

## d Webpack

In the early days, React was somewhat famous for being very difficult to configure the tools required for application development. To make the situation easier, [Create React App](#) was developed, which eliminated configuration-related problems. [Vite](#), which is also used in the course, has recently replaced Create React App in new applications.

Both Vite and Create React App use *bundlers* to do the actual work. We will now familiarize ourselves with the bundler called [Webpack](#) used by Create React App. Webpack was by far the most popular bundler for years. Recently, however, there have been several new generation bundlers such as [esbuild](#) used by Vite, which are significantly faster and easier to use than Webpack. However, e.g. esbuild still lacks some useful features (such as hot reload of the code in the browser), so next we will get to know the old ruler of bundlers, Webpack.

### Bundling

We have implemented our applications by dividing our code into separate modules that have been *imported* to places that require them. Even though ES6 modules are defined in the ECMAScript standard, the older browsers do not know how to handle code that is divided into modules.

For this reason, code that is divided into modules must be *bundled* for browsers, meaning that all of the source code files are transformed into a single file that contains all of the application code. When we deployed our React frontend to production in [part 3](#), we performed the bundling of our application with the `npm run build` command. Under the hood, the npm script bundles the source, and this produces the following collection of files in the `dist` directory:

```
├── assets
│   ├── index-d526a0c5.css
│   ├── index-e92ae01e.js
│   └── react-35ef61ed.svg
├── index.html
└── vite.svg
```

copy

The `index.html` file located at the root of the `dist` directory is the "main file" of the application which loads the bundled JavaScript file with a *script* tag:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
    <script type="module" crossorigin src="/assets/index-e92ae01e.js"></script>
    <link rel="stylesheet" href="/assets/index-d526a0c5.css">
  </head>
  <body>
    <div id="root"></div>

  </body>
</html>
```

copy

As we can see from the example application that was created with Vite, the build script also bundles the application's CSS files into a single `/assets/index-d526a0c5.css` file.

In practice, bundling is done so that we define an entry point for the application, which typically is the `index.js` file. When webpack bundles the code, it includes not only the code from the entry point but also the code that is imported by the entry point, as well as the code imported by its import statements, and so on.

Since part of the imported files are packages like React, Redux, and Axios, the bundled JavaScript file will also contain the contents of each of these libraries.

The old way of dividing the application's code into multiple files was based on the fact that the `index.html` file loaded all of the separate JavaScript files of the application with the help of script tags. This resulted in decreased performance, since the loading of each separate file results in some overhead. For this reason, these days the preferred method is to bundle the code into a single file.



Next, we will create a webpack configuration by hand, suitable for a new React application.

Let's create a new directory for the project with the following subdirectories (`build` and `src`) and files:

```
├─ build
├─ package.json
├─ src
├─ index.js
└─ webpack.config.js
```

copy

The contents of the *package.json* file can e.g. be the following:

```
{
  "name": "webpack-part7",
  "version": "0.0.1",
  "description": "practising webpack",
  "scripts": {},
  "license": "MIT"
}
```

copy

Let's install webpack with the command:

```
npm install --save-dev webpack webpack-cli
```

copy

We define the functionality of webpack in the *webpack.config.js* file, which we initialize with the following content:

```
const path = require('path')

const config = () => {
  return {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'main.js'
    }
  }
}

module.exports = config
```

copy

**Note:** it would be possible to make the definition directly as an object instead of a function:

```
const path = require('path')

const config = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'main.js'
  }
}

module.exports = config
```

copy

An object will suffice in many situations, but we will later need certain features that require the definition to be done as a function.

We will then define a new npm script called *build* that will execute the bundling with webpack:

```
// ...
"scripts": {
  "build": "webpack --mode=development"
},
// ...
```

copy

Let's add some more code to the *src/index.js* file:

```
const hello = name => {
  console.log(`hello ${name}`)
}
```

copy



When we execute the `npm run build` command, our application code will be bundled by webpack. The operation will produce a new *main.js* file that is added under the *build* directory:

```

→ webpack npm run build

> webpack-osa7@0.0.1 build
> webpack --mode=development

asset main.js 1.24 KiB [compared for emit] (name: main)
./src/index.js 56 bytes [built] [code generated]
webpack 5.68.0 compiled successfully in 110 ms
→ webpack cat build/main.js
/*
 * ATTENTION: The "eval" devtool has been used (maybe by default in mode: "development").
 * This devtool is neither made for production nor for readable output files.
 * It uses "eval()" calls to create a separate source file in the browser devtool.
 * If you are trying to read the output file, select a different devtool (https://webpack.js.org/configuration/devtool/)
 * or disable the default devtool with "devtool: false".
 * If you are looking for production-ready output files, see mode: "production" (https://webpack.js.org/configuration/mode/).
 */
/******/ (function() { // webpackBootstrap

```

The file contains a lot of stuff that looks quite interesting. We can also see the code we wrote earlier at the end of the file:

```
eval("const hello = name => {\n  console.log(`hello ${name}`)\n}\n\n// sourceURL=webpack://webpack-osa7/./src/index.js?");
```

copy

Let's add an *App.js* file under the *src* directory with the following content:

```

const App = () => {
  return null
}

export default App

```

copy

Let's import and use the *App* module in the *index.js* file:

```

import App from './App';

const hello = name => {
  console.log(`hello ${name}`)
}

App()

```

copy

When we bundle the application again with the `npm run build` command, we notice that webpack has acknowledged both files:

```

→ webpack npm run build

> webpack-osa7@0.0.1 build
> webpack --mode=development

asset main.js 4.21 KiB [emitted] (name: main)
runtime modules 670 bytes 3 modules
cacheable modules 144 bytes
  ./src/index.js 89 bytes [built] [code generated]
  ./src/App.js 55 bytes [built] [code generated]
webpack 5.68.0 compiled successfully in 114 ms

```

Our application code can be found at the end of the bundle file in a rather obscure format:



```
eval("__webpack_require__\.r(__webpack_exports__); \n/* harmony import */ var _App\n__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(/*! ./App */ \"./src/App.js\n\"); \n\n\nconst hello = name => { \n  console.log(`hello ${name}`) \n} \n\n\n(0, _App_\n__WEBPACK_IMPORTED_MODULE_0__[\"default\"]()) \n\n\n/# sourceMappingURL=webpack-\nos77./src/index.js?");
```

```
Configuration file
```

Let's take a closer look at the contents of our current `webpack.config.js` file:

```
const path = require('path')

const config = () => {
  return {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'main.js'
    }
  }
}

module.exports = config
```

Файл конфигурации написан на JavaScript, а функция, возвращающая объект конфигурации, экспортируется с помощью синтаксиса модуля Node.

Наше минимальное определение конфигурации практически объясняет само себя. Свойство `entry` объекта `configuration` указывает файл, который будет служить точкой входа для объединения приложения.

Свойство `__dirname` определяет местоположение, в котором будет храниться связанный код. Целевой каталог должен быть определен как *абсолютный путь*, который легко создать с помощью метода `path.resolve`. Мы также используем `__dirname` это переменная в Node, которая хранит путь к текущему каталогу.

## Связывание React

Далее давайте преобразуем наше приложение в минималистичное приложение React. Установим необходимые библиотеки.:

И давайте превратим наше приложение в приложение React, добавив знакомые определения в файл `index.js`:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
```

Мы также будем внести следующие изменения в *App.js* файл:

```
import React from 'react' // we need this now also in component files
```

Нам по-прежнему нужен файл `build/index.html`, который будет служить «главной страницей» нашего приложения и загружать наш объединенный код JavaScript с помощью тега `script`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
```

```
<title>React App</title>
</head>
<body>
  <div id="root"></div>
  <script type="text/javascript" src="./main.js"></script>
</body>
</html>
```

Когда мы объединяем наше приложение, мы сталкиваемся со следующей проблемой:

```
ERROR in ./src/App.js 2:2
Module parse failed: Unexpected token (2:2)
You may need an appropriate loader to handle this file type, currently no loader
s are configured to process this file. See https://webpack.js.org/concepts#loaders
| const App = () => (
>   <div>hello webpack</div>
| )
|
@ ./src/index.js 1:0-24 7:0-3

webpack 5.68.0 compiled with 1 error in 121 ms
```

## Загрузчики

В сообщении об ошибке webpack говорится, что нам может понадобиться соответствующий загрузчик *грузчик* для правильной компоновки файла *App.js*. По умолчанию webpack умеет работать только с обычным JavaScript. Хотя мы, возможно, и не знали об этом, мы используем [JSX](#) для отображения наших представлений в React. Чтобы проиллюстрировать это, следующий код не является обычным JavaScript:

```
const App = () => {
  return (
    <div>
      hello webpack
    </div>
  )
}
```

Копировать

Синтаксис, использованный выше, взят из JSX и предоставляет нам альтернативный способ определения элемента React для тега HTML *div*.

Мы можем использовать [загрузчики](#) для информирования webpack о файлах, которые необходимо обработать, прежде чем они будут собраны в пакет.

Давайте настроим загрузчик для нашего приложения, который преобразует код JSX в обычный JavaScript:

```
const path = require('path')

const config = () => {
  return {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'main.js'
    },
    module: {
      rules: [
        {
          test: /\.js$/,
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-react'],
          },
        },
      ],
    },
  },
}

module.exports = config
```

Копировать

Загрузчики определены в свойстве *module* в *Правила* массив.

Определение единого загрузчика состоит из трех частей:



```
{
  test: /\.js$/,
  loader: 'babel-loader',
  options: {
    presets: ['@babel/preset-react']
  }
}
```

[Копировать](#)

Свойство `test` указывает, что загрузчик предназначен для файлов, имена которых оканчиваются на `.js`. Свойство `loader` указывает, что обработка этих файлов будет выполняться с помощью [babel-loader](#). Свойство `options` используется для указания параметров загрузчика, которые настраивают его функциональность.

Давайте установим загрузчик и необходимые ему пакеты в качестве *зависимости для разработки*:

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

[Копировать](#)

Теперь объединение приложения в пакет будет успешным.

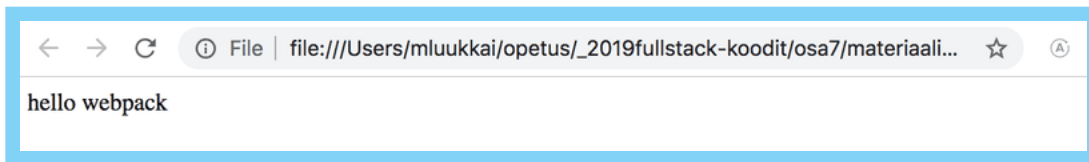
Если мы внесём некоторые изменения в компонент `App` и посмотрим на объединённый код, то заметим, что объединённая версия компонента выглядит так:

```
const App = () =>
  _react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement(
    'div',
    null,
    'hello webpack'
  )
```

[Копировать](#)

Как мы видим из приведенного выше примера, элементы React, которые были написаны на JSX, теперь создаются с помощью обычного JavaScript с использованием функции `React.createElement`.

Вы можете протестировать встроенное приложение, открыв файл `build/index.html` с помощью функции *открыть файл* в вашем браузере:



Стоит отметить, что если в исходном коде приложения используется *async/await*, то в некоторых браузерах браузер ничего не отобразит. [Поиск в Google](#) сообщения об ошибке в консоли прольет свет на эту проблему. Поскольку [предыдущее решение](#) устарело, теперь нам нужно установить еще две недостающие зависимости, а именно [core-js](#) и [regenerator-runtime](#):

```
npm install core-js regenerator-runtime
```

[Копировать](#)

Вам нужно импортировать эти зависимости в начало файла `index.js`:

```
import 'core-js/stable/index.js'
import 'regenerator-runtime/runtime.js'
```

[Копировать](#)

Наша конфигурация содержит почти все, что нам нужно для разработки React.

## Транспайлеры

Процесс преобразования кода из одной формы JavaScript в другую называется [транспилицией](#). Общее определение этого термина — компиляция исходного кода путём его преобразования из одного языка в другой.

Используя конфигурацию из предыдущего раздела, мы *преобразуем* код, содержащий JSX, в обычный JavaScript с помощью [babel](#), который в настоящее время является самым популярным инструментом для этой задачи.

Как упоминалось в первой части, большинство браузеров не поддерживают новейшие функции, появившиеся в ES6 и ES7, и по этой причине код обычно преобразуется в версию JavaScript, реализующую стандарт ES5.

Процесс транспиляции, выполняемый Babel, определяется с помощью [плагинов](#). На практике большинство разработчиков используют готовые [пресеты](#), которые представляют собой группы предварительно настроенных плагинов.

В настоящее время мы используем предустановку [@babel/preset-react](#) для транспиляции исходного кода нашего приложения:

```
{
  test: /\.js$/,
  loader: 'babel-loader',
```

[Копировать](#)

```
options: {
  presets: [ '@babel/preset-react' ]
}
```

Давайте добавим плагин [@babel/preset-env](#), который содержит всё необходимое для преобразования кода, использующего все новейшие функции, в код, совместимый со стандартом ES5:

```
{
  test: /\.js$/,
  loader: 'babel-loader',
  options: {
    presets: [ '@babel/preset-env', '@babel/preset-react' ]
  }
}
```

Копировать

Давайте установим пресет с помощью команды:

```
npm install @babel/preset-env --save-dev
```

Копировать

Когда мы переносим код, он преобразуется в старомодный JavaScript. Определение преобразованного компонента *приложения* выглядит следующим образом:

```
var App = function App() {
  return _react2.default.createElement('div', null, 'hello webpack')
};
```

Копировать

Как мы можем видеть, переменные объявляются с помощью ключевого слова `var`, поскольку JavaScript ES5 не понимает ключевое слово `const`. Функции со стрелками также не используются, поэтому в определении функции использовалось ключевое слово `function`.

## CSS

Давайте добавим CSS в наше приложение. Давайте создадим новый файл *src/index.css*:

```
.container {
  margin: 10px;
  background-color: #dee8e4;
}
```

Копировать

Тогда давайте использовать стиль в компоненте *App*:

```
const App = () => {
  return (
    <div className="container">
      hello webpack
    </div>
  )
}
```

Копировать

И мы импортируем стиль в *index.js* файл:

```
import './index.css'
```

Копировать

Это приведет к прерыванию процесса переноса:



```
ERROR in ./src/index.css 1:0
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type, currently no loader
s are configured to process this file. See https://webpack.js.org/concepts#loaders
> .container {
|   margin: 10;
|   background-color: #dee8e4;
|   @ ./src/index.js 4:0-21
```

При использовании CSS мы должны использовать загрузчики [css](#) и [style](#):

```
{
  rules: [
    {
      test: /\.js$/,
      loader: 'babel-loader',
      options: {
        presets: ['@babel/preset-react', '@babel/preset-env'],
      },
    },
    {
      test: /\.css$/,
      use: ['style-loader', 'css-loader'],
    },
  ],
}
```

Копировать

Задача [загрузчика css](#) заключается в загрузке CSS файлов, а задача [загрузчика стилей](#) заключается в создании и внедрении элемента *style*, который содержит все стили приложения.

При такой конфигурации определения CSS включены в *main.js* файл приложения. По этой причине нет необходимости отдельно импортировать стили CSS в основной *index.html* файл.

При необходимости CSS приложения также можно сгенерировать в отдельный файл, используя [mini-css-extract-plugin](#).

Когда мы устанавливаем загрузчики:

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

Копировать

Объединение снова пройдет успешно, и приложение получит новые стили.

## Webpack-dev-сервер

Текущая конфигурация позволяет разрабатывать наше приложение, но рабочий процесс ужасен (настолько, что напоминает процесс разработки на Java). Каждый раз, когда мы вносим изменения в код, нам приходится собирать его и обновлять браузер, чтобы протестировать.

[Webpack-dev-server](#) предлагает решение наших проблем. Давайте установим его с помощью команды:

```
npm install --save-dev webpack-dev-server
```

Копировать

Давайте определим npm-скрипт для запуска сервера разработки:

```
{
  // ...
  "scripts": {
    "build": "webpack --mode=development",
    "start": "webpack serve --mode=development"
  },
  // ...
}
```

Копировать

Давайте также добавим новое свойство *devServer* к объекту конфигурации в *webpack.config.js* файле:

```
const config = {
  entry: './src/index.js',
```

Копировать





```
output: {
  path: path.resolve(__dirname, 'build'),
  filename: 'main.js',
},
devServer: {
  static: path.resolve(__dirname, 'build'),
  compress: true,
  port: 3000,
},
// ...
};
```

Команда `npm start` теперь запустит сервер разработки через порт 3000, что означает, что наше приложение будет доступно по ссылке <http://localhost:3000> в браузере. Когда мы вносим изменения в код, браузер автоматически обновляет страницу.

Процесс обновления кода происходит быстро. Когда мы используем сервер разработки, код не упаковывается обычным способом в *main.js* файл. Результат объединения существует только в памяти.

Давайте расширим код, изменив определение компонента *приложения*, как показано ниже:

```
import React, { useState } from 'react'
import './index.css'

const App = () => {
  const [counter, setCounter] = useState(0)

  return (
    <div className="container">
      hello webpack {counter} clicks
      <button onClick={() => setCounter(counter + 1)}>
        press
      </button>
    </div>
  )
}

export default App
```

Копировать

Приложение работает отлично, и рабочий процесс разработки довольно плавный.

## Исходные карты

Давайте извлечем обработчик кликов в его собственную функцию и сохраним предыдущее значение счетчика в его собственном состоянии *values*:

```
const App = () => {
  const [counter, setCounter] = useState(0)
  const [values, setValues] = useState()

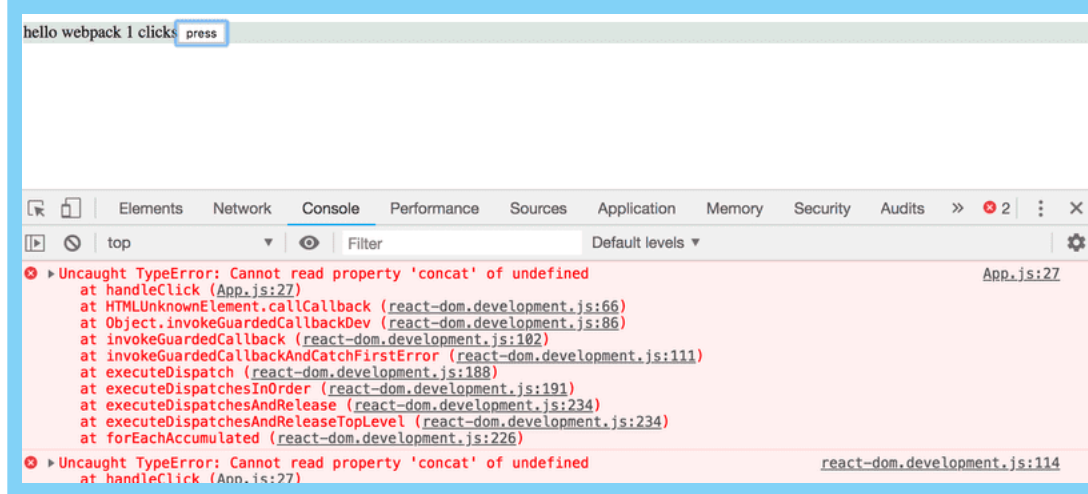
  const handleClick = () => {
    setCounter(counter + 1)
    setValues(values.concat(counter))
  }

  return (
    <div className="container">
      hello webpack {counter} clicks
      <button onClick={handleClick}>
        press
      </button>
    </div>
  )
}
```

Копировать

Приложение больше не работает, и консоль отобразит следующую ошибку:



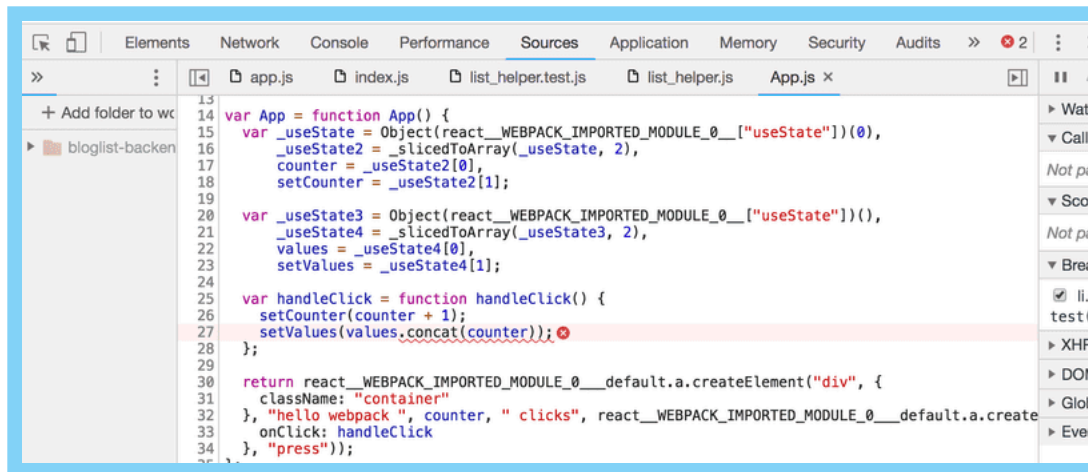


Мы знаем, что ошибка в методе onClick, но если бы приложение было немного больше, сообщение об ошибке было бы довольно сложно отследить:

App.js:27 Необработанная ошибка типа: невозможно прочитать свойство «concat» неопределённого значения в handleClick (App.js:27)

копировать

Расположение ошибки, указанное в сообщении, не соответствует фактическому расположению ошибки в нашем исходном коде. Если мы нажмём на сообщение об ошибке, то заметим, что отображаемый исходный код не похож на код нашего приложения:



Конечно, мы хотим видеть наш фактический исходный код в сообщении об ошибке.

К счастью, исправить это сообщение об ошибке довольно просто. Мы попросим webpack сгенерировать так называемую карту исходного кода для пакета, которая позволяет сопоставлять ошибки, возникающие во время выполнения пакета, с соответствующими частями исходного кода.

Карту источника можно сгенерировать, добавив в объект конфигурации новое свойство devtool со значением «source-map»:

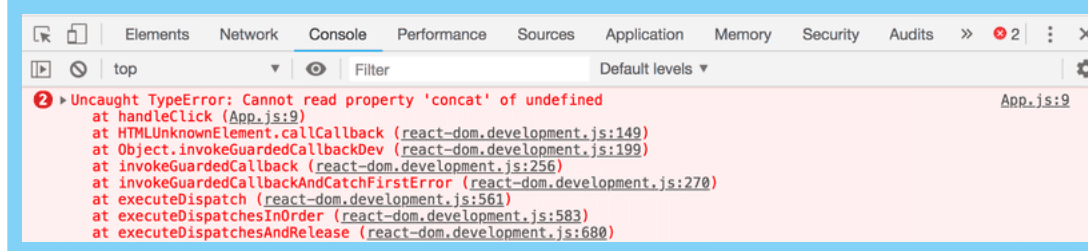
```
const config = {
  entry: './src/index.js',
  output: {
    // ...
  },
  devServer: {
    // ...
  },
  devtool: 'source-map',
  // ..
};
```

Копировать

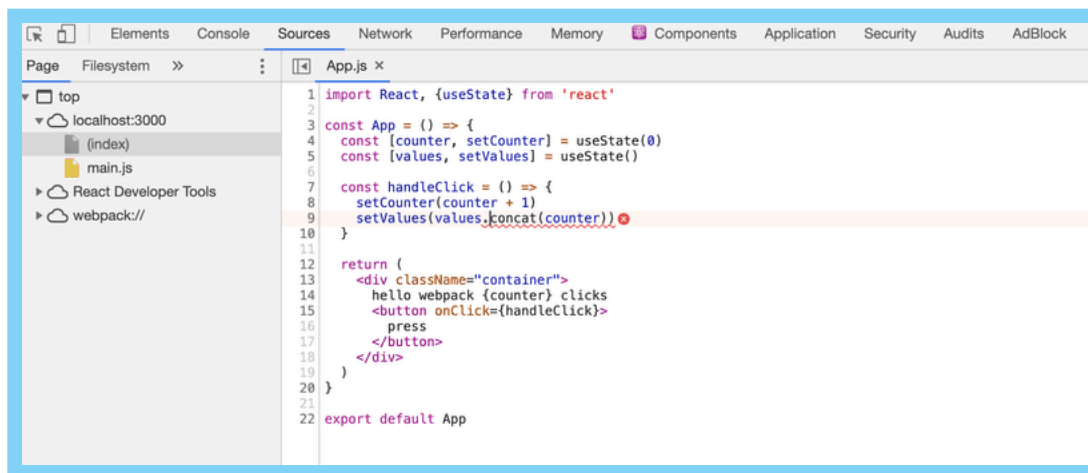
Webpack необходимо перезапускать при внесении изменений в его конфигурацию. Также можно настроить Webpack на отслеживание изменений, но в этот раз мы этого делать не будем.

Сообщение об ошибке теперь стало намного лучше

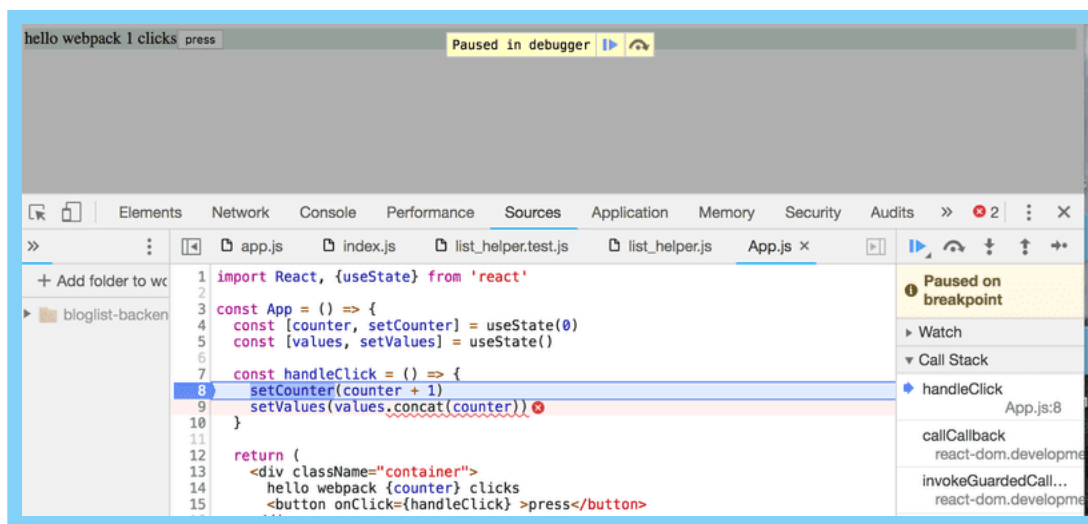




поскольку это относится к написанному нами коду:



Генерация исходной карты также позволяет использовать отладчик Chrome:



Давайте исправим ошибку, инициализировав состояние значений в виде пустого массива:

```
const App = () => {  
  const [counter, setCounter] = useState(0)  
  const [values, setValues] = useState([])  
  // ...  
}
```

Копировать

## Упрощение кода

При развертывании приложения в рабочей среде мы используем *main.js* пакет кода, который генерируется webpack. Размер *main.js* файла составляет 1009487 байт, хотя наше приложение содержит всего несколько строк нашего кода. Большой размер файла объясняется тем, что пакет также содержит исходный код для всей библиотеки React. Размер пакета кода имеет значение, поскольку браузер должен загрузить код при первом использовании приложения. При высокоскоростном подключении к Интернету 1009487 байт не являются проблемой, но если бы мы продолжали добавлять больше внешних зависимостей, скорость загрузки могла бы стать проблемой, особенно для мобильных пользователей.

Если мы проверим содержимое файла bundle, то заметим, что его можно было бы значительно оптимизировать с точки зрения размера файла, удалив все комментарии. Нет смысла вручную оптимизировать эти файлы, поскольку для этой работы существует множество инструментов.

Процесс оптимизации файлов JavaScript называется *минификацией*. Одним из ведущих инструментов, предназначенных для этой цели, является [UglifyJS](#).

Начиная с версии 4 webpack, плагин минимизации не требует дополнительной настройки для использования. Достаточно изменить прт-скрипт в файле *package.json*, чтобы указать, что webpack будет выполнять пакетирование кода в *производственном* режиме:



```
{
  "name": "webpack-part7",
  "version": "0.0.1",
  "description": "practising webpack",
  "scripts": {
    "build": "webpack --mode=production",
    "start": "webpack serve --mode=development"
  },
  "license": "MIT",
  "dependencies": {
    // ...
  },
  "devDependencies": {
    // ...
  }
}
```

Копировать

Когда мы снова объединяем приложение, размер полученного *main.js* значительно уменьшается:

```
$ ls -l build/main.js
-rw-r--r--  1 mluukkai  ATKK\hyad-all  227651 Feb  7 15:58 build/main.js
```

Копировать

Результат процесса минимизации напоминает старый добрый код на C: все комментарии и даже лишние пробелы и символы новой строки удалены, имена переменных заменены одним символом.

```
function h(){if(!d){var e=u(p);d=!0;for(var t=c.length;t;){for(s=c,c=[];++f<t;)s&&s[f].run();f=-1,t=c.length}s=null,d=!1,function(e){if(o===clearTimeout)return o=clearTimeout(e);if((o===1||!o)&&clearTimeout)return o=clearTimeout,clearTimeout(e);try{o(e)}catch(t){try{return o.call(null,e)}catch(t){return o.call(this,e)}}}(e)}a.nextTick=function(e){var t=new Array(arguments.length-1);if(arguments.length>1)
```

## Разработка и производственная конфигурация

Далее давайте добавим серверную часть в наше приложение, изменив теперь уже привычную серверную часть приложения note.

Давайте сохраним следующее содержимое в файле *db.json*:

```
{
  "notes": [
    {
      "important": true,
      "content": "HTML is easy",
      "id": "5a3b8481bb01f9cb00ccb4a9"
    },
    {
      "important": false,
      "content": "Mongo can save js objects",
      "id": "5a3b920a61e8c8d3f484bdd0"
    }
  ]
}
```

Копировать

Наша цель - настроить приложение с помощью webpack таким образом, чтобы при локальном использовании приложение использовало json-сервер, доступный через порт 3001, в качестве серверной части.

Затем пакетный файл будет настроен на использование серверной части, доступной по URL-адресу <https://notes2023.fly.dev/api/notes>.

Мы установим *axios*, запустим json-сервер, а затем внесём необходимые изменения в приложение. Чтобы всё изменить, мы будем получать заметки из бэкенда с помощью нашего [пользовательского хука](#) `useNotes` :

```
import React, { useState, useEffect } from 'react'
import axios from 'axios'
const useNotes = (url) => {
  const [notes, setNotes] = useState([])
  useEffect(() => {
    axios.get(url).then(response => {
      setNotes(response.data)
    })
  }, [url])
  return notes
}

const App = () => {
  const [counter, setCounter] = useState(0)
  const [values, setValues] = useState([])
  const url = 'https://notes2023.fly.dev/api/notes'
  const notes = useNotes(url)
```

Копировать



```

const handleClick = () => {
  setCounter(counter + 1)
  setValues(values.concat(counter))
}

return (
  <div className="container">
    hello webpack {counter} clicks
    <button onClick={handleClick}>press</button>
    <div>{notes.length} notes on server {url}</div>
  </div>
)
}
}

export default App

```

Адрес внутреннего сервера в настоящее время жестко задан в коде приложения. Как мы можем контролируемым образом изменить адрес, чтобы он указывал на рабочий внутренний сервер, когда код собран для производства?

Функция настройки Webpack имеет два параметра, *env* и *argv*. Мы можем использовать последний, чтобы узнать, *режим* определен в скрипте прм:

```

const path = require('path')

const config = (env, argv) => {
  console.log('argv.mode:', argv.mode)
  return {
    // ...
  }
}

module.exports = config

```

Копировать

Теперь, если мы захотим, мы можем настроить Webpack на работу по-разному в зависимости от того, установлена ли операционная среда приложения или *режим* на производство или разработку.

Мы также можем использовать [DefinePlugin](#) в webpack для определения *глобальных констант по умолчанию*, которые можно использовать в скомпилированном коде. Давайте определим новую глобальную константу *BACKEND\_URL*, которая принимает разные значения в зависимости от среды, для которой компилируется код:

```

const path = require('path')
const webpack = require('webpack')

const config = (env, argv) => {
  console.log('argv', argv.mode)

  const backend_url = argv.mode === 'production'
    ? 'https://notes2023.fly.dev/api/notes'
    : 'http://localhost:3001/notes'

  return {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'main.js'
    },
    devServer: {
      static: path.resolve(__dirname, 'build'),
      compress: true,
      port: 3000,
    },
    devtool: 'source-map',
    module: {
      // ...
    },
    plugins: [
      new webpack.DefinePlugin({
        BACKEND_URL: JSON.stringify(backend_url)
      })
    ]
  }
}

module.exports = config

```

Копировать

Глобальная константа используется в коде следующим образом:

```

const App = () => {
  const [counter, setCounter] = useState(0)
  const [values, setValues] = useState([])
  const notes = useNotes(BACKEND_URL)

```

Копировать



```
// ...
return (
  <div className="container">
    hello webpack {counter} clicks
    <button onClick={handleClick} >press</button>
    <div>{notes.length} notes on server {BACKEND_URL}</div>
  </div>
)
}
```

Если конфигурация для разработки и производства сильно отличается, возможно, стоит [разделить конфигурации](#) этих двух процессов на отдельные файлы.

Теперь, если приложение запускается с помощью команды `npm start` в режиме разработки, оно получает заметки с адреса <http://localhost:3001/notes>. Версия, запускаемая с помощью команды `npm run build`, использует адрес <https://notes2023.fly.dev/api/notes> для получения списка заметок.

Мы можем локально проверить собранную рабочую версию приложения, выполнив следующую команду в каталоге `build`:

```
npx static-server
```

Копировать

По умолчанию приложение будет доступно по адресу <http://localhost:9080>.

## Полифилл

Наше приложение готово и работает со всеми относительно новыми версиями современных браузеров, кроме Internet Explorer. Причина в том, что из-за `axios` наш код использует [Promises](#), а ни одна из существующих версий IE их не поддерживает:


















Browser compatibility <a href="#">↗</a>														<a href="#">Update compatibility data on GitHub</a>	
Desktop							Mobile								
Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js		
Basic support															
32	Yes	29	No	19	8	4.4.3	32	Yes	29	Yes	8	Yes	0.12		
Promise() constructor															
32	Yes	29 ★	No	19	8 ★	4.4.3	32	Yes	29 ★	Yes	8 ★	Yes	0.12 ★		
all															
32	Yes	29	No	19	8	4.4.3	32	Yes	29	Yes	8	Yes	0.12		
prototype															
32	Yes	29	No	19	8	4.4.3	32	Yes	29	Yes	8	Yes	0.12		

В стандарте есть много других вещей, которые IE не поддерживает. Что-то столь безобидное, как метод [find](#) для массивов JavaScript, превосходит возможности IE:



# Browser compatibility

[Update compatibility data on GitHub](#)

													
 Chrome	 Edge	 Firefox	 Internet Explorer	 Opera	 Safari	 Android webview	 Chrome for Android	 Edge Mobile	 Firefox for Android	 Opera for Android	 Safari on iOS	 Samsung Internet	 Node.js
Basic support													
45	Yes	25	No	32	8	Yes	Yes	Yes	4	Yes	8	Yes	4.0.0

В таких ситуациях недостаточно транспилировать код, поскольку транспилирование просто преобразует код из более новой версии JavaScript в более старую с более широкой поддержкой браузера. IE понимает Promises синтаксически, но просто не реализовал их функциональность. Свойство `find` массивов в IE просто *не определено*.

Если мы хотим, чтобы приложение было совместимо с IE, нам нужно добавить [polyfill](#), который представляет собой код, добавляющий недостающую функциональность старым браузерам.

Полизаполнения можно добавлять с помощью [webpack и Babel](#) или установив одну из многих существующих библиотек polyfill.

Библиотека polyfill, предоставляемая [promise-polyfill](#), проста в использовании. Нам просто нужно добавить следующее в существующий код нашего приложения:

```
import PromisePolyfill from 'promise-polyfill'

if (!window.Promise) {
  window.Promise = PromisePolyfill
}
```

Копировать

Если глобальный объект `Promise` не существует, что означает, что браузер не поддерживает Promises, полизаполненное обещание сохраняется в глобальной переменной. Если обещание polyfilled реализовано достаточно хорошо, остальной код должен работать без проблем.

С одним исчерпывающим списком существующих полифиллов можно ознакомиться [здесь](#).

Совместимость браузеров с различными API можно проверить на <https://caniuse.com> или веб-сайте Mozilla.

[Предлагаем изменения в материале](#)

Часть 7с

[Предыдущая часть](#)

Часть 7е

[Следующая часть](#)

About course

Course contents

FAQ

Partners

Challenge





UNIVERSITY OF HELSINKI







