotal.js — компонент, который посчитает итоговую сумму калорий за день.

Корневым компонентом по-прежнему будет App.js. Для каждого нового компонента нужно создать соответствующие файлы со стилями, а также файлы для тестирования:

Скопировать код

└── src/

├── App.js

├── App.css

├── App.test.js

├── ZenForm.js

├── ZenForm.css

├── ZenForm.test.js

├── ZenImage.js

├── ZenImage.css

├── ZenImage.test.js

├── ZenTotal.js

├── ZenTotal.css

├── ZenTotal.test.js

├── index.css

├── index.js

├── logo.svg

├── serviceWorker.js

└── setupTests.js

Структура выглядит не слишком сложной — в ней точно не запутаешься, даже со всеми новыми файлами. Обычно готовые приложения состоят из гораздо большего количества компонентов.

# Как организовать проект?

Существует много разных способов организации файлов. Например, мы можем поместить все файлы со стилями в одну директорию, файлы для тестирования в другую, а обычные скрипты — в третью:

Скопировать код

└── src/

├── scripts/

│ ├── App.js

│ ├── ZenForm.js

│ ├── ZenImage.js

│ ├── ZenTotal.js

│ └── index.js

├── styles/

│ ├── App.css

│ ├── ZenForm.css

│ ├── ZenImage.css

│ ├── ZenTotal.css

│ └── index.css

├── test-files/

│ ├── App.test.js

│ ├── ZenForm.test.js

│ ├── ZenImage.test.js

│ └── ZenTotal.test.js

├── logo.svg

├── serviceWorker.js

└── setupTests.js

Такую структуру уже можно считать организованной. Но в ней станет сложно ориентироваться, когда код приложения вырастет. Лучше всего организовывать файлы, основываясь на функциях, которые они выполняют:

Скопировать код

└── src/

├── components/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ ├── zen\_form/

│ │ ├── ZenForm.js

│ │ ├── ZenForm.css

│ │ └── ZenForm.test.js

│ ├── zen\_image/

│ │ ├── ZenImage.js

│ │ ├── ZenImage.css

│ │ └── ZenImage.test.js

│ └── zen\_total/

│ ├── ZenTotal.js

│ ├── ZenTotal.css

│ └── ZenTotal.test.js

├── index.js

├── logo.svg

├── serviceWorker.js

└── setupTests.js

Мы добавили ещё одну директорию — components/. Директория components/ содержит в себе все компоненты. И в этой же директории мы создали для каждого компонента новые файлы. Когда приложение будет расти, мы будем уверены, что структура не сломается, ведь мы создали легко масштабируемый шаблон. Такую структуру будем использовать в дальнейшей работе над проектом.

Такая файловая структура не подойдёт идеально для всех проектов. В некоторых случаях придётся выбрать другой способ организации структуры проекта. Самое главное — всё, что вы выбираете, должно согласовываться между собой. Если ваш проект организован определённым образом, обязательно придерживайтесь выбранного стандарта, поскольку это облегчит работу не только вам, но и вашим товарищам по команде.

В следующем уроке создадим новый проект с нуля и организуем его структуру.

# Импорт модулей

Раньше вы уже создавали проекты, а теперь узнали, почему их так важно структурировать. В этом уроке вы узнаете о способах импортирования модулей в проект.

Немного отвлечёмся от подсчёта калорий и рассмотрим новый проект под названием «Империя матрасов». Нужно создать лендинг для клиента. Вы уже знаете, как быстро справиться с этой задачей: нужно запустить npx create-react-app emperor mattress и немного структурировать проект. Теперь посмотрим, как формируется директория src/:

Скопировать код

└── src/

├── components/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ └── landing/

│ ├── Bed.js

│ ├── Bed.css

│ ├── Header.js

│ ├── Header.css

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

├── images/

│ └── bed.png

└── <!-- дополнительные файлы src/ -->

На этот раз мы разместили компоненты в директории landing/. Мы также создали директорию images/ внутри директории src/.

Наша основная цель — создать лаконичную стартовую страницу с продающим заголовком и яркой картинкой. Для этого были созданы два новых компонента: Header и Bed. Поскольку это презентационные компоненты, далее следует использовать обычные функциональные компоненты. Мы по-прежнему будем применять App.js в качестве корневого компонента, так что первым делом нужно импортировать новые компоненты в App.js. Сделаем это, применив ключевое слово import.

# Импорт компонентов

Подумайте, как лучше импортировать новые компоненты, и посмотрите на код:

Скопировать кодJSX

import React from 'react';

import Header from './landing/Header'; *// прошу любить и жаловать компонент header*

import Bed from './landing/Bed' *// и окажите компоненту bed тёплый приём*

import './App.css';

function App() {

return (

<div className="App">

<Header />

<Bed />

</div>

);

}

export default App;

Чтобы импортировать Header и Bed в App.js, пропишем перед ними ключевое слово import, затем сошлёмся на них, используя [относительные пути к файлам в проекте](https://praktikum.yandex.ru/trainer/web/lesson/dbbba53a-4209-4c8d-8bab-55728d9eeb37/task/c75161f9-1f46-40d0-9645-1d3b658c19fd/).

Обратите внимание, в каждом из импортируемых компонентов также присутствует инструкция export default:

Скопировать кодJSX

*// Header.js*

import React from 'react';

function Header() {

return (

<h1>Империя матрасов</h1>

);

}

export default Header;

*// Bed.js*

import React from 'react';

function Bed() {

return (

<img alt='A comfortable bed'/>

);

}

export default Bed;

Начало положено. Если ввести npm start в командную строку, браузер покажет результат автоматически. Если по какой-то причине этого не произошло, введите вручную адрес localhost:3000 в браузере.

Хм, результат выглядит, мягко говоря, аскетично и даже картинка не видна. Но это только начало. В следующем уроке вы узнаете, как можно оживить проект и придать ему стиль.

# Импорт изображений

В предыдущем уроке мы создали новые компоненты, затем импортировали их в корневой компонент, применив ключевое слово import. Сейчас начальная страница приложения — это заголовок на белом фоне. В этом уроке добавим на неё изображение.

# Добавление изображений

Как и в случае с другими модулями, воспользуемся инструкцией import для импортирования изображений. Посмотрим на содержимое директории src/:

Скопировать кодMARKDOWN

└── src/

├── components/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ └── landing/

│ ├── Bed.js

│ ├── Bed.css

│ ├── Header.js

│ ├── Header.css

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

├── images/

│ └── bed.png

└── <!-- дополнительные файлы src/ -->

Изображение bed.png нужно импортировать в Bed.js. Сделаем это внутри компонента:

Скопировать кодJSX

import React from 'react';

import bedPath from '../../images/bed.png'; *// Путь к изображению внутри сборки*

function Bed() {

return (

<img alt='A comfortable bed' src={bedPath}/>

);

}

export default Bed;

Мы импортировали файл изображения bed.png в компонент строчкой кода: import bedPath from '../../images/bed.png'. Как и в предыдущем уроке, использовали относительные пути к файлам. Внутри JSX задали значение {bedPath} атрибуту src тега <img>.

Если кто-то из команды дизайнеров решит изменить дизайн этой иллюстрации, они смогут сохранить другое изображение внутри images/ под именем bed.png, и «Вебпак» будет динамично обновлять сборку по мере необходимости.

Откроем результат в браузере:

Матрас и корона. Так повеселее.

А вот и кроватка! Теперь вы умеете добавлять картинки в компоненты. Если планируете расширять базу мультимедийных файлов приложения, заранее подумайте, как организовать эти файлы. Мы уже делали это [в пятом уроке](https://praktikum.yandex.ru/trainer/web/lesson/a58d8e0d-2ada-4c87-aeb8-d38efddf8321). В следующем уроке научимся настраивать CSS-стили в CRA-проектах.

# Работа с CSS

Лендинг всё ещё выглядит невзрачно, поэтому нужно подумать, как его стилизовать. По умолчанию утилита Create React App предоставляет CSS-файл App.css. Снова посмотрим на структуру директории src/:

Скопировать код

└── src/

├── components/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ └── landing/

│ ├── Bed.js

│ ├── Bed.css

│ ├── Header.js

│ ├── Header.css

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

├── images/

│ └── bed.png

└── <!-- остальные файлы в src/ -->

Сейчас в проекте есть и другие CSS-файлы — мы создали для каждого компонента свой. 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;

}

# Использование CSS в Create React App

Зададим цвет фона начальной страницы и флекс-свойством выровняем её содержимое по центру в файле index.css. Для этого изменим селектор body и добавим свойства:

Скопировать кодCSS

body {

*/\* остальные стандартные свойства Create React App \*/*

background-color: #65539a;

min-height: 100%;

display: flex;

flex-direction: column;

justify-content: center;

align-items: center;

}

Этот стиль применяется ко всему body:

Соседний магазин «Мир колготок» уже завидует

Пока все элементы внутри корневого <div> в файле App.js выровнены по центру страницы. А если позже мы захотим изменить выравнивание отдельных элементов? Чтобы это было возможно, будем использовать отдельные CSS-файлы для стилизации компонентов.

# Импорт CSS

Удалим свойство text-align: center; из файла App.css. После этого заголовок и изображение кровати выровняются по левой стороне экрана. Это выглядит не очень красиво. Проблему можно исправить, применив нужные свойства внутри компонентов.

Начнём с компонента Header: отцентруем текст, изменим размер шрифта и цвет заголовка. Откроем директорию landing/:

Скопировать кодMARKDOWN

└── landing/

├── Bed.js

├── Bed.css

├── Header.js

├── Header.css

└── <!-- остальные файлы -->

У каждого компонента свой CSS-файл — в них будем указывать нужные стили. Сейчас оба файла стилей пустые. Добавим стили в файл Header.css, применяя селектор класса .header-title:

Скопировать кодCSS

.header-title {

text-align: center;

color: #fff;

font-size: 40px;

max-width: 220px;

margin: 0 auto 58px;

text-transform: uppercase;

}

Уже неплохо, но компонент Header пока не подозревает, что он должен применить эти стили. Чтобы сказать ему об этом, импортируем стили в Header.js и правильно назначим класс .header-title внутри JSX:

Скопировать кодJSX

*// компонент Header теперь в Header.js*

import React from 'react';

import './Header.css' *// импортируем CSS-файл*

function Header() {

return (

<h1 className="header-title">Империя матрасов</h1> *// применяем класс*

);

}

export default Header;

Готово.

Чтобы расположить изображение по центру, нужно изменить CSS внутри Bed.css, заменив text-align на свойства display: block; и margin: 0 auto; селектора .bed-logo.

Когда CSS-файл будет готов к работе, нужно его импортировать в Bed.js:

Скопировать кодJSX

*//Bed.js*

import React from 'react';

import bedPath from '../../images/bed.png';

import './Bed.css' *// импортируем CSS-файл*

function Bed() {

return (

*// не забудьте о `className`*

<img className="bed-logo" alt="A comfortable bed" src={bedPath}/>

);

}

export default Bed;

Фух, закулисная работа позади. Посмотрим, как главная страница выглядит теперь:

Маркетологи "Вселенной абажуров" тоже немного напряглись

В целом, кардинальных изменений не произошло, но мы упорядочили код. Вы научились работать с CSS внутри проекта. Осталось подобрать подходящий шрифт для заголовка — займёмся этим в следующем уроке.

# Шрифты

Иногда в проектах требуется подключить особенные шрифты. Например, когда приложение реализовано в фирменном стиле компании. В открытом доступе можно найти множество шрифтов. Для этого удобно использовать библиотеки шрифтов, к примеру, [библиотеку Google Fonts](https://fonts.google.com/).

В этом уроке научимся подключать кастомные шрифты и приукрасим шапку приложения.

Мы подобрали шрифт, который одобрил заказчик приложения «Империя матрасов». Название шрифта — Roboto. Разместим файл шрифта в новой директории fonts/:

Скопировать кодMARKDOWN

└── src/

├── components

├── images

├── fonts/

│ └── Robotocondensed.woff

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

Применим этот шрифт к тегу h1 внутри компонента Header. Пропишем стили в файле Header.css

Скопировать кодCSS

@font-face {

font-family: 'Roboto Condensed';

src: url(../../fonts/Robotocondensed.woff) format('woff');

}

.header-title {

text-align: center;

color: #fff;

font-size: 40px;

font-family: 'Roboto Condensed', sans-serif;

max-width: 220px;

margin: 0 auto 58px;

text-transform: uppercase;

}

Мы задали шрифт Roboto Condensed в свойстве font-family для селектора класса .header-title. Теперь шрифт нужно подключить, как мы это делали в уроке [«Подключение шрифтов к странице»](https://praktikum.yandex.ru/learn/web/courses/dbf98e55-0f76-444b-850c-4538708ad571/sprints/1425/topics/9b7ee0d7-bc33-458c-ae00-b141fe8fac87/lessons/6c5cd533-ad2d-411e-b1f6-44ff164aa22a/).

Поскольку мы используем CRA со всеми преимуществами «Вебпака», внутри директивы @font-face укажем относительный путь к источнику. Это нужно, чтобы при сборке все файлы оказались в правильном месте.

Кроме того, применить шрифт нужно только к шапке, поэтому объявим шрифт через @font-face в файле Header.css. Впрочем, шрифты часто используются в разных местах приложения, поэтому мы также могли бы сделать это в index.css.

Обновим страницу:

Имперские шрифтовые амбиции

Отвлечёмся от шрифтов и подведём итоги пройденных уроков. Вы научились:

1. использовать ключевое слово import, чтобы связать модули внутри проекта CRA;
2. добавлять изображения в проект;
3. связывать CSS-файлами с компонентами;
4. добавлять и использовать шрифты.

Визуальная часть лендинга готова, но пока пользователю совсем нечего делать на нём. В следующем уроке добавим функциональность.

# Добавление классового компонента

Горькая правда лучше, чем сладкая ложь. Придётся признать: «Империи матрасов» пока ещё далеко до запуска. Добавим небольшую форму, в которой пользователь сможет указать свою электронную почту, чтобы узнать о заветном релизе и заказать матрас мечты.

Сперва создадим новый компонент для формы. До сих пор в этом проекте у нас не было необходимости использовать внутреннее состояние. На этот раз оно нам понадобится, чтобы пользователь увидел изменения в интерфейсе после отправки формы. Поэтому применим [классовый компонент](https://praktikum.yandex.ru/trainer/web/lesson/dd5c7aa1-9f36-45c2-98c2-d635130c36de/task/5ac1b4aa-af75-42fb-9266-2624df59ebe8/).

Сначала добавим в папку landing/ два новых файла — InfoForm.js и InfoForm.css:

Скопировать код

└── landing/

├── Bed.js

├── Bed.css

├── Header.js

├── Header.css

├── InfoForm.js

├── InfoForm.css

└── <!-- остальное содержимое директории -->

Следующим шагом настроим компонент InfoForm.js. Так выглядит полный код этого компонента:

Скопировать кодJSX

import React from 'react';

import './InfoForm.css';

class InfoForm extends React.Component {

constructor(props) {

super(props);

this.state = {

submitted: false,

};

}

handleSubmit = () => {

this.setState({ submitted: true });

};

render() {

if (this.state.submitted) {

return (

<p className="infoForm-text">Благодарим вас за подписку. Мы свяжемся с вами, когда царство комфорта

откроется!</p>

);

} else {

return (

<div className="infoForm-container">

<p className="infoForm-text">Заполните форму ниже, чтобы узнать об открытии «Империи матрасов»

первыми.</p>

<form onSubmit={this.handleSubmit} className="infoForm-form">

<input className="infoForm-input" type="email" placeholder={'Введите электронную почту'} />

<button className="infoForm-button" type="submit">Удивите меня</button>

</form>

</div>

);

}

}

}

export default InfoForm;

Теперь разберём его подробнее.

# Компонент в деталях

В самом начале файла мы указываем два импорта: react — для корректной работы JSX-кода и InfoForm.css со стилями формы.

Внутри конструктора мы вызываем super, передавая props, и инициализируем компоненты state так, чтобы this.state.submitted был равен false. Каждый раз, когда пользователь вводит электронную почту в форму и нажимает на кнопку, значение будет изменяться на true.

Далее мы определяем метод handleSubmit. Он будет переключать значение state с false на true при каждом вызове.

Изучим функцию render:

Скопировать кодJSX

render() {

if (this.state.submitted) {

return (

<p className="infoForm-text">

Благодарим вас за подписку. Мы свяжемся с вами, когда царство комфорта

откроется!

</p>

);

} else {

return (

<div className="infoForm-container">

<p className="infoForm-text">Заполните форму ниже, чтобы узнать об открытии «Империи матрасов» первыми.</p>

<form onSubmit={this.handleSubmit} className="infoForm-form">

<input className="infoForm-input" type="email" placeholder={'Введите электронную почту'} />

<button className="infoForm-button" type="submit">Удивите меня</button>

</form>

</div>

);

}

}

Вот что мы делаем в функции render:

1. Используем конструкцию if...else для отрисовки различного JSX-кода в зависимости от значения this.state.submitted. Если значение равно false, выведется элемент <div>, содержащий форму <form>. Если же значение true, отрисуется тег <p> с сообщением-подсказкой о подписке.
2. Привязываем метод onSubmit к <form>. Внутри JSX onSubmit вызывается при каждой отправке формы кликом на кнопку либо нажатием клавиши “Enter”. Для вызова метода значение this.handleSubmit передаётся в фигурных скобках, а значение state обновляется каждый раз, когда пользователь отправляет форму.
3. Задаём значение "email" атрибута type внутри <input>. Это позволяет использовать встроенную валидацию формы: пользователь должен будет ввести в этом поле адрес электронной почты, чтобы кнопка submit стала активной. [Вы научились это делать в шестом спринте](https://praktikum.yandex.ru/trainer/web/lesson/7b9481ef-c340-4060-9cf6-46793bdd0566/task/17ebf6ec-6b37-418e-9cee-af895ca2a2a5/).

В самом конце файла InfoForm.js мы добавили инструкцию export default InfoForm, чтобы экспортировать компонент для последующего использования в файле App.js.

Посмотрим, как теперь выглядит App.js:

Скопировать кодJSX

import React from 'react';

import Header from './landing/Header';

import Bed from './landing/Bed';

import InfoForm from './landing/InfoForm';

import './App.css';

function App() {

return (

<div className="App">

<Header />

<Bed />

<InfoForm />

</div>

);

}

export default App;

Добавился импорт компонента InfoForm, который мы затем включили в корневой элемент App.

Посмотрим, как лендинг выглядит в браузере:

Теперь у нашего приложения есть хоть и маленькая, но настоящая функциональность. На самом деле этого достаточно, чтобы задуматься о дебаггинге приложения — поиске и исправлении ошибок. Ведь начиная с этого момента проект будет становиться сложнее. В следующем уроке мы начнём изучать инструменты и способы отладки приложений на «Реакте».

Пример FoodAdder

import React from 'react';

import './FoodAdder.css';

class FoodAdder extends React.Component {

constructor(props) {

super(props);

this.state = {

FoodList: []

}

}

handleSubmit = () => {

}

render(){

return (

<form className="food-adder">

<input type="text" placeholder="Добавьте продукты" />

<button className="food-adder\_\_submit"></button>

</form>

);

}

}

export default FoodAdder;

import React from 'react';

import './AppContainer.css';

import FoodAdder from './FoodAdder.js';

function AppContainer() {

return (

<div className="container">

<div className="calories">

<h2>Калории:</h2>

</div>

<FoodAdder />

</div>

);

}

export default AppContainer;

# Дебаггинг проекта

Давным-давно, а именно — [в четвёртом спринте](https://praktikum.yandex.ru/trainer/web/lesson/262015a1-7382-440b-af2e-15c816c2b0fc), вы изучили способ отладки кода в веб-браузере. Всё это применимо и к проектам на «Реакте».

В нашем приложении мы ещё не научились обрабатывать данные, полученные от пользователя через форму. Доработаем код, чтобы исправить это. Затем проверим, что всё работает как надо с помощью инструментов, встроенных в браузер.

# Доступ к значениям элементов форм в «Реакте»

Сделаем всё это внутри файла InfoForm.js. Сначала добавим новое значение свойству state, чтобы хранить данные об электронной почте пользователя:

Скопировать кодJSX

this.state = {

submitted: false,

userEmail: '' *// здесь пока пустая строка*

}

Добавим слушатель onChange к элементу <input>. Он будет срабатывать каждый раз, когда в поле ввода вносятся изменения. Например, если пользователь наберёт строку «хочу матрас», onChange сработает 11 раз — по одному на каждый символ в строке.

Затем создадим новый метод handleChange и назначим его в качестве обработчика события onChange. Вот как будет выглядеть элемент input:

Скопировать кодJSX

<input onChange={this.handleChange} className="infoForm-input" type="email" placeholder="Введите электронную почту" />

И метод handleChange:

Скопировать кодJSX

handleChange = (evt) => {

this.setState({ userEmail: evt.target.value });

};

Этот метод будет обновлять this.state.userEmail, чтобы в него попадало самое новое введённое значение. Возможно, сразу не очень понятно, что происходит с переменной evt в теле this.setState. Разобраться в этом вам поможет урок [«Доступ к значениям элементов форм»](https://praktikum.yandex.ru/trainer/web/lesson/092c299b-72eb-4988-9c9b-930b12f15b72/task/953b37dd-7c00-4b52-94d0-97f0e61b7e51/). В этом уроке мы используем значения полей формы, чтобы обновить состояние компонента.

# Базовая отладка в «Реакте»: console.log

Разберём, какие инструменты отладки можно использовать в «Реакте». Добавим вызов console.log в обработчик handleSubmit, чтобы выводить в консоль значение this.state.userEmail каждый раз, когда пользователь отправляет форму:

Скопировать кодJSX

handleSubmit = () => {

console.log(this.state.userEmail);

this.setState({ submitted: true });

}

Знакомый способ! Если открыть приложение и ввести, допустим, “vasyafrontend@yandex.ru” в поле формы, а затем открыть консоль браузера, мы тоже увидим там “vasyafrontend@yandex.ru”.

# Отладка через debugger

Установить в коде контрольные точки можно при помощи уже знакомой вам строки debugger;. Добавим её сразу после вызова console.log внутри функции handleSubmit. Так мы остановим выполнение кода до тех пор, пока не дадим соответствующую команду из браузера:

Скопировать кодJSX

handleSubmit = () => {

console.log(this.state.userEmail);

debugger;

this.setState({ submitted: true });

}

Наше приложение мало что умеет, поэтому эти методы отладки пока не принесут большой пользы, но она станет очевидной с развитием приложения. Для качественной отладки приложений на «Реакте» понадобится более мощный инструмент, который вы изучите позже.

А пока научимся применять основные инструменты разработчика.

# Инструменты разработчика

Инструменты разработчика открываются по-разному в зависимости от операционной системы и браузера. В следующем уроке установим специальное расширение, которое работает внутри Яндекс.Браузера, Chrome, и Firefox, поэтому сосредоточимся на этих браузерах:

* В Яндекс.Браузере или Chrome нажмите сочетание клавиш Option + ⌘ + I (на macOS) либо Shift + Ctrl + I (на Windows или Linux);
* В Firefox нажмите сочетание клавиш Option + ⌘ + I (на macOS) либо Ctrl + Shift + I или F12 (Windows или Linux).

После этого в Chrome вы увидите нечто подобное в верхней части открывшейся панели:

image

Инструменты разработчика в Яндекс.Браузере и Chrome

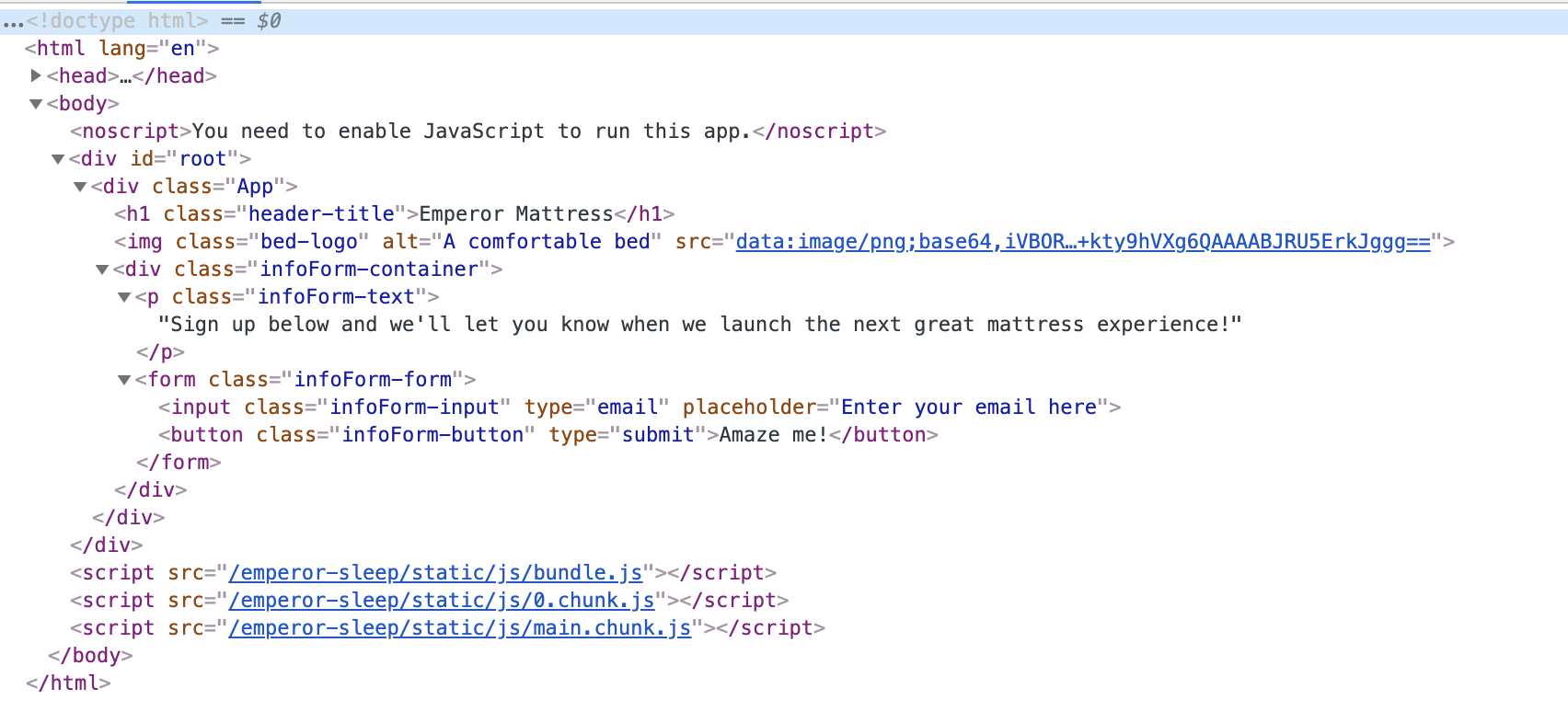
В Firefox аналогичная панель выглядит так:

image

Инструменты разработчика очень похожи в этих двух браузерах. Для проверки DOM используют вкладку “Elements” в Chrome либо вкладку “Inspector” в Firefox. Чтобы открыть консоль в обоих браузерах, нужно просто выбрать вкладку “Console”.

## Изучение DOM

Открыв инструменты разработчика в браузере и проверив DOM нашего проекта, мы увидим такой HTML-код:



DOM страницы в инструментах разработчика

С помощью инструментов разработчика можно получить много полезной информации о проекте. Но для работы с «Реактом» нам нужны специальные инструменты. Рассмотрим их в следующем уроке.

# Установка расширения 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 «Реактом».

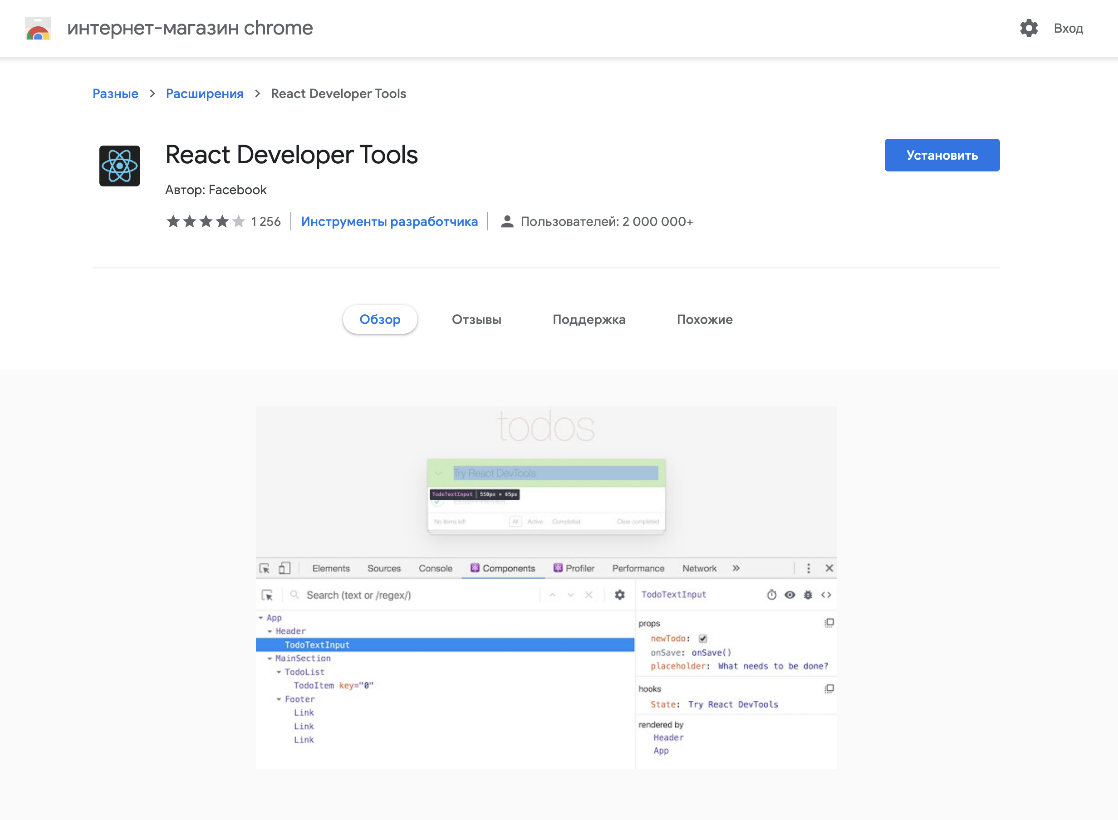
Прежде чем мы научимся пользоваться этим инструментом, нужно его установить. Это несложно, поэтому урок станет для вас передышкой.

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

## Установка в Яндекс.Браузере и Chrome

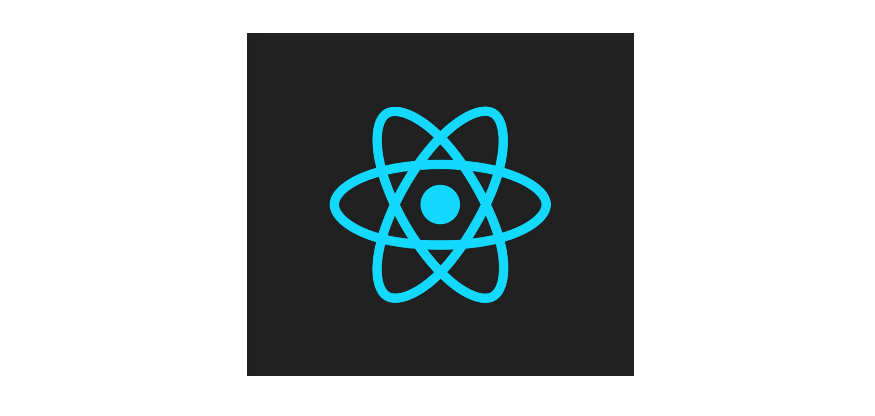
Перейдите [по ссылке в браузере](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=ru).

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:



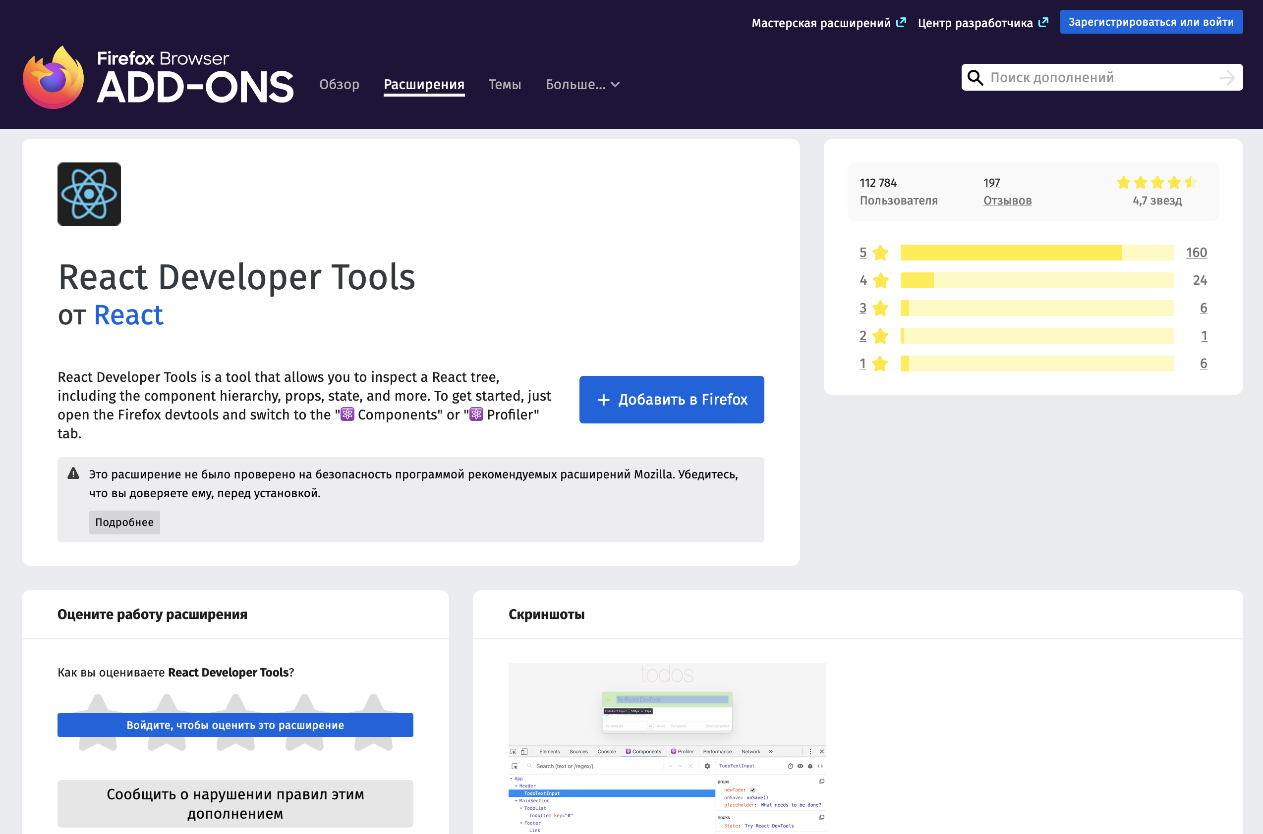
Иконка становится цветной на сайтах c «Реактом»

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/).

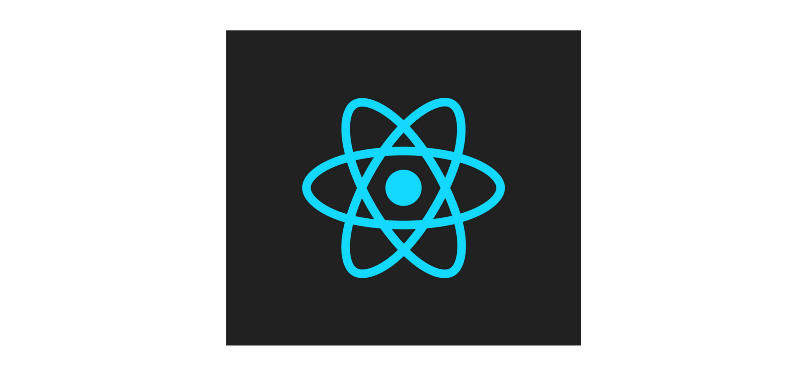
Вы увидите такую страницу:



Нажмите кнопку «Добавить в Firefox».

Появится всплывающее окно, запрашивающее 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:



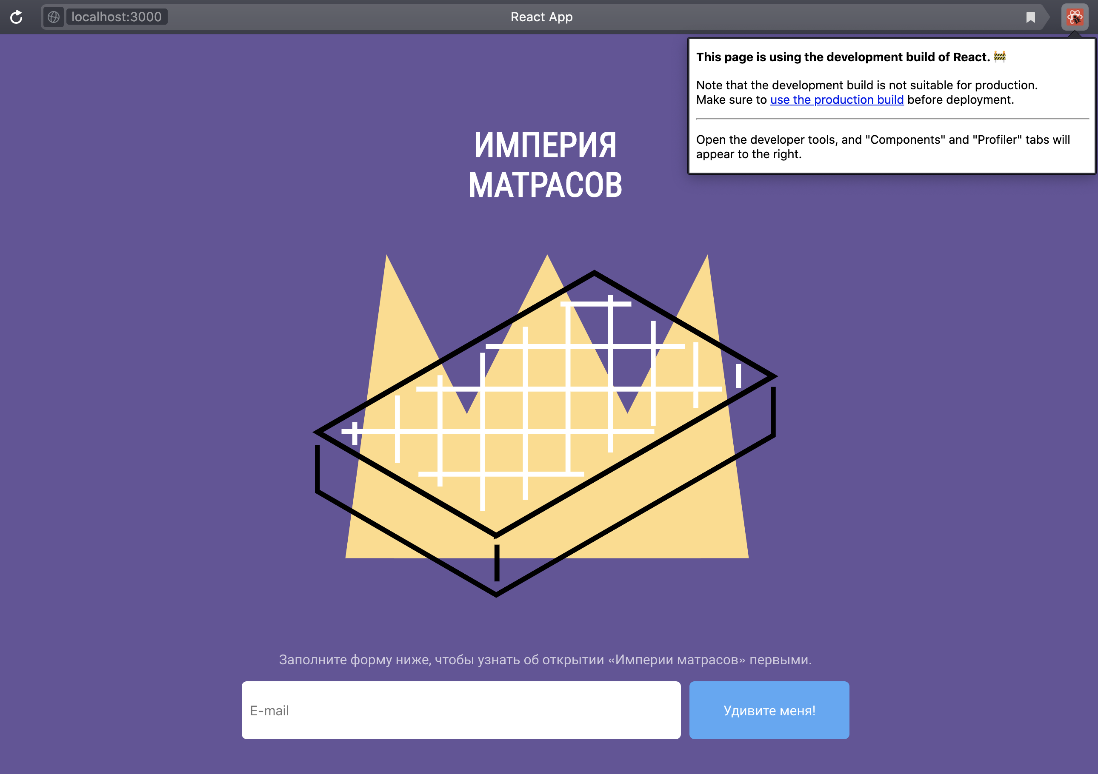
Иконка становится цветной на сайтах c «Реактом»

Если видите эту иконку, ура! Поздравляем. Переходите к следующему уроку.

# Использование расширения React DevTools

Теперь, когда расширение React DevTools установлено, вернёмся к «Империи матрасов». Для этого запустите npm run start и перейдите по адресу <http://localhost:3000/>. Вы увидите, что иконка React DevTools стала красной. Чтобы узнать почему — кликните по ней.

Появится попап React DevTools с сообщением «На этой странице используется сборка „Реакта“ для разработки». Значит, расширение обнаружило проект:



Так и должно быть

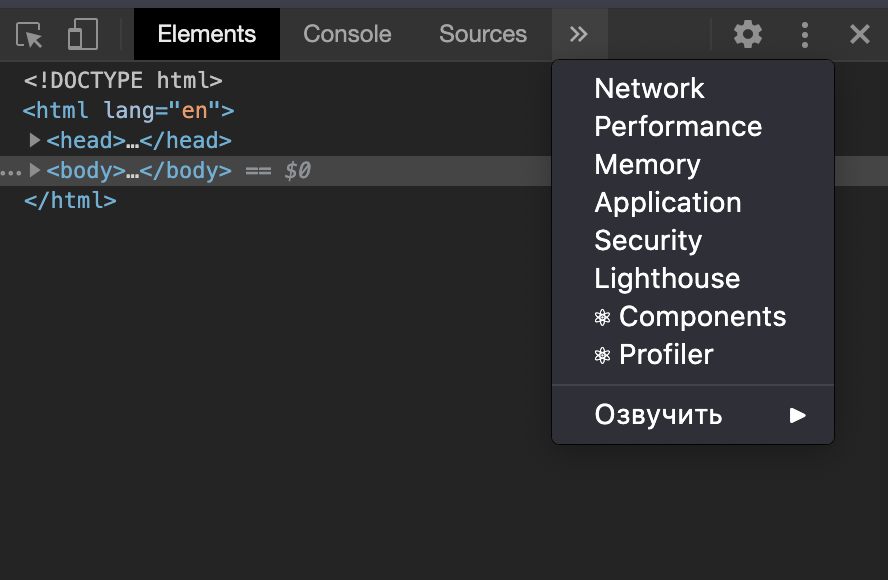
Внизу попапа появится текст: «Откройте инструменты разработчика. Вкладки “Components” и “Profiler” появятся справа».

Поговорим об инструментах разработчика в браузере подробнее.

# Использование React DevTools в браузере

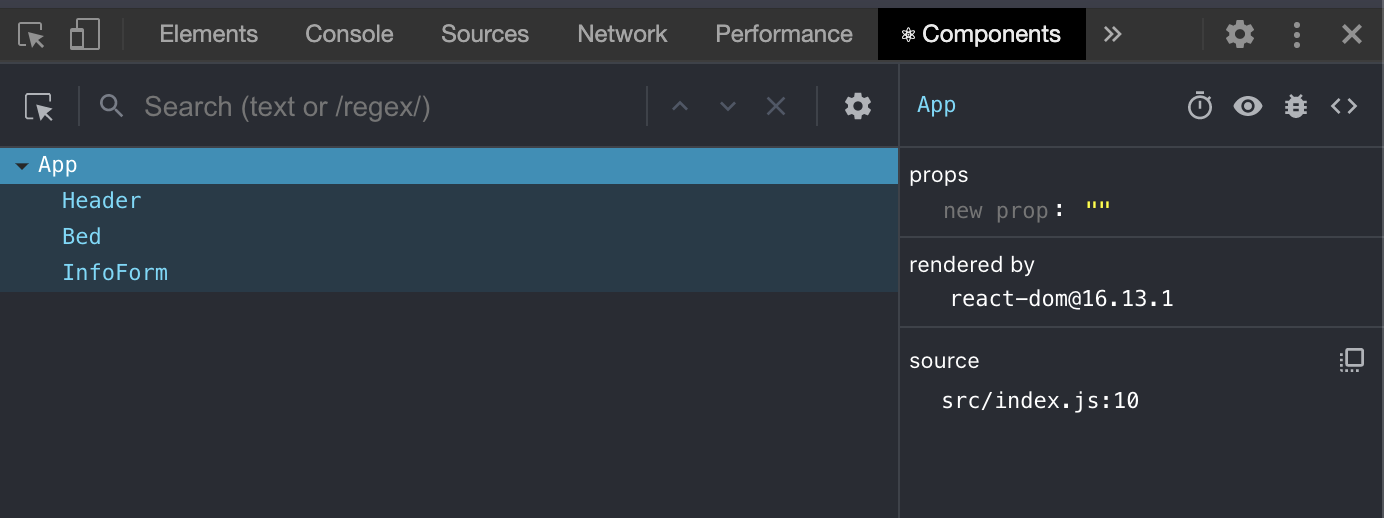
[В 13 уроке](https://praktikum.yandex.ru/trainer/web/lesson/02f9d7c9-f0c8-4434-aadc-a3c18faed060) мы рассказывали, как открыть инструменты разработчика в браузере. В этом уроке откроем те же инструменты разработчика, но с установленным расширением React DevTools.

В инструментах разработчика находится много вкладок. С некоторыми из них вы уже знакомы: например, “Console” и “Elements” или “Inspector” в Chrome. Рассмотрим остальные вкладки. Чтобы увидеть все доступные инструменты, нажмите на символ “»”, расположенный справа от вкладок:



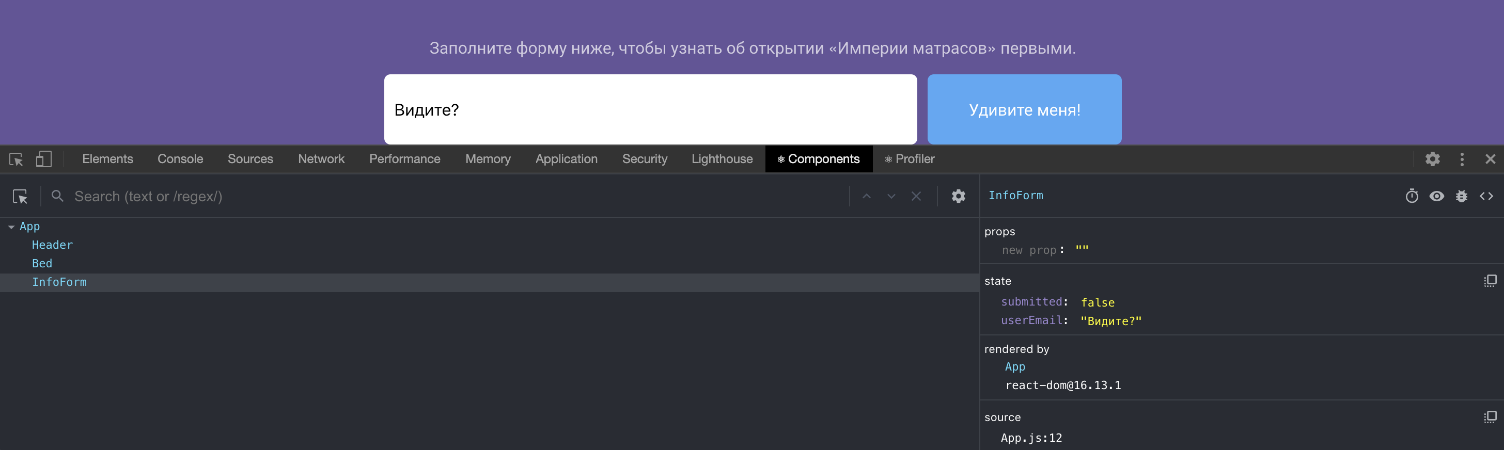
# Вкладка “Components” в React DevTools

После установки расширения React DevTools появятся две вкладки: “Components” и “Profiler”. Сперва рассмотрим вкладку “Components”. Вот что покажет инструмент, когда будет загружено приложение «Империя матрасов»:



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

Подробная информация о компоненте находится внизу вкладки:



Видите?

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 InfoForm.

Особого внимания заслуживают значения стейта компонента, которые обновляются в режиме реального времени. На скриншоте выше введённое в поле для электронной почты «видите?» автоматически появилось в инструментах разработчика.

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.

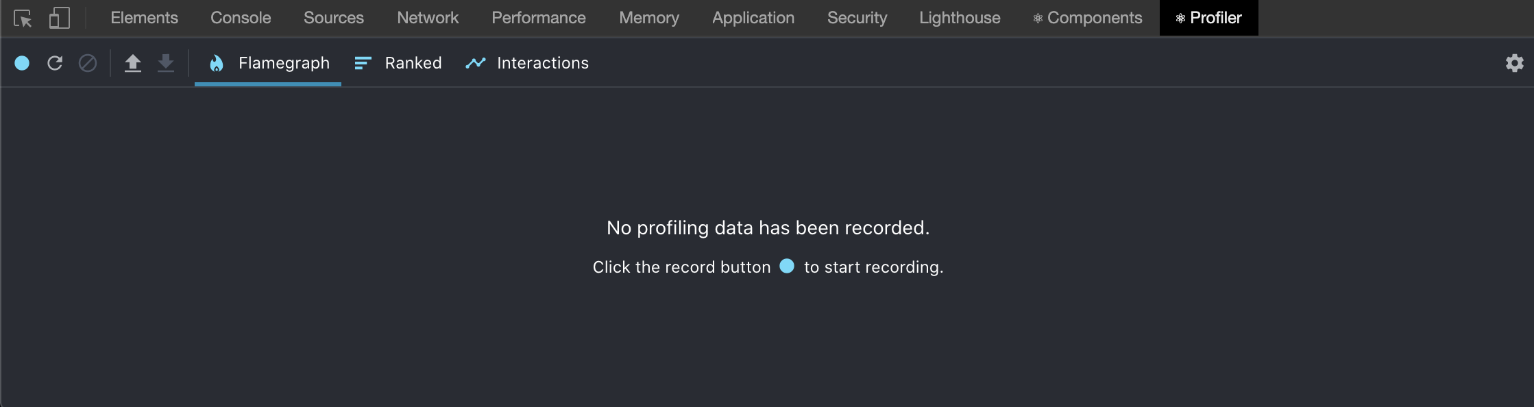
Если использовать эти расширенные возможности вместе с console.log и debugger, можно более точно анализировать изменения состояния и оперативно отслеживать и устранять баги в проектах на «Реакте».

# Вкладка “Profiler” в React DevTools

Поговорим о сравнительно новой функции React DevTools — “Profiler”. Она позволяет детально изучить приложение: отслеживает, как часто оно отрисовывается, и 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.

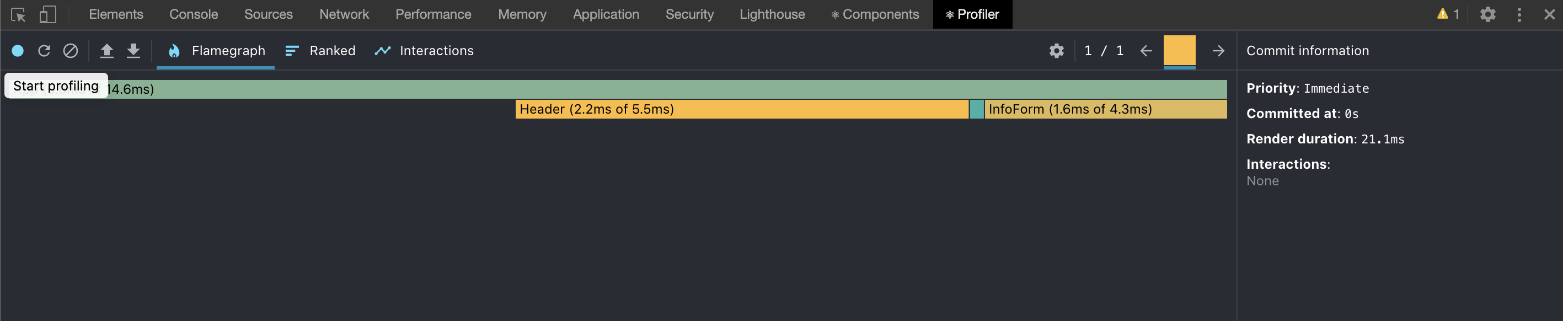
Откроем вкладку “Profiler”:



Чтобы начать собирать данные, нужно нажать на кнопку записи. Сейчас в форме «Империи матрасов» есть только одно поле — для почтового адреса. Когда пользователь отправляет форму, она отрисовывается заново, и пользователь получает сообщение.

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

Вот что мы увидим, если соберём данные приложения «Империя матрасов»:



Мы узнаем, какие компоненты были отрисованы, когда и сколько времени это заняло.

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

**Сборка проекта**

Во всех предыдущих уроках мы разрабатывали проект. Но пора уже его запускать, поэтому научимся собирать проект. Что для этого нужно?

Внутри файла package.json, который был создан с помощью CRA, хранится 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. При сборке проекта нам потребуется запустить скрипт "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 написанного кода, наряду с другими ресурсами: JS, CSS и шрифтами.

Затем эти файлы можно разместить на любом сервере, к которому у нас есть доступ. 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 ранее установили 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:5000. Введите этот адрес в браузере и увидите свой проект!

Увидимся в следующей теме, когда вы будете готовы двигаться дальше.

**Заключение**

Оказалось несложно, правда? В этой теме вы закрепили полученные знания о «Реакте» и узнали, как использовать Create React App для быстрого создания и запуска проекта.

Вы также изучили файловые структуры и узнали, почему они важны в проектах. В этой теме вы разобрали структуру шаблонного CRA проекта, а также научились работать в его экосистеме: импортировать и экспортировать модули, работать с CSS, импортировать изображения и добавлять кастомные шрифты.

Помимо этого, вы научились работать с инструментами для отладки проектов на «Реакте». Установили расширение React DevTools, которое помогает упорядочивать и контролировать процесс создания проектов на «Реакте».

Наконец, вы узнали, как создавать версию проекта для деплоя и запустили её на локальной машине.

В следующей теме вы узнаете о новом синтаксисе хуков, который поможет посмотреть на «Реакт» с другой стороны.