**React: устанавливаем Webpack 5**

В основной программе вы использовали CRA как уже готовую инфраструктуру для проекта. Под капотом CRA находится обычный сконфигурированный Webpack. Но иногда CRA — не лучшее решение. В этом уроке расскажем, как подготовить собственную инфраструктуру React-приложения.

**Инициализация приложения**

Первый шаг — инициализация приложения. Для этого в директории будущего проекта нужно выполнить команду инициализации:

Скопировать кодBASH

npm init

После чего в директории будет создан файл package.json со стандартным набором полей.

**Базовый файл конфигурации Webpack**

Настройка Webpack происходит в файле webpack.config.js. В CRA он спрятан внутри node\_modules/. Мы не используем CRA, поэтому создадим в проекте файл с названием webpack.config.js:

Скопировать кодJSX

const path = require('path');

*// Импортируем пакет path*

module.exports = {

entry: path.resolve(\_\_dirname, './src/index.js'),

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: [ ],

},

],

},

resolve: {

extensions: ['\*', '.js', '.jsx'],

},

output: {

path: path.resolve(\_\_dirname, './dist'),

filename: 'bundle.js',

},

devServer: {

static: path.join(\_\_dirname, './dist'),

compress: true,

port: 3000,

},

};

Конфигурации файла webpack.config.js:

* entry — точка входа для работы Webpack. Для проекта на CRA автоматически создаётся файл index.js, который нельзя удалять. Без этого Webpack не поймёт, где брать JavaScript-содержимое проекта.
* module — набор правил для работы с различными файлами проекта. Здесь мы говорим Webpack о том, как работать с кодом, стилями, изображениями, шрифтами и т.д.:
  + rules — конкретное правило для определённых типов файлов. В конфигурации выше мы указываем все js- и jsx-файлы таким правилом: test: /\.(js|jsx)$/.
  + resolve — способ работы с модулями. В нашем случае поле extensions отвечает за порядок обработки модулей.
* output — настройка скомпилированного проекта:
  + path — путь и название директории, где расположится скомпилированный проект.
  + filename — название бандла. Бандл — скомпилированный JavaScript.
* devServer — локальный сервер для разработки:
  + static — директория, из которой код «подтягивается» на локальный сервер.

Для начала работы этого будет достаточно. Сам Webpack мы установим чуть позже, а пока разберёмся с транспиляцией кода.

**React с Babel**

В качестве инфраструктуры для React-приложения мы будем использовать сконфигурированный Webpack, а для транспиляции кода в обычный JavaScript — Babel. Webpack позволяет собрать все компоненты проекта в один большой JavaScript-файл, а Babel пытается «говорить» с браузером «на одном языке». Babel преобразует новые фичи последних JavaScript спецификаций в код, понятный всем браузерам. Он нужен и самому React: по умолчанию JSX-синтаксис и .jsx файлы не поддерживаются браузерами, Babel же транспилирует весь JSX-код в обычный JavaScript.

Установим в командной строке Babel, [пресет Babel для React](https://babeljs.io/docs/en/babel-preset-react) и загрузчик, необходимый для интеграции Babel с Webpack:

Скопировать кодJSX

npm install --save-dev @babel/core @babel/preset-env @babel/preset-react babel-loader

Создайте файл .babelrc. Как и Webpack, это файл конфигурации, но уже для Babel. После этого в конфигурацию нужно добавить пресет — способ транспиляции JavaScript-кода:

Скопировать кодJSX

{

"presets": [

"@babel/preset-env",

"@babel/preset-react"

]

}

Когда Babel сконфигурирован, его нужно связать с конфигом Webpack, иначе ничего не заработает. Связать Babel с Webpack достаточно просто: в правиле работы с js-файлами нужно указать babel-loader в массиве use:

Скопировать кодJSX

const path = require('path');

module.exports = {

entry: path.resolve(\_\_dirname, './src/index.js'),

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

*// Сообщаем вебпаку, что для работы с js-, jsx-файлами*

*// следует использовать babel-loader*

use: ['babel-loader'],

},

],

},

resolve: {

extensions: ['\*', '.js', '.jsx'],

},

output: {

path: path.resolve(\_\_dirname, './dist'),

filename: 'bundle.js',

},

devServer: {

static: path.join(\_\_dirname, './dist'),

compress: true,

port: 3000,

},

};

Это всё, что нужно сделать, чтобы транспилировать React в обычный JavaScript с помощью Babel и Webpack. Теперь мы можем писать React с JSX.

**React с Webpack**

Пока структура папок для приложения, которое использует Webpack и Babel, выглядит так:

Скопировать кодBASH

- node\_modules/

- dist/

-- index.html

- src/

-- index.js

- package.json

- webpack.config.js

Теперь самостоятельно, без помощи CRA, установим в проект сам React. Поставим пакеты [react](https://www.npmjs.com/package/react) и [react-dom](https://www.npmjs.com/package/react-dom). Установите их в командной строке:

Скопировать кодJSX

npm install --save react react-dom

Когда CRA создаёт инфраструктуру проекта, то в src/index.js уже есть заготовка кода. Мы напишем код, аналогичный заготовке из CRA:

Скопировать кодJSX

import React from 'react';

import ReactDOM from 'react-dom';

const title = 'React с Webpack и Babel';

ReactDOM.render(

<h1>{title}</h1>,

document.getElementById('app')

);

Как вы уже знаете, ReactDOM нужен второй параметр — HTML-элемент, внутри которого будет складываться вся разметка. Поэтому в index.html в директории dist/ добавим div с id:

Скопировать кодJSX

<!DOCTYPE html>

<html>

  <head>

    <title>Hello React</title>

  </head>

  <body>

<div id="app"></div>

    <script src="./bundle.js"></script>

  </body>

</html>

Перед первым запуском приложения пора установить и сам Webpack. Добавим все необходимые пакеты:

Скопировать кодBASH

npm install --save-dev webpack webpack-cli webpack-dev-server

Эта команда установит сам webpack, а также консольную утилиту для работы с webpack (webpack-cli) и пакет для запуска локального сервера.

Теперь подключим Webpack к webpack.config.js:

Скопировать кодJSX

const path = require('path');

const webpack = require('webpack');

*// Подключим сам webpack*

module.exports = {

entry: path.resolve(\_\_dirname, './src/index.js'),

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: ['babel-loader'],

},

],

},

resolve: {

extensions: ['\*', '.js', '.jsx'],

},

output: {

path: path.resolve(\_\_dirname, './dist'),

filename: 'bundle.js',

},

devServer: {

static: path.join(\_\_dirname, './dist'),

compress: true,

port: 3000,

},

};

**Настройка окружения**

В CRA для запуска приложения в dev режиме используется команда start. Чтобы добавить похожий функционал в свой проект, необходимо создать новый скрипт в файле package.json:

Скопировать кодJSON

{

"name": "react-webpack-project",

"version": "1.0.0",

"license": "ISC",

"private": true,

"scripts": {

// скрипт start запускает webpack

"start": "webpack serve --mode=development"

},

"devDependencies": {

"@babel/core": "^7.14.3",

"@babel/preset-env": "^7.14.4",

"@babel/preset-react": "^7.13.13",

"babel-loader": "^8.2.2",

"webpack": "^5.38.1",

"webpack-cli": "^4.7.0"

},

"dependencies": {

"react": "^17.0.2",

"react-dom": "^17.0.2"

}

}

Теперь попробуем запустить приложение с помощью npm start. Эта команда инициирует процесс сборки приложения, а также запустит локальный сервер, благодаря которому страница будет доступна в браузере.

Если перейти в браузере по адресу, который вы указали в конфигурации — в примере это localhost:3000 — на странице отобразится заголовок «React с Webpack и Babel». Это значит, что всё работает.

Но если поменять в коде title на «React с Babel и Webpack», то на странице ничего не изменится. Обновлять страницу каждый раз неудобно, поэтому настроим автоматическое обновление при внесении изменений.

**Hot module replacement в React**

React-hot-loader (он же Hot Module Replacement) — функциональность быстрого внесения изменений на локальный сервер. Hot Module Replacement отображает изменения, внесённые в код, без перезагрузки страницы.

Установите Hot Module Replacement в командной строке:

Скопировать кодJSX

npm install --save-dev react-hot-loader

Затем добавьте к файлу webpack.config.js такую конфигурацию:

Скопировать кодJSX

const webpack = require('webpack');

const path = require('path');

module.exports = {

entry: path.resolve(\_\_dirname, './src/index.js'),

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: ['babel-loader'],

},

],

},

resolve: {

extensions: ['\*', '.js', '.jsx'],

},

output: {

path: path.resolve(\_\_dirname, './dist'),

filename: 'bundle.js',

},

plugins: [

*// добавим плагин для hmr*

new webpack.HotModuleReplacementPlugin()

],

devServer: {

static: path.join(\_\_dirname, './dist'),

compress: true,

port: 3000,

*// сообщим dev-серверу, что в проекте используется hmr*

hot: true

},

};

В последних версиях Webpack нет необходимости дополнительно устанавливать React-hot-loader. Достаточно указать флаг hot равный true.

В файле src/index.js нужно воспользоваться самим плагином:

Скопировать кодJSX

import React from 'react';

import ReactDOM from 'react-dom';

const title = 'React with Webpack and Babel';

ReactDOM.render(

<div>{title}</div>,

document.getElementById('app')

);

module.hot.accept();

Теперь можно перезапустить приложение. Попробуем внести изменения в index.js: обновления сразу отобразятся на странице браузера. Если удалить строку с module.hot.accept ();, то при внесении изменений в код браузер будет перезагружаться.

**Тестирование конфигурации в работе**

Добавим в приложение несколько компонентов и проверим, скомпилируются ли все модули в готовый бандл. В файл src/index.js импортируем компонент App:

Скопировать кодJSX

import React from 'react';

import ReactDOM from 'react-dom';

import App from './app';

const title = 'React with Webpack and Babel';

ReactDOM.render(

<App title={title} />,

document.getElementById('app')

);

module.hot.accept();

Затем создадим в папке src/ новый файл:

Скопировать кодBASH

cd src/

touch app.js

И добавим в него функциональный компонент:

Скопировать кодJSX

import React from 'react';

const App = ({ title }) =>

<h1>{title}</h1>;

export default App;

Круто, первый функциональный компонент готов! А мы сделали минимально необходимую настройку Webpack. Можем двигаться дальше.

# Webpack 5: Как пользоваться CSS-модулями на React

CSS-модули — один из самых популярных способов стилизации React компонентов. Использование CSS-модулей гарантирует отсутствие коллизий с другими стилями, так как на этапе компиляции к классам добавляются уникальные идентификаторы. Для CSS-модулей неважно, используете ли вы только CSS или более продвинутый препроцессор (например, SASS): вы всё равно можете записать эти стили в файлы и положить их рядом с React-компонентами.

Прежде чем использовать CSS-модули в React, нужно немного настроить приложение. Для начала вам понадобятся ещё несколько загрузчиков для Webpack. Они позволят Webpack в том числе и объединять CSS-файлы:

Скопировать код

npm install css-loader style-loader --save-dev

Добавьте в файл webpack.config.js новые загрузчики для работы с CSS-файлами:

Скопировать кодJAVASCRIPT

module.exports = {

...

module: {

rules: [

...

{

test: /\.css$/i,

exclude: /node\_modules/,

use: [

'style-loader',

{

loader: 'css-loader',

options: {

modules: true,

},

},

],

},

],

},

...

};

Вот и всё, что нужно для настройки CSS-модулей в Webpack. Теперь перейдём к первому CSS-файлу. Назовём его src/style.css:

Скопировать кодCSS

.title {

color: red;

}

После этого вы можете импортировать CSS-файл в один из React-компонентов. А затем — использовать CSS-класс, определённый в CSS-файле вашего React-компонента. Для этого импортируйте его и примените определённый CSS-класс как свойство className React-компонента.

Скопировать кодJSX

import React from 'react';

import styles from './style.css';

export const DangerText = ({ text }) => (

<p className={styles.title}>{text}</p>

);

Теперь можно добавить больше CSS-файлов рядом с React-компонентами. Обычно для каждого React-компонента используется один файл со стилем. В этом CSS-файле вы можете добавлять столько CSS-классов, сколько нужно для стилизации React-компонента. Для этого просто импортируйте стили из CSS-файла и используйте их в компоненте React, как в примере выше.

# Webpack 5: Использование изображений в React

В этом уроке вы узнаете, как настроить Webpack для использования изображений в приложении на React.

Для начала поместите файлы изображений в одну папку. Например, в src/ может быть папка assets/, в которой есть images/:

Скопировать код

- src/

--- assets/

----- images/

------- dog.jpg

Затем установите один из популярных загрузчиков Webpack, который повсеместно используется разработчиками для обработки изображений:

Скопировать кодBASH

npm install url-loader --save-dev

Последним шагом подключите новый загрузчик в конфигурацию Webpack:

Скопировать кодJAVASCRIPT

module.exports = {

...

module: {

rules: [

...

{

test: /\.(jpg|png)$/,

use: {

loader: 'url-loader',

},

},

],

},

...

};

Этот загрузчик связывает расширения файлов изображений jpg и png с приложением. В случае, если вы используете другие расширения файлов для изображений, укажите их через вертикальную черту | по аналогии с примером. Также url-loader поддерживает дополнительные опции, о которых можно прочитать [в официальной документации](https://webpack.js.org/loaders/url-loader/).

Теперь вы знаете, как использовать изображения в React-приложении с Webpack 5. Например, в React вы можете включить изображение с помощью HTML-элемента img и его атрибута src:

Скопировать кодJSX

import React from 'react';

import MyImage from './assets/images/myimage.jpg';

const App = ({ title }) => (

<div>

<span>{title}</span>

<img src={MyImage} alt="torchlight in the sky" />

</div>

);

export default App;

# Webpack 5: Использование шрифтов в React

Подключение шрифтов при использовании обычного Webpack в React очень похоже на то, как это делается в CRA. Единственное отличие в том, что шрифты — новый формат файлов. А значит, для них нужно создать новое правило в webpack.config.js. Другими словами, придётся чуть-чуть поработать с ключом modules в конфигурации.

## Обработка файлов шрифтов с помощью Webpack

Добавим несколько файлов шрифтов в директорию fonts/. Выберем шрифт OpenSans (он нам очень понравился):

Скопировать код

- src/

--- assets/

----- fonts/

------- OpenSans-Bold.woff

------- OpenSans-Bold.woff2

------- OpenSans-Regular.woff

------- OpenSans-Regular.woff2

Для работы с файлами шрифтов воспользуемся [Asset Modules](https://webpack.js.org/guides/asset-modules/) — специальным типом модулей, которые позволяют добавлять в сборку ресурсы, не прибегая к помощи отдельных загрузчиков.

Этот функционал уже встроен в Webpack и нам достаточно лишь научить его фидеть файлы шрифтов. Для этого расширим массив rules. Добавим в него правило работы с файлами форматов woff и woff2:

Скопировать кодJAVASCRIPT

module.exports = {

*// ...*

module: {

rules: [

*// ...*

{

test: /\.(woff|woff2)$/,

type: 'asset/resource',

},

],

},

*// ...*

};

Теперь файлы шрифтов будут обрабатываться Webpack. Нам остаётся только подключить их в проекте. Используем директиву @font-face и подключим наш любимый шрифт OpenSans:

Скопировать кодCSS

@font-face {

font-family: 'OpenSans';

font-style: normal;

font-weight: 700;

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

}

html,

body {

font-family: 'OpenSans', sans-serif;

}

OpenSans появился в проекте, три строчки в конфигурации Webpack добавлены. Двигаемся дальше.

**Как использовать ESLint в Webpack 5: руководство по установке**

В этом уроке мы познакомимся с ESLint — инструментом для проверки и унификации стиля кода на любом выбранном стандарте JavaScript. ESLint приводит код к единому стилю, помогает автоматически исправлять некоторые ошибки и интегрируется со многими инструментами разработки.

Если вы работаете в команде, важно выработать общий стиль написания кода, и следовать правилам, чтобы код выглядел одинаково. Такое единообразие помогает другим разработчикам лучше понимать ваш код и избегать ошибок.

**ESLint**

Начнем с установки [библиотеки eslint](https://github.com/eslint/eslint). Вы можете установить её в проект с помощью npm install eslint из корневого каталога проекта.

Вы также можете установить расширение/плагин ESLint для вашего редактора/IDE. Например, расширение ESLint для VSCode можно найти [по ссылке](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint). После этого вы должны увидеть все ошибки, которые ESLint найдёт в редактора/IDE.

**ESLint + Webpack + Babel**

Так как в проект установлен Webpack, ему нужно сообщить, что мы хотим использовать ESLint в процессе сборки. Для этого установите [eslint-loader](https://github.com/webpack-contrib/eslint-loader) в командной строке для зависимостей вашего проекта из корневой папки проекта:

Скопировать кодBASH

npm install --save-dev eslint-loader

Затем используйте загрузчик ESLint в файле настроек Webpack webpack.config.js:

Скопировать кодJAVASCRIPT

...

module: {

rules: [

{

test: /\.(js)$/,

exclude: /node\_modules/,

use: ["babel-loader", "eslint-loader"]

}

]

},

...

Теперь весь исходный код, который проходит через Weback, будет автоматически проверяться ESLint. Как только вы запустите приложение, оно выдаст ошибку: "No ESLint configuration found". Добавим файл конфигурации в корневом каталоге проекта в командной строке:

Скопировать кодBASH

touch .eslintrc

Затем в новом файле .eslintrc создадим пустой набор правил ESLint:

Скопировать код

{

"rules": {}

}

Позже вы можете указать в этом файле конкретные правила. Но сначала снова запустим приложение. Вы можете (снова) столкнуться с ошибками синтаксического анализа, например: "The keyword 'import' is reserved" или "The keyword 'export' is reserved". Они возникают из-за того, что ESLint ещё не знает о функциях JavaScript, включённых в Babel. Например, операторы импорта или экспорта — функции JavaScript ES6. Для линтинга исходного кода используйте пакет [babel-eslint](https://github.com/babel/babel-eslint). С его помощью линтинг применится ко всему, что трансформирует Babel. Для этого в корневом каталоге проекта введём:

Скопировать кодBASH

npm install --save-dev babel-eslint

Затем в файле конфигурации .eslintrc добавим в качестве парсера babel-eslint:

Скопировать код

{

"parser": "babel-eslint",

"rules": {}

}

Важно: если предыдущая ошибка с функциями JavaScript, включёнными в Babel, всё ещё отображается в IDE/редакторе (вы могли установить плагин/расширение ESLint), перезапустите IDE/редактор и проверьте, появится ли ошибка снова. На этот раз её быть не должно.

Теперь вы сможете запустить приложение без ошибок ESLint. Ошибки не отобразятся, потому что пока вы не указали правила.

**Правила ESLint**

Правила ESLint используют для стандартизации стиля кода. Мы предлагаем самостоятельно ознакомиться [со списком правил](https://eslint.org/docs/rules/). А пока добавим первое правило в файл конфигурации .eslintrc:

Скопировать кодJAVASCRIPT

{

...

"rules": {

"max-len": [1, 70, 4, {ignoreComments: true}]

}

...

}

Это правило проверяет длину символов в строке кода. У него несколько аргументов:

1. Тип ошибки:
   * "off", или 0, — turn the rule off,
   * "warn", или 1, — turn the rule on as a warning (doesn't affect exit code),
   * "error", или 2, — turn the rule on as an error (exit code will be 1).
2. Максимальная длина строки.
3. Табуляция: определение ширины для символов табуляции.
4. Дополнительные параметры.

В нашем примере, если строка превышает 70 символов, вы получите предупреждение при запуске приложения с помощью npm start. Попробуйте вызвать это предупреждение: напишите строку кода длиной более 70 символов. ESLint должен ответить вам что-то вроде: "Line 5 exceeds the maximum line length of 70". Вы можете настроить правило, чтобы разрешить более длинные строки:

Скопировать кодJAVASCRIPT

{

...

"rules": {

"max-len": [1, 120, 4, {ignoreComments: true}]

}

...

}

Если вы всё ещё видите предупреждение, это ваш шанс поработать над стилем кода.

**Готовая конфигурация ESLint от Airbnb**

Согласитесь, было бы удобно придумать набор правил ESLint и использовать их для каждого проекта. Существуют различные общие конфигурации ESLint и одна из самых популярных — [конфигурация Airbnb ESLint](https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb), основанная [на руководстве по стилю Airbnb](https://github.com/airbnb/javascript). Вы можете установить конфигурацию с помощью следующей команды в командной строке из корневого каталога проекта:

Скопировать кодBASH

npx install --dev eslint-config-airbnb

После этого вы можете ввести конфиг в файл конфигурации .eslintrc для ESLint:

Скопировать кодJAVASCRIPT

{

"parser": "babel-eslint",

"extends": ["airbnb"],

"rules": {

"max-len": [1, 70, 2, { "ignoreComments": true }]

}

}

Примечание: вы можете сохранить собственные правила ESLint (например max-len из предыдущего примера), чтобы расширить набор правил ESLint от Airbnb. Но мы бы не рекомендовали создавать собственные правила ESLint. Лучше выбрать одну из наиболее популярных конфигураций ESLint среди крупных компаний и следовать ей. Если вы уже продвинутый пользователь JavaScript, то можете добавлять свои правила к ESLint, расширив их или самостоятельно придумав новую конфигурацию.

После повторного запуска приложения в командной строке или проверки вывода в IDE/редакторе с установленным плагином/расширением ESLint вы можете увидеть новые всплывающие предупреждения или ошибки ESLint. Самое время, чтобы их исправить.

**Как отключить правила ESLint**

Иногда вы можете увидеть множество нарушений правил ESLint в командной строке или IDE/редакторе. Вам нужно решить следовать ли общепринятым рекомендациям. Но если вы не уверены в предупреждении ESLint, поищите правило в документации и оцените, хотите ли его использовать. Вы можете либо исправить предупреждение в указанном файле исходного кода, либо полностью удалить или отключить правило, если считаете, что оно вам не нужно.

Если вы хотите глобально удалить правило ESLint, просто удалите его из файла .eslintrc (если вы указали его сами, а не взяли из популярного руководства по стилю вроде Airbnb). Если правило добавлено из популярного руководства, вы можете только отключить его. Например, правило ESLint no-unused-vars из конфигурации Airbnb ESLint можно отключить так:

Скопировать кодJAVASCRIPT

{

"parser": "babel-eslint",

"extends": ["airbnb"],

"rules": {

"no-unused-vars": 0

}

}

Также вы можете отключить собственные или расширенные правила ESLint в соответствующем файле исходного кода:

Скопировать кодJAVASCRIPT

*/\* eslint-disable no-unused-vars \*/*

const myUnusedVariable = 42;

*/\* eslint-enable no-unused-vars \*/*

Кроме этого, вы можете отключить правило ESLint во всём файле или его оставшейся части, просто не включая правило ESLint снова:

Скопировать кодJAVASCRIPT

*/\* eslint-disable no-unused-vars \*/*

const myUnusedVariable = 42;

Теперь вы знаете всё о ESLint и можете стилизовать код с применением популярной конфигурации ESLint (например Airbnb). Также вы разобрались, как добавлять собственные правила, отображать нарушения в IDE/редакторе/командной строке, исправлять нарушения, удалять или отключать правила ESLint.

# Webpack 5: Как использовать ESLint в React

В этом уроке разберём, как настраивать ESLint для React. Мы уже настроили базовую работу линтера и расширили правила работы с JS и JSX-кодом в webpack.config.js:

Скопировать кодJAVASCRIPT

*//...*

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: ["babel-loader", "eslint-loader"]

}

]

},

*//...*

Но это ещё не всё. В первую очередь ESLint — набор правил. Для JSX существует свой набор правил и обычных возможностей линтера будет недостаточно. Существует множество расширений для линтинга. Мы воспользуемся одним из самых популярных — стайл-гайдом airbnb.

Все расширения конфигураций ESLint устанавливаются как npm-пакеты. Установим конфигурацию airbnb для ESLint:

Скопировать кодBASH

npx install-peerdeps --dev eslint-config-airbnb

После установки пакета нужно расширить конфигурацию линтера. Это делается в файле .eslintrc. В ключе extends указывается массив, элементом которого сделаем airbnb:

Скопировать кодJAVASCRIPT

{

"parser": "babel-eslint",

"extends": [ "airbnb" ]

}

После этого при запуске приложения и каждой повторной компиляции в консоли будут появляться сообщения о нарушениях и предупреждениях стайл-гайда airbnb. Ошибки, как и везде, важно исправлять.

Также можно добавить к вашему редактору кода плагин ESLint. Это позволит не дожидаться компиляции кода: плагин будет показывать некорректные участки кода ещё до сохранения.

Проект собран, все дополнительные файлы подключены. А код будет сразу проверяться на наличие ошибок. Вы — великолепны! Самое время переходить к следующему спринту.